Universidade do Porto
Faculdade de Engenharia
FEUP

Timing Analysis of an Embedded Architecture for a Real-Time Power Line Communications Network

António Manuel de Sousa Barros

Dissertação submetida para satisfação parcial
dos requisitos do grau de mestre
em
Eng. Electrotécnica e de Computadores
(Área de especialização de Telecomunicações)

Dissertação realizada sob a supervisão de
Professor Doutor António Miguel Pimenta Monteiro
do Departamento de Engenharia Informática
da Faculdade de Engenharia da Universidade do Porto
e
Doutor Luís Miguel Rosário Pinho
do Centro de I&D CISTER
do Instituto Superior de Engenharia do Porto

Porto, Setembro de 2008

Abstract

Although voice and data transmission over power-lines is not a recent technology, the use of this medium to support time-constrained communications is still a subject of research. In the scope of an European R&D project (REMPLI - Real-time Energy Management via Powerlines and Internet) a new communication architecture was proposed, based on a two-level hierarchic system (medium-voltage and low-voltage). In order to support the required network previsibility and efficiency requirements, one of the project partners developed a new embedded architecture based on the Hyperstone's processor HyNet32XS interfacing with a power-line chipset (DLC-2C). The uClinux operating system was ported for this architecture to support the development of the new software architecture.

Due to the architecture characteristics and the (multi)master/slave behaviour of the medium access layer, new packet routing and scheduling protocols – fundamental for the correct functioning of the network – were specified. In this context, the main objective of this dissertation is to study the timing behaviour of this architecture, focusing in the behaviour of the routing and scheduling protocols. Particularly important is the validation of the timing behaviour of these new protocols.

Keywords: Real-time networks, embedded systems, embedded Linux, power-line communications.

Sumário

Embora a transmissão de voz e dados em redes eléctricas (Power-line Communication) não seja uma tecnologia recente, a sua utilização para suportar comunicação com requisitos temporais é ainda foco de investigação. Para esse efeito, no âmbito de um projecto europeu (REMPLI - Real-time Energy Management via Powerlines and Internet) foi proposta uma nova arquitectura de comunicação num sistema hierárquico de dois níveis (média tensão e baixa tensão). Devido às características da arquitectura e ao comportamento (multi)mestre-escravo da camada de acesso ao meio, foram especificados novos protocolos de encaminhamento e escalonamento de pacotes, fundamentais para o correcto funcionamento da rede.

De forma a suportar os requisitos de previsibilidade e eficiência da rede, foi desenvolvida por um dos parceiros do Projecto uma nova arquitectura de processamento embebida baseada no processador HyNet32XS da Hyperstone e no módulo de comunicações sobre linhas de transmissão eléctrica DLC-2C. Para suporte, o sistema operativo uClinux foi portado para esta arquitectura.

A validação das características temporais desta arquitectura é um factor de relevância na validação dos resultados do projecto. Mais ainda, a validação do comportamento temporal dos novos protocolos de encaminhamento e escalonamento de pacotes é um factor imprescindível para o sucesso do projecto. Neste âmbito, o objectivo desta dissertação é o de estudar o comportamento temporal desta arquitectura, especificamente para suportar os protocolos de encaminhamento e escalonamento desenvolvidos no projecto.

Palavras-chave: Redes de comunicações de tempo-real, sistemas embebidos, comunicações via linha de transmissão eléctrica, *embedded linux*.

Acknowledgements

It is impossible to refer every person and institution worthy of my acknowledgement for helping me carry this dissertation through, but for the exceptional value of their support, some persons are impossible not to mention.

First of all I would like to express my gratitude to my supervisors: Miguel Pinho for giving me the opportunity to participate in this challenging project, his outstanding supervision, the invaluable advice, the permanent pressure to never miss deadlines, and Professor Miguel Monteiro for his interest in my work, his extraordinary generosity and helpfulness in the guidance of this dissertation and for all the support given during the period this work lasted.

I want to thank all the people in IPP-HURRAY! Research Group, at the School of Engineering of the Polytechnic Institute of Porto, for creating such a challenging and motivating environment for technology innovation. A special word for my team colleagues Filipe Pacheco and Luís Marques for making team work so easy, pleasant and stimulating!

I would like to thank Maksim Lobashov from Institute of Computer Technology at the Vienna University of Technology, for his collaboration during integration.

I would also like to thank my parents for the constant love and support during all my life, and encouragement to never stop improving myself.

And, finally, a very special thanks to Dulce, for all the love and encouragement, and clearing my skies whenever they are dark and clouded!

Contents

List of Figures

List of Tables

List of acronyms

ABI	Application Binary Interface
ACE	Actuator Control Electronics
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
CAN	Controller Area Network
CENELEC	European Committee for Electrotechnical Standardization
COTS	Commercial-Of-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DeMux	Demultiplexer/Multiplexer
DSP	Digital Signal Processor
ECU	Electronic Control Unit
EN	European Standard
FBW	Fly By Wire
FEC	Forward Error Correction
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
IC	Integrated Circuit
IEC	International Electrotechnical Commission
IO	Input/Output

IP	Internet Protocol
IPCTransID	Inter Process Communication Transaction Identifier
ISA	Instruction Set Architecture
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ITU	International Telecommunication Union
IVA	Imaging and Video Accelerator
LLC	Logical Link Control
MAC	Media Access Control
NFS	Network File System
NL	Network Layer
NLI	Network Layer Interface
OFDM	Orthogonal Frequency Division Multiplexing
OO	Object Oriented
OS	Operating System
OSI	Open Systems Interconnection
PFC	Primary Flight Computer
PDU	Protocol Data Unit
PLC	Power-Line Communication
POSIX	Portable Operating System Interface
POTS	Plain Old Telephone Service
QM	Queue Manager
RCI	REMPLI Communication Interface
RCIM	RCI Manager
REMPLI	Real-time Energy Management via Power-Lines and Internet
RISC	Reduced Instruction Set Computer
RS	Recommended Standard
RTOS	Real-Time Operating System
RUSN	REMPLI Unique Serial Number

SCADA	Supervisory Control And Data Acquisition
SoC	System-on-a-Chip
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TFTP	Trivial File Transfer Protocol
TL	Transport Layer
TRM	Transport Route Manager
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

The use of the electrical power distribution grid as a communications medium for grid operation tasks – such as ripple control – started several decades ago, although classical applications were not too much demanding [1]. The higher dissemination of the electrical grid access, as compared to other native communication utilities, led to a recent trend of using it as transport medium of digital communications. However, the hostile physical characteristics of this medium for high data rates and frequency bands, verified on field tests, has reduced the will to use the electrical distribution grid for broadband applications.

Current energy distribution and management involves many tasks – such as load balancing, grid monitoring and reconfiguration – that require data acquisition, along with a supporting communication infrastructure. These tasks can be more effective if consumption measurement resolution can be spatially and temporally increased [1]. Native communication networks, such as Plain Old Telephone Service (POTS) and General Packet Radio Service (GPRS) are already used for Supervisory Control And Data Acquisition (SCADA) but the cost of these solutions often impair the dissemination of remote monitor and actuator equipment, which is only installed in central points, like transformer and supply stations [2]. Increasing the spatial resolution of the energy measurements would imply increased recurring costs if the same native communication media continue to be used. The omnipresent feature of power-line and the savings provided by an already deployed network are appealing enough for electricity distribution companies to push applications supported by the electrical power grid. Nevertheless, this medium is only effective if it guarantees the necessary data rates to provide the required timing resolution of remote measurements.

The Real-time Energy Management via Power-Lines and Internet (REMPLI) project (European program NNE5-2001-00825) [3] objective was to develop a distributed infrastructure suitable for real-time monitoring and control of energy consumption. In the short-term, this communication infrastructure should support the operation of energy distribution networks and, in the medium-term it may be available to support additional services, even if some are yet not foreseen [4].

One of the primary goals of REMPLI is to provide the registration of energy consumption, not only from meters located at the costumer's premises but also from meters located at branching points of the distribution network, with higher timing resolution. Besides the cost saving advantages from automated remote meter reading and billing, the collected data can be statistically processed to identify fine-grained consumption patterns which are fundamental for effective planning of energy distribution networks. Real- or near real-time monitoring of the energy consumption can also permit the energy supplier to apply tariff schemes that condition the costumers to certain behaviours that may lead to a

more rational use of the distribution network[1]. In a liberalised energy market, the REMPLI system can also be an useful tool for costumers and energy suppliers/distributors: as each meter can be remotely accessible to any energy company in the market, the costumer is no longer attached to a single company and can chose the contract that best suits the intended usage profile. Synchronous data collected from distributed meters can also be used in energy loss detection and faster detection of its source (leakage, short circuit or fraud).

Another primary goal of REMPLI is to provide a platform to support the management of distributed networks. The multiple types of data collected and transported by the REMPLI communication infrastructure permits the development of advanced energy management functions that can be employed by using remote/distributed control equipment.

Besides the immediate goals referred, the REMPLI system has an open-design that is scalable and allows the addition of new services that can be built on top of the system interface provided by REMPLI.

The underlying communication system is based on combined Power-Line Communication (PLC) and Internet network. The use of PLC technology is justified by the fact that, on one hand, most of the energy metering and control equipments are already wired to the low-voltage and/or mid-voltage electric networks, and, on the other hand, it is difficult to access certain equipments using wireless technology as they are often located in closed environments and/or with metallic obstacles (reinforced concrete walls and tubes).

In a wide-area PLC network, transmitting a packet from a source to a not immediately reachable destination requires the packet relay of intermediate nodes (repeaters). However, considering the dynamic topology change and impossible prediction of the power-line attenuation, repeaters cannot be statically configured [5]. How to design the efficient routing protocols for dynamically adapting the power-line circumstances and shortening the transmission time under stringent bandwidth limitation consists thus a challenging problem.

The REMPLI communication stack includes the *physical*, the *network*, the *transport* and the *application* layers [6]. The network layer implements a master/slave time division network, and short-distance routing mechanisms supported by node status tables. One network segment can hold multiple masters, operating in different time slots or different frequency bands, but a slave can be simultaneously connected to multiple masters if they operate in the same frequency band. The time division multiple access demands very strict timing requirements from the network layer.

The transport layer handles the end-to-end communication, providing inter-network routing, address translation, data fragmentation/assembling, request/response pairing and an alarm service. Masters know in real-time the status of associated slaves and calculate all possible paths for all active slaves. For each available path it is also calculated the link quality, taking in accounting the average delay and error rate. Besides the routing tasks, the transport layer is also responsible to adapt the variable-size application data to the network fixed-size packets, performing fragmentation at the source and assembling at the destination. Fragmentation can be a complex operation on bridges, when the two network segments the bridge connects have different packet sizes and the bridge cannot simply act as a repeater, but has to perform re-fragmentation instead.

When an application requests the transport layer to perform some task, the transport layer will later inform the application about the result of the request processing within a specific time limit. The timing requirements of the transport layer are not so strict as the timing requirements of the network layer but since the underlying communication network is narrowband and the available data rate is a valuable resource, the performance of the transport layer must be higher than the slowest component

[1] Smoothing the production and network load profiles.

(the network), in order to not impair the whole system performance. In this way, the analysis of the timing requirements is absolutely necessary to verify the validity of this architecture.

1.1 Research objectives

The main objective of this dissertation is to analyse the timing behaviour of this set of complex protocols in the architecture's Transport Layer. Due to the complex functionality required from the REMPLI Transport Layer (TL) and its less restrictive time constrains[2], this protocol layer was unusually developed as a user-space application. Nevertheless, and considering the narrow bandwidth the communications medium has, it is not admissible that the software components introduce additional delays, therefore it is necessary to verify if the complexity of the REMPLI TL implementation does not transform the implementation of this layer in the bottleneck of the complete system.

1.2 Contributions

The specification of the REMPLI system introduces an increased complexity to what is common in the transport layer of generic communication protocol stacks. The routing and scheduling algorithms developed within the project were proven functional in a simulation environment, but a physical prototype must be implemented and validated in the field before a product being deployed.

This dissertation contributes to this overall goal providing:

1. an analysis of the REMPLI Transport Layer architecture;
2. a fully functional prototype of the REMPLI Transport Layer for the specific architecture developed within the project;
3. efficiency and predictability analysis of the REMPLI Transport Layer, in the embedded architecture developed during the project.

This prototype is to be deployed in a field test to verify the validity of the routing and scheduling protocols in real-life conditions, and should serve as basis of the final commercial product.

1.3 Structure of this dissertation

This dissertation is divided into 7 chapters, as follows.

Chapter 2 provides an overview of real-time embedded systems and development methodologies. In this context, the chapter also presents an overview of the REMPLI system and a description of communication and software architectures.

Chapter 3 gives an insight of the hardware and software environments in which the REMPLI system executes. The hardware was developed specifically for this system, and a Linux kernel version for microcontrollers without a memory management unit was ported for this architecture. Along with the kernel port, a toolchain was also ported in order to allow the development of kernel extensions, device drivers and applications.

[2] Compared with the time constrains associated with the Network Layer operation.

Chapter 4 then presents the details of the REMPLI Transport Layer, the main focus of this dissertation. The four main modules of this communication layer are introduced together with a detailed description of their functionality and of the services they provide.

Chapter 5 describes how the REMPLI TL was developed within the REMPLI architecture. This development started with simulation code which included the functionality of the four modules, and then was necessary to implement the mechanisms that allowed modules to execute concurrently and communicate both inside the layer and with the adjacent layers.

In real-time systems the timing behaviour is the top concern since the value of the computed result of the correct solution also depends on the time it is provided. Therefore, laboratory tests were conducted to assess the usefulness of the developed protocols. The results are presented and analysed in chapter 6.

Finally, this dissertation concludes with chapter 7, which summarizes the dissertation contributions and analyses its main conclusions. The chapter also identifies topics for future research that can be potentially explored.

Chapter 2

Context

2.1 Embedded systems

An *embedded system* is a computer that is implemented for a specific purpose, frequently to control a larger system in which the computer is inserted [7]. Embedded systems have an innumerable range of applications, like pocket media players, vehicle electronic stability control, industrial process control or rocket guidance systems. Unlike a general purpose computer which may perform a wide range of tasks, the hardware design of an embedded system takes in consideration the limited set of tasks that will be performed, so the components are selected to fit the requirements. This hardware design allows to reduce the final unit cost, which is very important in large scale applications or in mass-production commercial products.

Embedded computers have to interact with the physical elements of the surrounding system by means of interface devices. Figure 2.1 represents the general type of interfaces that can be found on an embedded system. *Sensors* acquire data that allows the computer to estimate the status of the system like, for instance, a gyroscope in a rocket guidance controller. *Actuators* allow the computer to modify the system status, like attitude jets used to maintain a rocket in the trajectory. Embedded computers have reduced or none human interface because they are mostly used in applications that require small or none human intervention. Communications interfaces often are available as serial port and/or network adapter and are useful to control or monitor the embedded computer activity, to cooperatively exchange data with other embedded systems[1] or even use remote disk servers. It is also common that embedded systems without user interface have a console application that allows remote user interaction from another computer.

The main component in an embedded system is the *microcontroller*, which encapsulates in a chip the Central Processing Unit (CPU) with memory and other peripherals like I/O interfaces, timers and analog-to-digital converters. Depending on the application field of the embedded system, the CPU can be a processor from 4-bit up to 64-bit architecture. Memory in microcontrollers are of two types: volatile and non-volatile. The non-volatile is used to store the software that the embedded system will execute and the current trend is to be based on flash memory because it allows simple reprogramming of the embedded system. The volatile memory is used to store the application data and, in some cases, the on-chip memory is sufficient for the task (like in the anti-lock braking system of a car vehicle), but in more demanding cases, the computer must use external volatile memory.

Most embedded computers are intended to work under real-time constraints. In a real-time system,

[1] Like the Electronic Stability Control system communicates with the Fuel Injection, Brake Control and Steering systems, whenever a vehicle is in risk of losing stability.

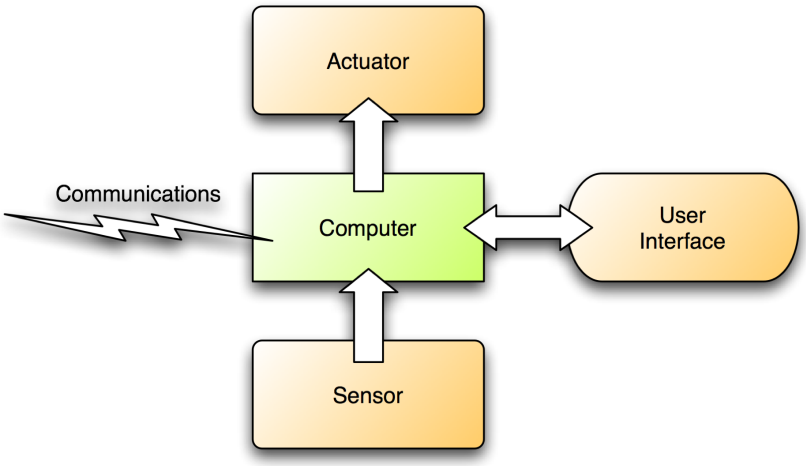

Figure 2.1: Typical interfaces on an embedded system.

an operation is considered valid only if it is supported by a correct calculation and is performed within a specific time period [8]. The strictness of the deadline of a real-time task can be divided in two categories [9]:

- A *hard real-time* task is a task that must be performed before the deadline, otherwise it will be valueless and will constitute a failure of the system like, for example, to activate the fire extinguishers in a transformer station when a fire is detected.
- A *soft real-time* task retains some value if performed after the deadline. For instance, when a remote household electricity meter reading takes longer than the specified timeout value, the request can be repeated without damage to the system, although the returned value is not as accurate as wanted.

Some real-time systems, like a nuclear power plant control system, have to cope with severe reliability requirements and any failure can be potentially catastrophic. A task that can lead to this situation if it misses the deadline is called *critical* and it must be assured that the task always makes its deadline, even under equipment failure or system workload. A combination of scheduling policies and equipment redundancy are employed to assure 100% that these critical tasks will never miss deadlines.

Depending on the size and complexity of the tasks, one of several embedded software architectures can be employed. For the least complex cases, a *control loop* calls routines that performs the required tasks. On a *interrupt controlled system*, each task routine is associated with a system interrupt and when the system receives one interrupt it triggers the execution of the respective interrupt handler. Under a *preemptive multitasking* architecture, tasks have access to the processor in controlled time slices. With more complex applications, an embedded system may use an operating system adapted for an embedded system, being controlled either by a microkernel or a monolithic kernel. The microkernel has reduced functionality which is complemented by processes executing in user-space. Oppositely, the monolithic kernel has a relatively large size but includes most of the facilities found in desktop systems which facilitates the development of software, at the expense of more computer resources and less predictability.

However, in more complex systems, such as REMPLI, the complexity and dimension of the system implies that the underlying embedded architecture is a full-fledged computer system. In this

case, embedded systems resort to more powerful hardware architectures and to adapted versions of general-purpose operating systems. There is, nevertheless, still the need to guarantee the hard real-time requirements of the control loops (and soft real-time requirements of the applications) and the interface to the surrounding system. This is one of the main challenges of the REMPLI architecture.

2.1.1 Embedded software development

The advances in semiconductor technologies and associated decrease of processor cost and size has widened the application range of embedded systems. In the past, an embedded system would be small and simple, or probably a composition of few subsystems with barely none interaction. However, currently, the quantity and complexity of applications for embedded systems is growing rapidly, and the increasing computational power of processors is leading designers to allocate more functionality to software. All summed up, new challenges for embedded software development are being experienced in recent years. As an example, embedded electronics are believed to be responsible for 90% of innovation in the automotive products being 80% of that in the area of software [10]. The weight of embedded systems is also increasing: in aeronautics 50% of the cost of development of a new aeroplane stands for embedded hardware and software; in the automotive industry, embedded systems stand for 35% of a vehicle development costs (almost 40% of it standing for software), as opposed to the traditional 25% for power-train.

Real-time software development has to deal with constraints that are not considered on hosted systems, such as hard timing constrains, limited hardware resources, selected hardware platform, reliability and safety factors, cost factors and time-to-market, which require different development techniques [11]. Furthermore, in safety-critical systems, the timing behaviour of the implemented application is closely related to the hardware environment (*e. g.* processor, Operating System (OS) or communications network/bus). The increase in complexity required from real-time systems is challenging the traditional embedded software development techniques and productivity is decreasing, which is a serious problem when facing the tight time-to-market that are normal in some industrial fields.

In broad terms, a development process can be divided into 6 phases: Analysis, Specification, System Design, Implementation, Integration and Production [12]. Although currently these phases are not strictly sequential, their roles and tasks still hold.

The development of a new product starts with the description of the overall system behaviour, a high-level abstraction of system functionality completely independent from an implementation point of view [13]. Functional and non-functional requirements define how the system is expected to perform.

Based on the requirements, a functional specification is realised defining what has to be implemented. The specification is an objective description of the functionality of the system that does not imply an implementation. A specification usually contains a set of constraints that must be met and a set of design criteria that should be observed.

In large systems, the functionality of the system is modularly decomposed into subsystems. These subsystems may be developed in-house, or contracted to suppliers (which is frequent in automotive and aeronautic industries), so each subsystem will have its own functional specification that will be delivered to the respective developing team.

The specification of a product or subsystem is the starting point for the designer to select an hardware platform and devise algorithms that meet the functional requirements. In this way, the functional requirements and the experience of the designer are important factors when deciding the system architecture. The designer must also decide on which functions should be implemented on

hardware and which functions should be programmed as software.

The hardware platform selection is based in the intersection of specification constraints and semiconductor characteristics: the processor must meet a minimum speed and the memory must meet a minimum size. In recent years, there is a trend for Integrated Circuit (IC) manufacturers to prefer the production of chips that will serve for multiple designs, in order to balance development costs with the number of unit sales, reducing the range of options available. For some applications, the designers also have available Commercial-Of-The-Shelf (COTS) platforms. These factors may lead to an over-designed platform, but for a smaller cost. An over-designed platform also allows software updates and/or the addition of new functionalities that extend the application [13].

The implementation is the process in which the designed solution will become a practical product. During this phase, the programming team writes the software code for the target hardware. Right from the begining of the implementation, the programming team is limited by the characteristics of the platform architecture selected by the designer. Traditionally, in order to achieve the most efficient code, embedded source code was written using a low-level programming language, such as assembly or C, which would be assembled/compiled to generate the program that would run in the processor architecture. This method is very effective for simple applications, but writing code so dependent of a specific platform raises several problems in most of current applications.

Software complexity. System designers are moving more and more functionality from hardware to software, claiming that software is more flexible to posterior corrections or extensions of functionality. This means that embedded software is being demanded to perform more functions of increasing complexity. Implementing extensive complex functionality in assembly on in ”low-level” C is prone to increase the number of misconception errors.

Software reusability. When the source code is too dependent of the specific platform architecture, it becomes very difficult to port it to a different platform architecture: this can happen even if the processor is replaced by another processor of the same family but with a modified Instruction Set Architecture (ISA). This issue is of particular importance when the life-cycle of the product is longer than the life-cycle of its components (hardware and software), and during production, warranty and after-sales assistance subsystems have to be replaced by newer technology [14]: *e. g.* during the life-cycle of an motorcar model, most electronic subsystems will be replaced by newer generation subsystems.

These issues have been addressed in recent years, in order to increase productivity of embedded software development and decrease the incidence of software errors. One of the most important steps is to create system standards that allow the independence of software from hardware components. A standardised Application Programming Interface (API) can hide diverse ISAs, providing a unified high-level interface to the hardware [14, 13]. This abstraction layer is supposed to include:

- a Real-Time Operating System (RTOS) that wraps the programmable cores and the memory subsystem, and provides a concurrency model;
- device drivers providing an interface to I/O subsystem services;
- a network communication service to provide applications access the network subsystem.

A practical example of this orientation is the Automotive Open systems Architecture (AUTOSAR) initiative [15]. Based on the Open Systems and the Corresponding Interfaces for Automotive Electronics (OSEK) specifications for an embedded operating system [16], the AUTOSAR defines a modular

software architecture that supports hardware independence, so software can be reused in Electronic Control Unit (ECU) from different suppliers[2], as long as adaptation middleware conforming to AUTOSAR is provided. This means that software can be used in the same vehicle model, even the ECU supplier changes, and also that the same software module can be reused in other models (instead of being rewritten from scratch) [17], so new products may reach the market faster.

Integration refers to the phase in which all system components (hardware, software and communications network) are assembled and tested, and is probably phase that requires more time and effort [18]. During this phase the system should be exhaustively tested. Detected errors have then to be eliminated, and calibrations of subsystems may be necessary to correct deviations from the specified system behaviour.

Testing a distributed embedded system, in which several subsystems are connected by communication networks/buses can be a difficult task: unintentional feature interaction can occur, making the system respond in a unexpected manner under certain circumstances [17]. Another issue that requires deep analysis is the behaviour of the communications platform, specially if it can handle the amount of communications in a multiplexed medium accessed by multiple applications executing in parallel, each with specific time constraints, and every network node does not exhibit jitter.

Once the product prototype is considered ready for market, it starts the production. During the product life-cycle, some factors may require software updates: error patches, add or improve functionality, or maybe adapt the source code to a new generation of hardware platform.

2.2 Embedded system architectures

In order to exemplify the diversity of systems, this section presents four practical cases of real-time systems. The real-time systems presented range from a small-sized device with large-production and short lifecycle such as a smartphone, to a large production but safety-critical system such as automotive, a mission-critical and high-priced airliner and, finally, a industrial control system.

2.2.1 Cell phone

Since the first GSM network was launched in 1991, that mobile phone devices have been disseminated to millions of costumers worldwide. Technological-push due to advances in more spectral-efficient modulations and semiconductors in combination to market-pull for new services are responsible for the incredible evolution of mobile phone devices: from the initial devices capable of voice-calls and text messaging (Short Message Service), to the current multimedia devices capable of playing music, carrying out video-calls, Internet access, digital TV reception and GPS navigation. In 2002 appeared the first GSM mobile phones with a imaging camera and other productivity applications such as an organizer, e-mail client and desktop computer synchronization, and since then two profiles of smartphones have emerged: business-oriented phones and multimedia phones. Both profiles present similar hardware characteristics, but part of the included applications differ in functionality.

The Nokia N82, depicted in Figure 2.2, is a smartphone announced as a ”multimedia portable computer” [19] and released in late 2007. This cell phone is driven by the Texas Instruments OMAP 2420 processor, based on the 32-bit ARM11 architecture, running at the speed of 330 MHz [20]. This processor, represented in Figure 2.3 holds two cores:

the ARM1136 CPU running at 330 MHz, for general processing, and

[2]Obviously as long as the application constraints are met.

Figure 2.2: Nokia N82. Photo credit: Nokia.

the TI TMS320C55x^{TM}Digital Signal Processor (DSP) running at 220 MHz, for heavy signal processing tasks required by 3G applications like videocalls or Digital Video Broadcast - Handheld (DVB-H).

The OMAP 2420 processor also includes a 2D/3D graphics accelerator for image rendering and a Imaging and Video Accelerator (IVA) for video encoding and decoding at rates up to 30 frames per second.

The Nokia N82 has a 128 MB of SDRAM, which provides approximately 90 MB of free executable RAM, after boot up.

The operating system that manages the N82 resources is the Symbian OS v9.2, based on the S60 3rd Edition platform, Feature Pack 1. Symbian is an open operating system designed for mobile phones, and has a real-time microkernel architecture that executes exclusively in ARM processors. It contains features such as preemptive multitasking, memory protection, and perfectly copes with multicore processors.

2.2.2 Automotive

Effective electronic systems started being introduced in automotive vehicles in the 1970's decade. The first system developed was the Electronic Fuel Injection, but various others – like the anti-lock braking system and the airbag system – soon began to be adopted by the automotive manufacturers. Recently, car vehicles have been evolving from an assemble of highly modularised embedded subsystems with small or none intercommunication between them, to a highly integrated system with large amounts of data being exchanged between the diverse ECUs, in order to provide new security and comfort features.

One practical example is the Mercedes-Benz S Klasse model released in late 1999 – the W220 – which contained more than 50 ECUs interconnected by 3 different bus systems, exchanging 150 types of messages and 600 signals [21]. One of the most important ECUs in a car is the Engine Control Unit, and on S Klasse vehicles motorised with the 3.2 liter V6 diesel engine (the S 320CDI V6), the Engine Control Unit is the Bosch EDC16CP31. This ECU is powered by the high-performance Freescale MPC563, a 32-bit PowerPC microcontroller that can operate at a clock speed range from 40 to 66 MHz, equipped with 64-bit floating-point unit [22] which is fundamental to cope with the complex computations performed on the multiple engine floating point parameter values. As represented in Figure 2.4, MPC563 holds 512KB of flash memory and 32KB of RAM; despite the in-chip flash memory, the ECD16CP31 ECU includes a M58BW016DB chip which provides 16 Mbit (512kbit x32) [23] of external flash, over a 32-bit data bus. The MPC563 also contains 3 CAN 2.0B bus modules

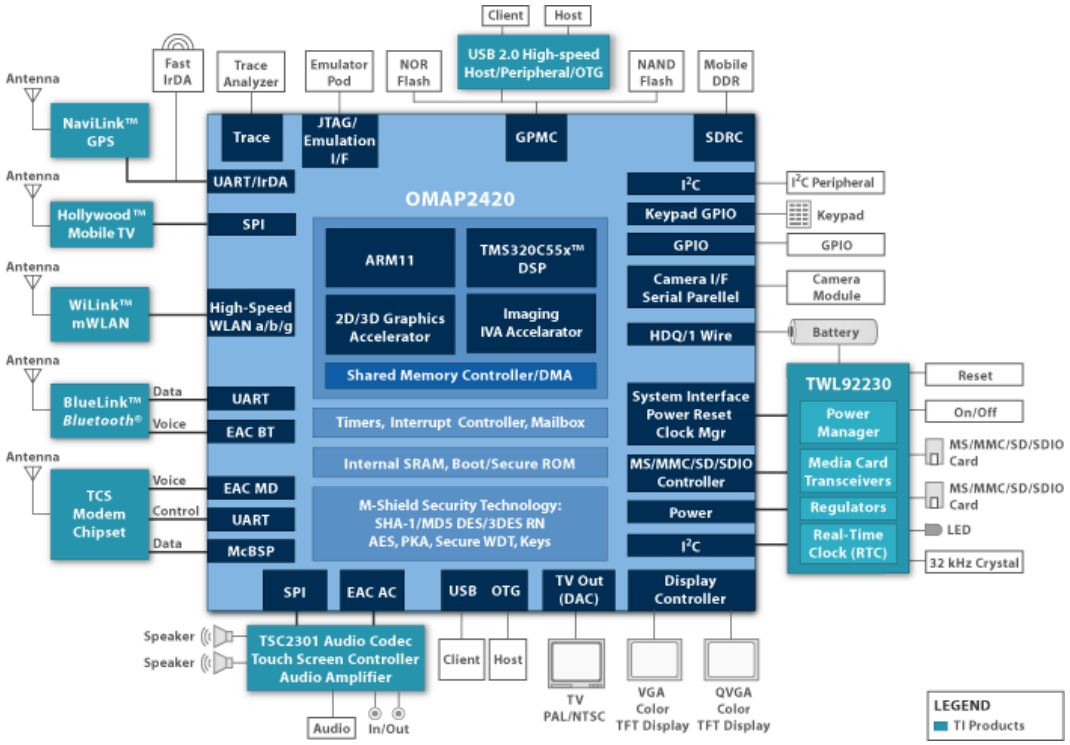

Figure 2.3: OMAP 2420 block diagram. Graphic credit: Texas Instruments.

for communication with other devices, and two queued 10-bit analog-to-digital converter modules (QADC64E_A, QADC64_B) providing a total of 32 analog channels.

2.2.3 Airliner

Aircraft flight control systems include the flight control surfaces, the pilot controls at the cockpit and connecting linkages. Classical flight control systems can be mechanical in which the pilot controls are linked to the control surfaces by mechanical parts or hydromechanical in which hydraulic circuits transmit the pilot commands to the control surfaces. These classical systems pose several restrictions of design because require careful planning for parts placement and routing of links, and are heavy. The transmission of commands by electrical control circuits and computers, known as Fly By Wire (FBW), allows more versatility in design and saves weight. In 1987, the Airbus 320 was the first commercial airliner to fully employ this concept.

The Boeing 777, illustrated in Figure 2.5, entered into service in June 1995 and is currently the largest twinjet in service. The 777 is the first commercial model from Boeing to employ a FBW primary flight control system. In a FBW aircraft there is no mechanical link between the pilot commands and the aerodynamic surfaces: instead, the pilot commands are digitised by multiple position transducers, and a set of computers will issue the appropriate commands to actuators that control the aerodynamic surfaces.

The Boeing 777 flight control system is constituted by two types of components: the Primary Flight Computer (PFC) and the Actuator Control Electronics (ACE) units. The ACE units contain transducers to control the analog command signals and inertial and air data, or analog interfaces to electrically control the electrohydraulic actuators for surface positioning [24]. The PFC is the central

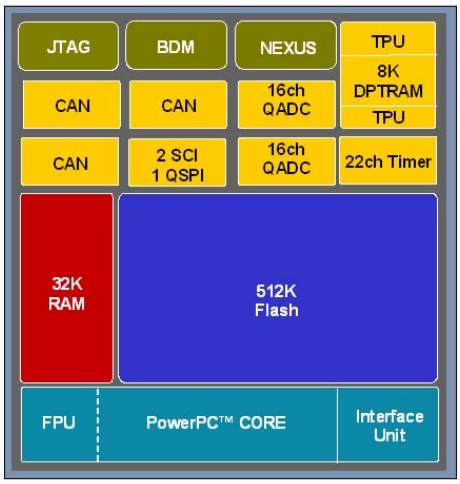

Figure 2.4: MPC563 block diagram. Graphic credit: Freescale Semiconductor.

Figure 2.5: Boeing 777. Photo credit: Boeing.

computation component of the FBW system. All digital flight control components are interconnected by an ARINC 629 data bus, a time division multiplex medium that supports up to 120 simultaneous users.

Triple redundancy for all hardware resources – computing system, electrical power, hydraulic power system and communications system – is used for fault-tolerance in the mission-critical FBW concept, so the PFC, the ARINC 629 data bus and all ACEs are triplicated into Left, Centre and Right flight control systems.

The primary flight control system counts with three PFCs, that provide triple redundant computational channels. All computers are working simultaneously so if one fails, the flight control system can continue its operation without any sort of transient behaviour. Each PFC contains three dissimilar computational lanes, each lane with its power supply and microprocessor and interfaces to the three data buses. Each lane uses a different processor architecture – Intel 80486, Motorola 68040 or AMD 29050 – which leads to dissimilar interface hardware circuits and dissimilar Ada compilers[3]. Each PFC has one computational lane in command mode, and the remaining two lanes monitoring the command lane activity. Any of the computational lanes in a PFC can assume the command mode.

Each PFC and ACE reads data from the three data buses but is only allowed to write on its associated bus, preventing an ARINC 629 transmitter failure or a bus power failure to disrupt more than one data bus. Having access to all buses allow each PFC to monitor the other two PFCs activity and, when detected, decide to through out of service a malfunctioning PFC. On the ACEs side, multiple buses access allows to continue to receive commands, even if its directly corresponding bus is shut down of service.

2.2.4 Industry automation

It has been long since automation mechanisms are employed in the industrial field, but electronics brought a new dimension to versatility and productivity in most industrial applications, from textile manufacturing to metal mining. Diverse scenarios has been leading to diverse approaches and technologies to implement industrial automation solutions.

One technology that is vastly employed in the automation of industrial processes is the *Programmable Logic Controller*. A Programmable Logic Controller is a rugged real-time digital computer designed to control machinery under severe conditions, such as found on factory floors or open-air industrial environments. Every Programmable Logic Controller has a set of multiple inputs and another set of multiple outputs which interface with sensors and actuators; usually, these two sets can be modularly customisable by means of interface modules/cards. The processing routine is an infinite loop in which the programme tasks are continuously performed, evaluating the state of inputs and calculating the next state of outputs. Automation solutions based on Programmable Logic Controllers typically aim at applications in which is expected some degree of modifications of the production process along the time (a different programme can be loaded to the device and the sensors/actuators setup can be altered). The modularity of these generic devices allows highly customised automation solutions with products already available in the market, at smaller costs and reduced implementation time, as compared to a specifically designed control system.

The cost of developing a new specific controller system can only be covered by a large-volume application, and it can only be economically effective if the controlled system is not expected to suffer modifications during its operational life-time[4]. In such cases, the controller system may be designed

[3]The source code written in Ada by GEC-Marconi (today BAE Systems Electronics Limited) is common in all computational lanes.

[4]As an example of such type of systems, one model of a CNC machine tool is supposed to be produced in hundreds or

and packed in a single chip – the *System-on-a-Chip (SoC)* – in which the processor core(s), memory, external device interfaces and other hardware components are all integrated in a single integrated circuit, reducing the complexity and dimensions of the printed circuit board and the number of discrete board components. The SoC can be implemented on a Field-Programmable Gate Array (FPGA) – an integrated circuit which is configured after manufacturing, normally by a hardware description language – or, in large production volumes, by an Application-Specific Integrated Circuit (ASIC) – an integrated circuit which is already configured from factory. Software for controller systems based on SoC can range from an application specific programme to a multitasking preemptive OS running multiple concurrent applications, depending on the dimension and complexity of the controlled system. Critical applications may require an operating system with a deterministic real-time scheduler, such as Wind River VxWorks or RTLinux, which assures that concurrent hard real-time tasks will meet the deadlines while executing soft or non real-time tasks in lower priorities.

While generic or specific controllers have an active role in the operation of the controlled equipment, *industrial computers* are used instead to supervise and control the overall system activity. In this sense, industrial computers are not concerned in assisting and controlling the tasks of an individual machine, but rather to monitor the status of specific variables and to be aware of certain events, having a broader-scale vision of the production system. The timing requirements for an industrial computer are not so stringent as the timing requirements that dedicated controllers have to cope, so industrial computers can be managed by non-deterministic OS. Industrial computers have rugged designs and quality computer components to withstand harsh industrial environments: dust-filters, cooling systems and redundant power supplies and data-storage. Some models of industrial computers are equipped with more expansion slots than typically found in desktop computers, to host interface boards that connect the computer to the controlled system components.

2.3 The REMPLI system

The REMPLI project aims to design and implement a communication infrastructure for real-time distributed data acquisition and control operations, exploiting the power grid as the communication medium. The immediate target application is remote meter reading with high time-resolution, being the meters energy, gas, water or heat meters.

The utility companies – the users of the REMPLI system – will have access to detailed information about how the energy is consumed by end-users, so energy flow may be better controlled and leakage can be detected more efficiently. The REMPLI system will be able to determine the actual status of the power grid and can be equipped to remotely terminate supply of energy, if required [25]. Energy billing and energy management are examples of high-level services that can be developed on top of the REMPLI system.

The PLC infrastructure provided by the REMPLI system will allow to access remote metering and control equipment, remaining open to various kinds of add-on services. The schematic architecture of the communication network is illustrated in Figure 2.6.

The REMPLI communications infrastructure can be divided in four segment types:

- low-voltage segments, covering groups of energy consumers;
- medium-voltage segments, between the primary and secondary transformer stations;

thousands of units. Although the set of tasks of the CNC machine is programmable and can be changed several times a year, and the tools used are interchangeable, the controlled machine components – motors, tool turrets – are kept unchanged.

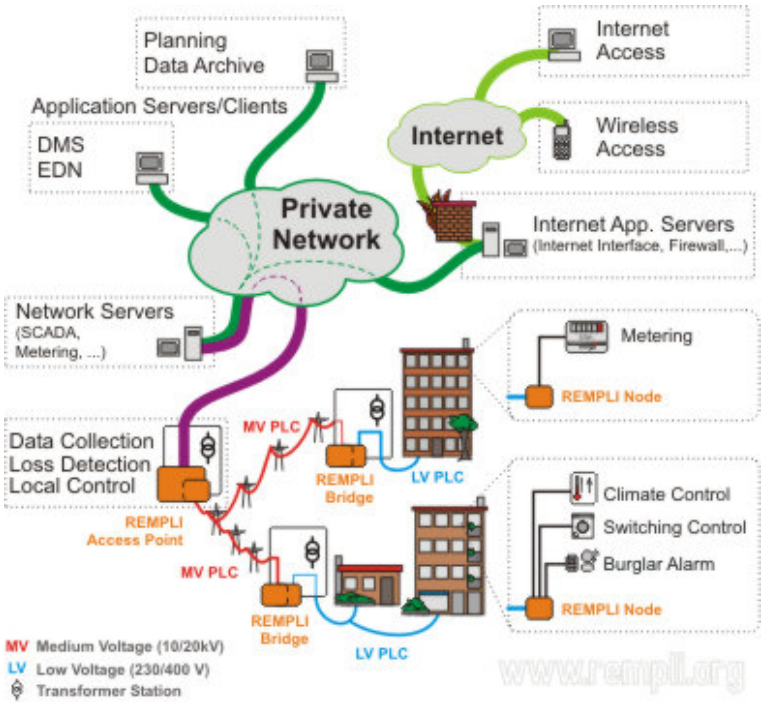

Figure 2.6: REMPLI system architecture.

- TCP/IP- or IEC 60870-based [26] segments between the primary transformer stations and Application server(s);
- TCP/IP communications between Application Servers and correspondent clients.

At the bottom-level of the infrastructure, the REMPLI Nodes are installed at the consumer site. A REMPLI Node includes a set of standard protocol inputs for communications with metering and control devices. Besides the inputs, REMPLI nodes are also provided with outputs capable of switching off and on the supply of electricity/heat/gas/water to a particular consumer, upon commands from the utility company. The Nodes are coupled with a PLC interface in order to connect with the power grid, but they can also be provided with alternative means of communications, like GPRS link.

At the top-level of the infrastructure, the Application Servers of utility companies are connected at the REMPLI Private Network [27]. Each Application Server is responsible for a dedicated function, such as metering, billing or SCADA; some servers may provide processed data to clients located outside the REMPLI Private Network (e.g. Internet, UMTS). An Application Server can access any REMPLI Node using the services of a REMPLI Access Point, a device that performs the gateway between the TCP/IP and medium-voltage PLC segments.

The software architecture of the REMPLI Node allows to execute in parallel different applications, each provided with an interface to the PLC environment. Every application running at the Node is reachable on the other side of the REMPLI system. This way, Application Server(s) can request data collected by a Node application or send commands to the application for self-configuration or to actuate an attached peripheral device [28, 1, 29].

The power-line network consists of one or multiple Low-Voltage segments where the Nodes are located, and one Mid-Voltage segment where the Access Points are installed. Communication at both levels is based in the Master/Slave paradigm. The Medium-Voltage segment is coupled to the

Low-Voltage segments by REMPLI Bridges, at secondary transformer stations. The REMPLI Bridge is a device which interconnects a Medium-Voltage PLC slave and a Low-Voltage PLC master. The REMPLI Bridge is completely transparent for Access Point-Node communications, simply forwarding Access Point requests to Nodes and Node responses back to Access Points. In this sense, an Access Point sees all Bridges and Nodes as if directly connected to it, and the communications network becomes a single master/slave communication environment.

2.4 REMPLI communications architecture

In actual industrial applications, metering and control devices communicate with application servers using a particular protocol, such as EN-62056-21 or M-Bus (metering applications) or IEC 60870-5-101 (SCADA applications). Application Servers expect to find their respective remote (metering or control) devices directly connected to them, using the referred protocols for communication.

The REMPLI communication system has the objective of tunnelling transparently particular metering and SCADA protocols over the power-line network [2, 30], in such a way that application servers and remote devices do not need to suffer any modification to maintain their regular operations, despite the selected physical network in use. Although the REMPLI architecture allows data transmission over different communications media such as POTS and GSM, the main focus of REMPLI is the PLC.

The communications architecture between a REMPLI Access Point and a single REMPLI Node can be illustrated as in Figure 2.7. This architecture can be easily divided into three logical subsystems: the *Access Point Application*, the *Node Application* and the *Power-line Communications System*.

The REMPLI Node is able to physically connect to metering and control devices, by means of specific hardware interfaces and logical drivers. Each driver is dedicated to a single combination of hardware interface and communication protocol, and is able to handle multiple devices of the same equipment[5]. A driver at the REMPLI Node can perform metering data processing and compressing, in order to reduce the usage of network bandwidth. This processing must be supported by the correspondent REMPLI Access Point driver, so the original data can be reconstructed.

The REMPLI software architecture allows the development and integration of drivers to support additional protocols, as long as they are supported by respective hardware interfaces.

Certain protocols data, like the pulse-based S0 (IEC 62053-31/DIN 43864) cannot be directly transported over the PLC. The architecture of REMPLI nodes allows the development of protocol translators so data originated from one device may be translated to a fully-supported protocol, such as EN-62056-21, understood by metering software on the Access Point side.

The REMPLI drivers at the Access Point side perform the conversion of metering/SCADA protocols received from the Nodes into their TCP/IP-based equivalents. This task can be achieved by simple tunnelling of by standard conversion[6]. As mentioned above, a Node driver may introduce some deviations from the original standard in order to optimise bandwidth usage. These deviations must be undone by the respective Access Point driver, so the tunnelled protocol fully complies to the standard, as seen by the metering/SCADA devices and Application Servers.

The Power-Line Communication System is formed by a set of software and hardware components which are responsible for data transmission over a single-segment (Medium Voltage) or dual-segment

[5] For instance, a single M-Bus driver in one REMPLI Node can handle several M-Bus meters, located in one building.

[6] As an example of standard conversion, SCADA protocol IEC 60870-5-101 on PLC medium has to be converted to IEC 60870-5-104 on IP network.

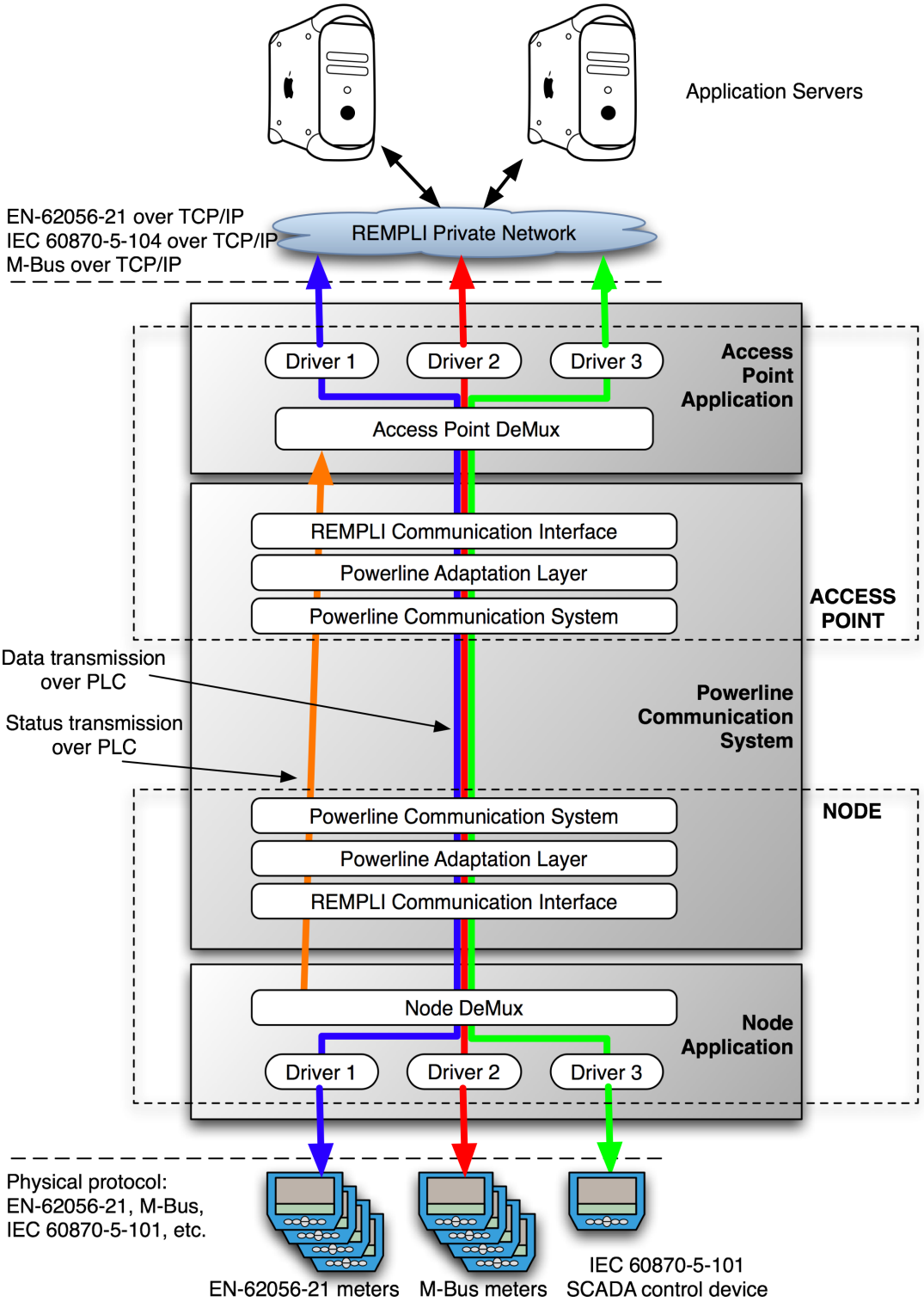

Figure 2.7: REMPLI communications architecture over PLC.

(Medium and Low Voltage) power-line networks. The Power-Line Communication System employs a three-layer architecture – equivalent to OSI model transport, network and physical layers – to handle packet-oriented driver-to-driver communication. Metering and SCADA data is requested to Node drivers by Access Points drivers, according to the master/slave model. Protocol requests and replies are transported in Protocol Data Units (PDUs) which are encapsulated into REMPLI network packets from driver to driver.

The PLC System offers fast status transmission, allowing a Node to send fast signalling bits to an Application Server, for internal driver-to-driver communication.

2.5 REMPLI software architecture

2.5.1 Software architecture design

As illustrated in Figure 2.7, both Node and Access Point interface with the *Communication System* (in the case depicted, power-line) through two component blocks: the *REMPLI Communication Interface* (RCI) and the *Power-line Adaptation Layer*.

For each medium, a specific set of hardware and software – Communication System – provides functions and services corresponding to Physical and Data Link layers in Open Systems Interconnection (OSI) Basic Reference Model [31].

Adaptation Layers are used to establish a common interface to different types of physical media that may support REMPLI communications, such as power-line, ISDN, POTS or GSM. Adaptation layers implement the functionality [32] corresponding to Network and Transport layers in Open Systems Interconnection (OSI) model, over the respective medium:

Delivery of arbitrarily-sized PDU. Delivery of PDU from the Access Point to a single or multiple Node, and the other way around is the most visible functionality. Communications may be reliable (if delivery fails, the communications system will perform several retransmission attempts before quitting) and unreliable (a transmission failure is ignored). Access Points may unicast, multicast or broadcast data, while Nodes can only broadcast.

Concurrent transmission and priority management. As the Access Point architecture allows several Drivers executing simultaneously, it is possible that multiple PDU may be concurrently submitted for transmission to target Nodes. Each PDU has a priority class associated, which allows scheduling transmissions, according to priority. A higher-priority PDU is always transmitted before any lower-priority PDU, even if it is necessary to suspend a transmission already in progress.

Fast status information transmission. Every Node can set a size-limited bitfield in which all modifications can be quickly propagated to the Access Point, as out-of-band data. Nodes are only allowed to modify their status bitfield, and Access Points can only receive status information.

Network discovery. The REMPLI network is able to report in real-time which Nodes are currently logged in, so it is possible for the Access Point Application to detect potential problems with Nodes and calculate alternative paths to route packets through other Access Points.

Point-to-point connection management. Some communications systems, such as POTS and ISDN require point-to-point connections to Nodes before communications become possible.

The REMPLI Communication Interface is a specification that establishes a common access to the functionality of any Adaptation Layer independently from the communications medium, and is available at Access Points and Nodes, so that applications at both sides may exchange data.

At the opposite side of the communication stack, *Access Point Drivers* have to engage communication with their respective counterparts at the Nodes, in order to satisfy requests issued by possible multiple Application Servers which are connected to the REMPLI Private Network.

Node Drivers have mainly to attend Access Point Driver requests, exchanging commands and data with connected meters and SCADA equipment [33].

As the Power-line Communication System provides packet-oriented communication, each PDU is independently addressed and delivered to a specific Node. Apart from network addressing, it is also necessary to address individual drivers at the Node and at the Access Point. This additional addressing is performed in the *De/Multiplexer*, which multiplexes and de-multiplexes communication streams between different pairs of drivers. Additionally, if multiple Application Servers are simultaneously connected to the same driver, De/Multiplexer helps driver to distinguish which incoming response PDU should be routed to which application (request/response identification). The De/Multiplexer component at Access Points is also responsible to handle communication path switching and redundancy: if an Access Point can not reach directly one node, the De/Multiplexer can try to re-route communications through another Access Point which has available a direct communication path to the Node.

2.5.2 Software architecture implementation

The actual implementation of the software concept described in 2.5.1 is within three main software components: *Network Layer*, *Transport Layer* and *Driver De/Multiplexer* [6]. Figure 2.8 illustrates the stack of communication software components found in every REMPLI device – Access Points, Nodes and Bridges. Although these three types of devices share the same communication stack design, making the communication stack at both sides conceptually symmetric, each software component is optimised for the respective device type[7].

The access to the physical medium is carried by specific hardware. In the case of PLC, it is employed low- and medium-voltage PLC modems, depending on the segment to which the device is connected. The primary task of these modems – corresponding to the Physical layer in the OSI model – is to perform the operations required for data transmission: data coding and modulation. The channel access technique selected is Time Division Multiple Access (TDMA), and Orthogonal Frequency Division Multiplexing (OFDM) as the transmission method [34]. Access to PLC modem functionality and configuration is available to upper layers through a specific device driver.

The Network Layer is responsible for managing the Master/Slave type of medium access in the network [35]. Its main role is to take care of device-to-device communication, mapping to the Network OSI Layer. Still, the REMPLI Network Layer must also manage some LLC/MAC functionality (extending over other lower OSI layers). The Network Layer implements three priority queues to support diverse traffic urgency. Data packets have generally small sizes, to minimise retransmission of long blocks in the case of data corruption by noise degradation, being the maximum fragment size (depending on the configuration) ranging from 40 bytes to 128 bytes.

The REMPLI Transport Layer manages the end-to-end Access Point/Node communications, eventually using Bridges. The REMPLI Communication Interface (RCI) is implemented as part of the Transport Layer, working as the access interface to upper layers. One of the main functions of the

[7] E. g. there are three different versions of Transport Layer, one for each device type.

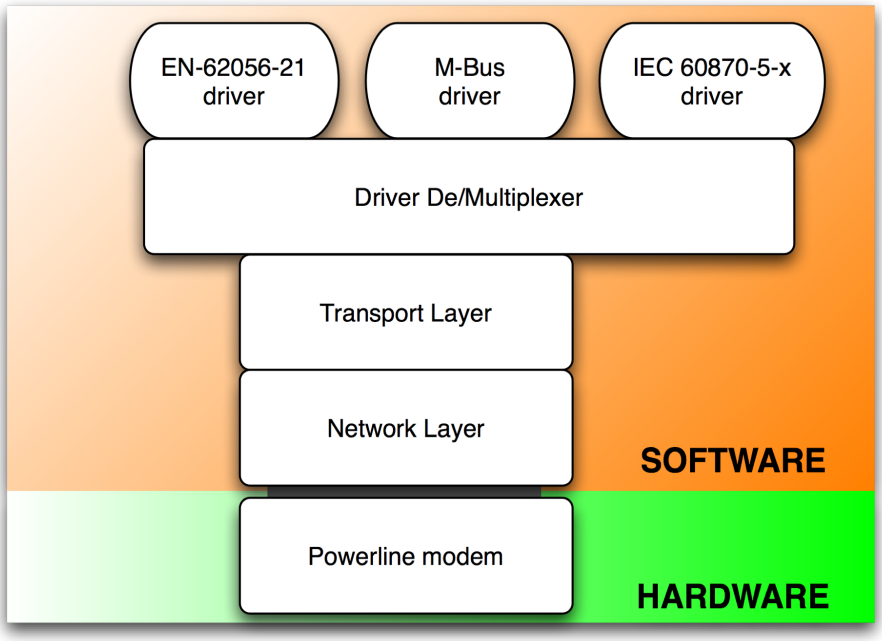

Figure 2.8: Communication software building blocks.

REMPLI Transport Layer is fragmentation (including correct order delivery) mapping directly to the respective OSI Layer. Due to the particular topology of the REMPLI system, communication between Access Points and Nodes via Bridges is managed at the Transport layer level, so the REMPLI Transport Layer must also handle some routing functionality (normally associated to the OSI Network Layer). The Transport Layer includes the following functionality [32]:

- Fragmentation and Bundling.
- REMPLI Node Address / REMPLI PLC address conversion
- Routing (via REMPLI Bridges).
- Support of different Network Layer implementations.
- Retransmission (TCP-like service).

The Driver De/Multiplexer is more connection-oriented and handles not only the redundancy between several Access Points but also the demultiplexing of the Drivers in each end and so it is mapped to the OSI Session layer.

The Drivers are mapped to the OSI Presentation layer since they have several conversion and control functions that optimize the usage of the RCI functionality for a particular protocol [36].

Chapter 3

Embedded platform

3.1 Hardware platform

REMPLI devices[1] are embedded systems that should operate remotely, requiring minimum to none human intervention. iAd GmbH[2], one of REMPLI industrial partners, developed a specific embedded hardware platform, which served for software verification and also as a prototype basis for future products. At the core of the REMPLI embedded processor board is the hyNet 32XS, a Network Processor from Hyperstone AG[3]. The hyNet 32XS is a 32-bit RISC/DSP microprocessor that integrates components for Ethernet, USB 1.1, CAN-Bus, ATM and UART interfacing [37], but its high versatility of interfacing also allows this processor to communicate easily with M-BUS metering devices. The processor board prototype supports the following interfaces:

- two Ethernet channels (electrical or optical);
- two serial ports, in RS232 or RS485 mode;
- one CAN interface, fully compliant with the ISO 11898 standard [38];
- one ISDN interface, fully compliant with ITU-T I.430 standard [39];
- one USB port, compliant with USB specification 1.1.

The processor board includes 32 MB flash memory module with 16-bit data width, and 64 MB of DRAM with 32-bit data width. The flash memory is mountable at the file system, and can hold a local installation of the OS and applications. The embedded system can also boot up from a remote network OS image by Trivial File Transfer Protocol (TFTP) [40], and Network File System (NFS) protocol [41, 42, 43] enables the use of alternative remote file systems. These two features are quite useful during software development and testing phases, as it allows faster verification cycle times and file logs extraction.

Communications over PLC are achieved by means of a PLC modem board coupled to the processor board. REMPLI Bridges have two modem boards, as they have to connect simultaneously to two network segments, while Access Points and Nodes have only a single modem board connected. Figure 3.1 presents a prototype of the REMPLI Bridge, where is possible to discern visually the processor board on top of two PLC modem boards.

[1] Access Points, Bridges and Nodes.
[2] http://www.iad-de.com
[3] http://www.hyperstone.com

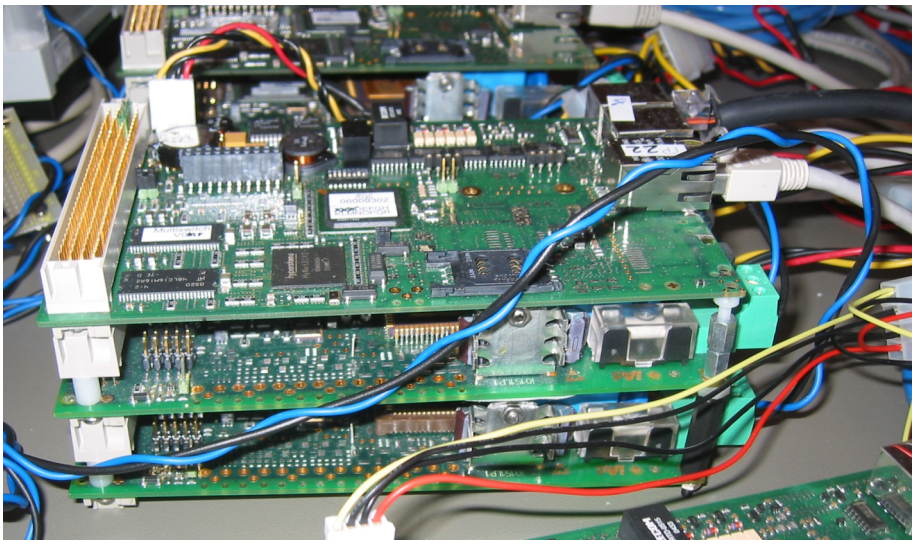

Figure 3.1: REMPLI Bridge prototype.

The modem board holds the DLC-2C/CA chipset, a high-performance narrow band communication controller MCM (Multi Carrier Modulation) and power line chipset provided by iAd GmbH. This chipset consists of one analog and one digital chip, and includes an interface to the hyNet 32XS processor, as illustrated in Figure 3.2.

The DLC-2C digital chip consists of a 16-bit digital signal processor, digital filters, synchronization unit, automatic gain control, de/encoder and also an interface to the hyNet 32XS. The DLC-2CA analog chip performs the adaptation to the communications medium, carrying out the Digital-to-Analog and Analog-to-Digital conversions. This chipset allows communications via high, medium and low-voltage lines, under heavy-disturbance conditions, in the frequency range from 9 to 490 kHz, with data rates from 9.6 to 576 kbps. OFDM is the modulation scheme for data transmission and Forward Error Correction (FEC) is used to reduce the effects of noise affected media. This chipset is compatible to standards EN 50065 (CENELEC) [44] and IEC 61000-3 [45].

3.2 Software platform

3.2.1 Operating system

The complexities associated with the requirements of each type of REMPLI device discards the possibility of one single monolithic application to carry all the operations of the device, thus it is necessary the employment of an operating system to manage each device. The described architecture runs an operating system from the UNIX family, which brings valuable advantages, as the UNIX API is well-known and extensively documented, an important factor that eases the development and maintenance of stable software.

The system kernel is a uClinux port to the hyNet 32XS, a derivative of Linux kernel intended for microcontrollers without Memory Management Unit [46], already ported to several microprocessor architectures. This kernel port offers the standard Linux multitasking environment, so multiple applications can execute concurrently. Additionally, this kernel allows multithreading, which is of enormous use for the development of complex applications that have to deal with multiple asynchronous

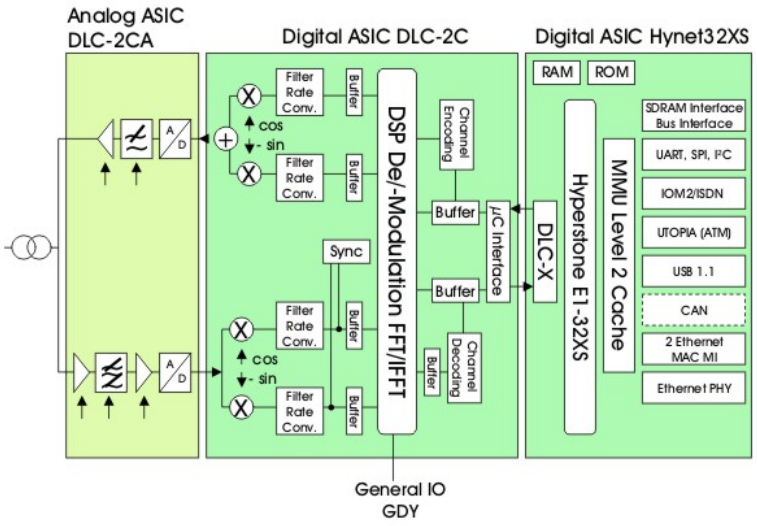

Figure 3.2: DLC-2C/CA and hyNet 32XS integration (courtesy iAd GmbH).

inputs. The development of device drivers follows the standard Linux device driver documentation [47].

3.2.2 Programming tools

Access to operating system services is provided by uClibc, a small C standard library intended for embedded systems [48] which was also ported to the hyNet 32XS architecture. uClibc has a different set of goals than a native library – like glibc – has. As an example, glibc attempts to be compliant with most standards, to be available on most of the operating systems and architectures, and strives for Application Binary Interface (ABI) compatibility, which results in a very large sized library. Taking a different perspective, uClibc attempts to provide the maximum possible functionality in the smallest amount of space. uClibc was focused since the design phase to be a C library for embedded Linux, so many optimization was possible as many of the glibc features were not necessary. Furthermore, uClibc permits the selection of supported features, so it is possible to generate a tailor-made configuration for a specific application, at the cost of different ABI for different configurations.

A toolchain including the GNU C-Compiler (gcc), the GNU debugger (gdb) and the GNU binutils has been ported by Hyperstone AG for the hyNet 32XS, to build the Linux kernel and applications [49]. The toolchain is available for desktop Linux, so the Linux kernel and applications are built by cross-compilation. Building the kernel or an application requires the traditional sequence of operations/tools depicted in Figure 3.3, where uClibc is linked only to applications.

3.2.3 Kernel space / User Space

Implementing the totality of the complex REMPLI communication protocol stack in the kernel of the operating system would be too cumbersome, disabling the full use of capabilities which are only available in user space processes. However, building protocols as user space processes must not impair the responsiveness of the communication. Therefore, the REMPLI protocol stack is distributed over the kernel space and the user space [50]:

- for responsiveness issues, the Network Layer (NL) is implemented in the kernel,

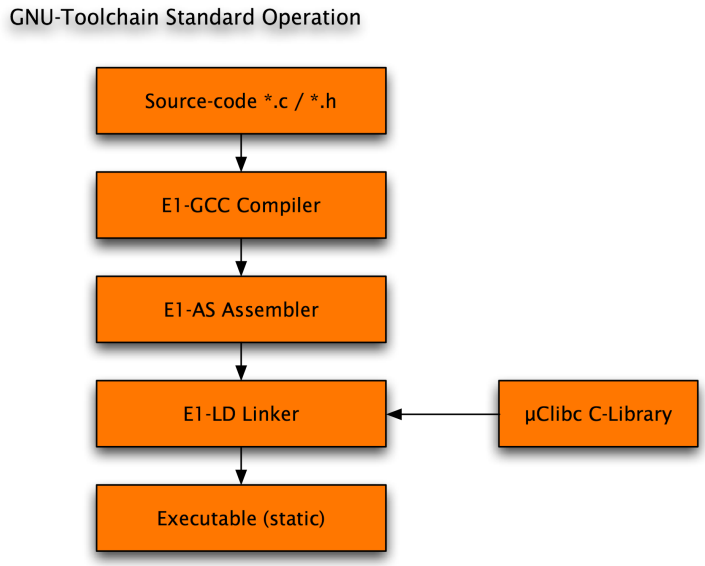

Figure 3.3: Application build sequence of operations (courtesy iAd GmbH).

- to cope with the functional requirements associated complexities, the Transport Layer and the Application Drivers Demultiplexer/Multiplexer (DeMux) are implemented as user space processes.

The Network Layer is implemented as two Linux device drivers inside the kernel: one for Master Network Layer and another for Slave Network Layer. These device drivers provide services to exchange data over the PLC network and device control as well. The Access Point kernel includes only the Master NL device driver, and the Node kernel includes only the Slave NL device driver. Since the bridge holds one Master and one Slave communication devices, its kernel includes both Master NL and Slave NL device drivers.

The other REMPLI communication stack layers are implemented as user space processes, taking profit from the services offered to applications by the kernel, such as multitasking, multithreading and Interprocess Communication mechanisms.

Chapter 4

REMPLI Transport Layer

The REMPLI Transport Layer is built by four main modules: *RCI Manager (RCIM)*, *Network Layer Interface (NLI)*, *Queue Manager (QM)* and *Transport Route Manager (TRM)*. These four modules are interconnected, as presented in Figure 4.1.

The RCI Manager module distributes messages between the Queue Manager / Transport Route Manager and the Applications themselves using the REMPLI Communication Interface.

In the opposite side of the Transport Layer, the Network Layer Interface module distributes messages between the Queue Manager/Transport Route Manager and the Network Layer, performing parameter conversion when needed.

The Queue Manager module manages all the packet data information. Tasks like queue generation, disposal, fragmentation and transport system header processing are done at this module. This module also multiplexes requests from the Applications and Transport Route Manager to the Network Layer Interface data transmission services.

The Transport Route Manager module handles not only the scheduling tasks but also all the tasks that need internal Transport Layer communication. When a new data request is queued in the Queue Manager it is the task of the Transport Route Manager to provide the appropriate route (or to reject the request due to unavailable path). The Transport Route Manager will also instruct the Queue Manager when each queue should be served.

4.1 The RCI Manager

Messages which their origin or destination is the DeMux, are handled by the RCI Manager.

The interface between Driver DeMux and the Transport Layer is the REMPLI Communication Interface (RCI). According to the RCI functional description [32, 51], the Driver DeMux and the TL exchange three types of messages – *Command*, *Response* and *Notification* – over two types of channels – *Command channel* and *Notification channel*.

A message which its origin is an application is called Command. Inversely, a message which is destined to an application may be a Response to a command or a Notification.

Commands and their respective Success/Error Responses are routed over Command channels[1], while Notifications are routed to the Driver DeMux over a specific Notification channel[2]. The Transport Layer supports multiple simultaneous Command channel connections, but only allows one Notification channel at a time, as depicted in Figure 4.1.

[1] Identified by CMD in Figure 4.1

[2] Identified by NTFY in Figure 4.1

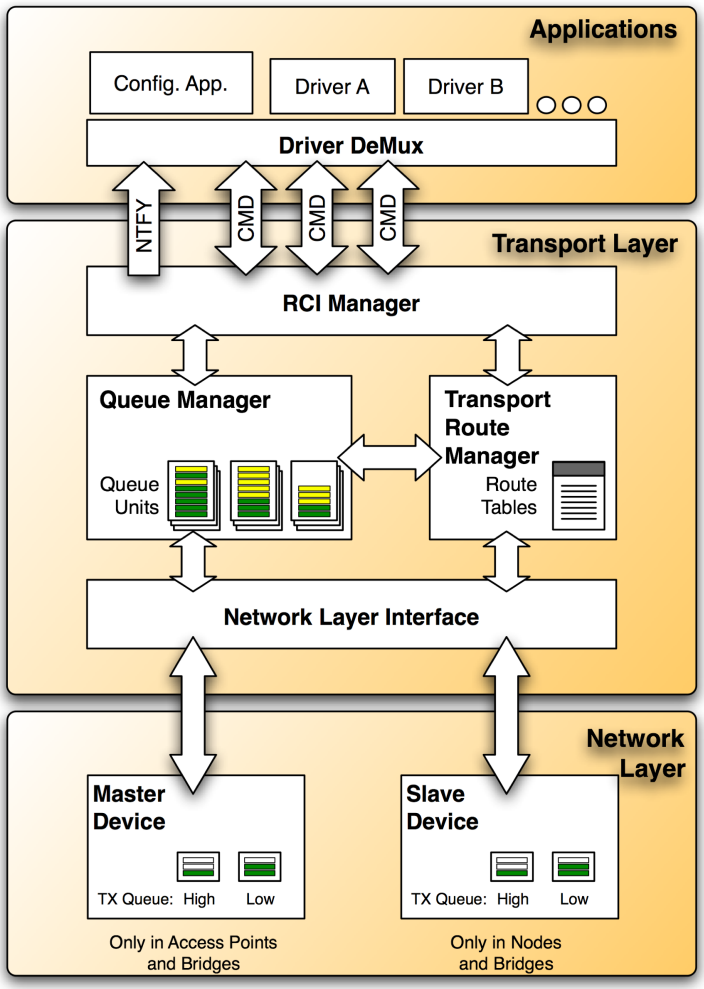

Figure 4.1: Transport Layer integration with other communications software.

4.1.1 Commands and OK/Error Responses

Applications issue Commands to request Transport Layer services. Depending on the action required, the Command message may be delivered to the Queue Manager or to the Transmit Route Manager. Possible actions may be:

- Request to get data from another REMPLI device.
- Request to send data to another REMPLI device.
- Request information about the status of the network.

For each Command received, the Transport Layer is expected to reply to the respective application with a SUCCESS/ERROR Response, over the same channel from where the Command was received. As multiple Command channels may be simultaneously open, the RCI Manager must have a mechanism to select the proper channel to transmit each Response.

A Success Response is returned when the Transport Layer was able to process correctly the issued command. Success Responses may indicate that data was fully transmitted over the network or carry data requested to the TL, such as status flags or live nodes lists. A Success Response only indicates that the Command was correctly processed by the TL, so it never carries data requested from another REMPLI device.

An Error Response is returned every time the Transport Layer was unable to process the received command. The Error Response indicates the cause of failure, which may occur when the processing of the command takes longer than the specified timeout, an error in transmission of data happens or other possible conditions.

The Driver DeMux does not need to wait for the response to a previous command before issuing another command over the same channel[3]. The Driver DeMux identifies each Command with a unique *Inter Process Communication Transaction Identifier (IPCTransID)* value. Every Response carries the same IPCTransID value as the respective Command, so the Driver DeMux is able to match both messages. Figure 4.2 presents a possible case, in which the Error Response relative to Command identified with `IPCTransID 922` is received by the Driver DeMux after two other commands have being sent over the same channel. This example also illustrates that the sequence of responses may not match the sequence of commands, as the Transport Layer processes commands in parallel and processing time may differ.

4.1.2 Notifications

The Transport Layer generates a Notification in presence of a relevant event, which is delivered to the Driver DeMux through a unique unidirectional Notification channel. Notifications on the Access Point side allow applications to know about modifications on the network status and also to receive data (responses to previous requests of data) from Nodes. Notifications on Bridges and Nodes generally carry data requests issued by Access Point applications, but may also occasionally carry software updates.

[3] In extreme, the Driver DeMux may issue a sequence of commands over one channel, before receiving the response to the first command of the sequence.

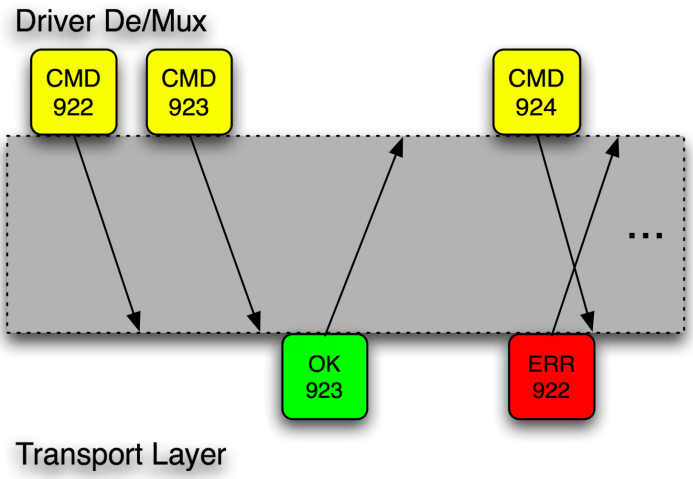

Figure 4.2: Timing perspective of Commands and Responses over one Command channel.

4.2 The Network Layer Interface

The Network Layer Interface switches messages between the Network Layer and the Queue Manager and Transmit Route Manager modules.

The Network Layer may deliver packets to the Transport Layer whenever relevant events on the network occurred or data from another REMPLI device was received. According to the type of packet received, the Network Layer Interface will route it to the Queue Manager or to the Transport Route Manager. Generally, received data and indications regarding data transmission are routed to the Queue Manager, while network status updates are delivered to the Transport Route Manager. Indications of other REMPLI devices logging in or out the network are reported to both QM and TRM, as such information is necessary for QM to manage data queues and TRM to calculate routes.

Regarding the packets addressed to the Network Layer, most of the packets from the QM carry data that is expected to be transmitted over the network. The TRM may issue requests in order to update its knowledge of the actual network status.

The Network Layer offers access to the network communications device. Due to the master/slave communications paradigm, there are two types of network communications devices: Master device and Slave device. The Master device is present in the REMPLI devices that act as network masters – Access Points and Bridges – while the Slave device is present in the REMPLI devices that act as network slaves – Nodes and Bridges. Figure 4.3 illustrates how the communications devices are employed on the PLC network. The REMPLI Bridge interconnects two network segments: on the segment where are Access Points the Bridge acts as slave, while on the other segment, the Bridge acts as the master for the Nodes.

In the case of Access Points and Nodes, the NLI tasks are quite straightforward, as all network packets are routed to/from the single network communications device present. On Bridges, the NLI must analyse the type of each packet being sent to the Network Layer and determine the communications device the packet should be delivered.

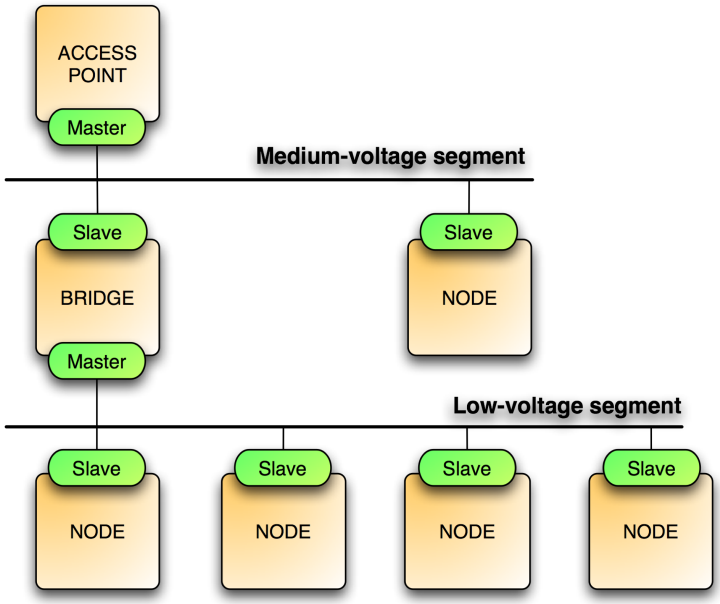

Figure 4.3: Master and Slave devices in REMPLI devices on the network.

4.3 The Queue Manager

The main task of the Queue Manager is to store temporary data before it is sent to the network or to the applications. Other tasks include:

- Pair requests and responses using the Transport System header data and stored information.
- Provide communication service to the Transport Route Manager own messages.
- Packets fragmentation/assembling.
- Bridge forwarding.

When the Queue Manager receives a new request from the applications it will store the data of the request and inform the Transport Route Manager about the new queue. The Transport Route Manager will reply first by sending route information or a ”discard queue” command. After this, the Transport Route Manager will instruct the Queue Manager to send to the network each fragment in the queue and the Queue Manager will keep the Transport Route Manager informed on the data transmission process. When all fragments are processed the queue information is deleted at both the Queue Manager and the Transport Route Manager.

On reception from the network the Queue Manager rebuilds the data information if needed and delivers the final data block with additional service information either to the application or the Transport Route Manager.

Generally, the Transport Layer must fragment the application requests in several frames and reassemble them at the destination since the Network Layer uses small fixed-length frames. The Transport Layer must support several parallel transactions, so the Queue Manager must identify each fragment with a packet id. For correct-order delivery (the Network Layer may deliver fragments out-of-order) the Queue Manager must also include fragment order information. Both these fields

have been reduced to a minimum to limit the impact of the Transport System header on the data payload. The Queue Manager at REMPLI Bridges also supports different sized frames for the two network segments: Bridges will keep temporary buffers for the en-route packets in order to optimize this fragment-size conversion. Due to the technology used, the raw data length of the Network Layer packets is always a power of 2, but since this data length includes the Network Layer own headers, the bridges may not do simple division or union of fragments.

4.4 The Transport Route Manager

The Transport Route Manager module is the main responsible for management of the system. In addition to the main run-time route determination and scheduling of fragments, the Transport Route Manager also takes care of the system start-up (including remote device plug-and-play operation) and automatic route discovery and route status dissemination. Using connection/disconnection information from the Network Layer, the Transport Route Manager keeps track not only of the current network topology (i. e. the available routes between an Access Point and a Node) but also of the addresses used in each route. To achieve this goal the Bridges forward route information between the two network segments when needed. The Transport Layer must handle (and match) 3 different addresses:

1. *REMPLI Unique Serial Number (RUSN).* This is a long number, that identifies univocally any REMPLI device. This number is set at factory, and there should be no multiple REMPLI devices identified with the same RUSN.

2. *REMPLI Node Address*. This is a long number, used to identify Node devices at Access Point Application requests. The REMPLI Node Address is normally associated with an installation site, so Access Point Application requests are addressed to a network location, instead of a specific REMPLI device, hiding hardware details to Access Point Applications. A configurable table maps Unique Serial Numbers to Node Addresses, so the requests to a particular location are aimed to the corresponding REMPLI device. When a REMPLI device needs replacement, the address translation table is reconfigured, so the REMPLI Node Address remains the same while the corresponding RUSN is updated.

3. *REMPLI Network Address*. This is a short number, used for Network Layer requests and responses. The REMPLI Network Address is assigned by the master when a slave logs into the network segment, similarly to the Dynamic Host Configuration Protocol (DHCP) available on IP networks [52]. It is the task of the TRM to convert Node Addresses into a route exploitable by the use of Network Layer services.

4.4.1 Route Selection and Cost Estimation

The two main services used by Access Point Applications will be a Request/Response service and a Request without Response service. The main task of the TL with these two services is to decide which route should be used to send the request from the Access Point (AP) to the Node using the available Bridges. In the example depicted in Figure 4.4, a request from Access Point 001 to Node 203 may be routed either using Bridge 102 or Bridge 104.

For simplicity the REMPLI system routes all fragments of a particular request (and response, if applicable) using the same path. This simplification means that routing decisions are only taken on

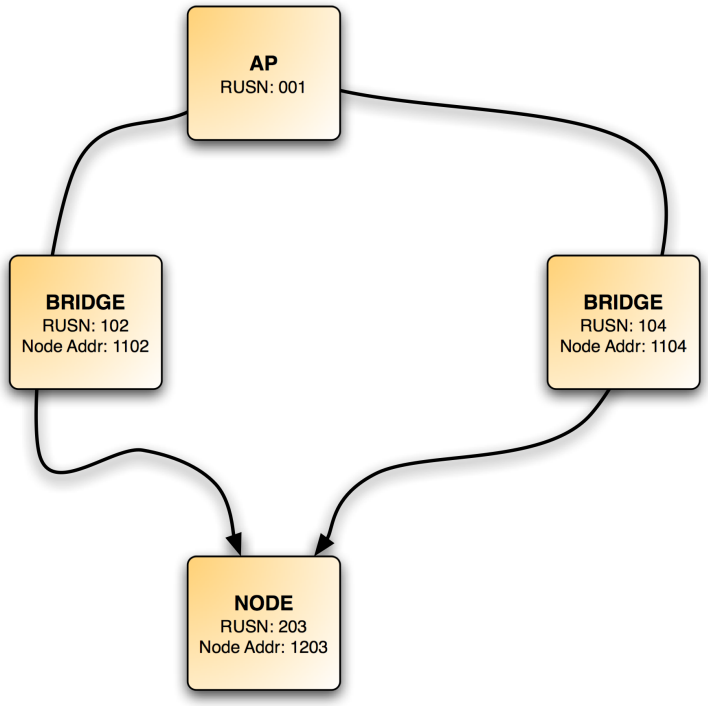

Figure 4.4: Two possible routes to reach Node 203.

the Access Point and once per request/response. It also means that path information must be sent in the packets.

The route selection process is done on a request-by-request basis. The Access Point scheduler may use the Probable Cost of each route to make the routing decision. To estimate the cost of each route (one way) the transport system uses (as in Figure 4.5):

- Access Point/Bridge Link quality.
- Bridge queue status.
- Bridge/Node Link quality.

The Access Point/Bridge Link quality is available directly at the Access Point's Transport Route Manager but it is also, by definition, an estimate: the Link Quality indicates the previous link quality. Due to the inconstant characteristics of the data communication medium, this value may change unexpectedly and the scheduling algorithm must have this effect into account.

Bridge/Node Link quality is data relayed from the Bridge to the Access Point by the Transport Route Manager. It has the same unpredictability problem of the previous plus the significant additional delay due to Bridge to Access Point communication.

Bridge queue status is also data relayed from the Bridge to the Access Point by the Transport Route Manager. Here the unpredictability is also a factor since other Access Points may influence these queues.

For a request/response service we should also include the queuing delay on the Bridge in the reverse direction and use a correction factor for the transport delays due to the extra overhead needed

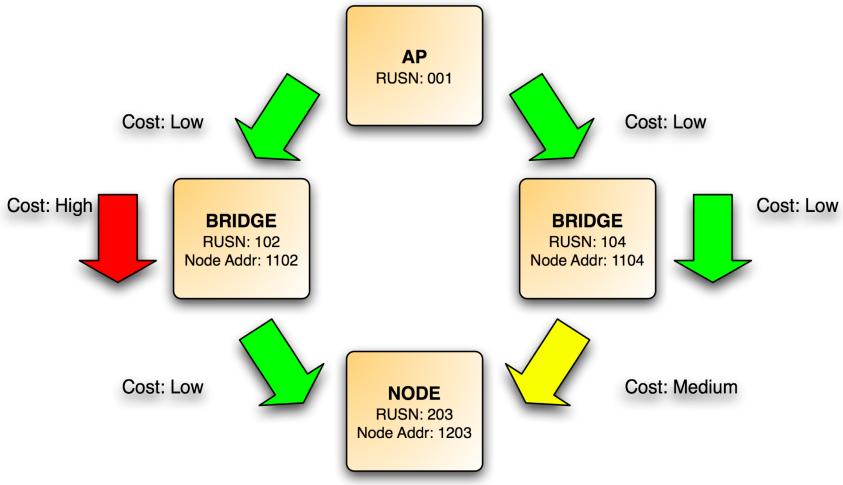

Figure 4.5: Route cost estimation.

to gather the information on the slave. The exact scheduling algorithm is out of the scope of this document. But the Transport Route Manager has all the remote information so the internal scheduler can make the best possible choice for a given scenario.

4.4.2 Scheduling

Despite the fact that the Transport Route Manager has all the remote information so the internal scheduler can be studied to make more adequate choices for a given scenario, the current implementation is a strict priority based scheduler, with round-robin for each priority.

Considering the NL Priorities, TL maps all priorities greater or equal to *0* to the ”low priority” NL Queue. TL Priority *-1* is mapped to ”high priority” NL Queue. All TL Priorities lower than -1 are mapped to ”super-high priority” NL Queue. At the slave side the TL processing of priorities is similar to the Master side, but the NL will treat ”super-high priority” requests as they were ”high-priority” requests.

Chapter 5

Transport Layer implementation

5.1 Mapping the Simulation module

5.1.1 Simulating the REMPLI system: a brief introduction

The starting point for the REMPLI Transport Layer implementation was the REMPLI system software simulation. In fact, during several months both system simulation and implementation were developed simultaneously and cooperatively, being the simulation a valuable tool for testing and debugging during all the implementation process. Although this document does not focus on REMPLI simulation, an introduction to this subject is necessary in order to understand the basis that support the implementation work.

Building a REMPLI device model

During the design phase, the Transport Layer concepts were tested on a simulation environment. The tool selected for this task was OMNeT++, an open source discrete event simulation environment, whose primary application area is the simulation of communication networks [53, 54]. OMNeT++ provides a component architecture for modelling systems, in which components (modules) are programmed in C++, then assembled into larger components and models using a high-level language (NED - Network Description language) [55].

The four main modules of the REMPLI Transport Layer – TRM, QM, NLI and RCIM – where first developed as OMNeT++ model components, being each module described as a C++ class. Each class contains all the functionality required by the three types REMPLI devices (Access Point, Bridge and Node):

- common functionality is immediately available to any of the three types of REMPLI devices;
- specific functionality of a REMPLI device is activated on compilation time.

This approach allows to build a Transport Layer tailored to the target REMPLI device model according to its type, and improves software development as modifications in common functionality are immediately reflected on all device types. These four Transport Layer modules can be assembled together we can produce three Transport Layer models[1], which are available to construct more complex models. Simulation models of the three types of REMPLI devices are defined assembling the models of tailor-made Transport Layer and Master/Slave medium interface [56]. Figure 5.1 illustrates the model of

[1] One Transport Layer for each type of REMPLI device.

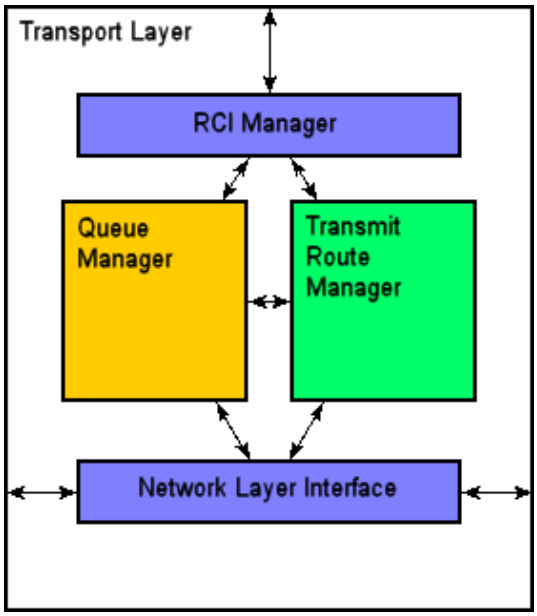

Figure 5.1: A REMPLI Bridge TL model (graphic captured from OMNeT++).

a REMPLI Bridge, where the four components and respective communication channels (represented by arrows) are visible. The RCIM and the NLI provide communications to exterior components. In this case, the Network Layer Interface has two communication streams pointing outside the Transport Layer, which are to be connected to the Master and Slave medium interfaces.

Emulating the Physical Layer

One task of the REMPLI project was the development by iAd GmbH (Germany) and LORIA (France) of a Physical Layer Emulator to validate the network protocol [57]. The emulator consists of a set of C++ classes, which allowed a simple integration in OMNeT++ projects. The emulator C++ classes receive information about the packets that were sent in a time slot and, when requested, calculate the transmission of those packets for a time slot and return the results. To integrate the emulator on the OMNeT++ a module was created (PLEmulator) that uses an instance of the emulator. This module interfaces with the emulator, being responsible for:

- Receiving OMNeT++ messages from REMPLI devices, reconstructing the REMPLI packets and schedule them in the emulator.
- Triggering the emulation of a slot time, calculating which stations receive which packets, and with which error status.

Modelling the system

Once all system components – REMPLI devices and Physical medium – were modelled, it is possible to create a system model containing any number of device instances and network morphology. Each instance of a REMPLI device receives unique parameters that unambiguously identifies it and define its behaviour.

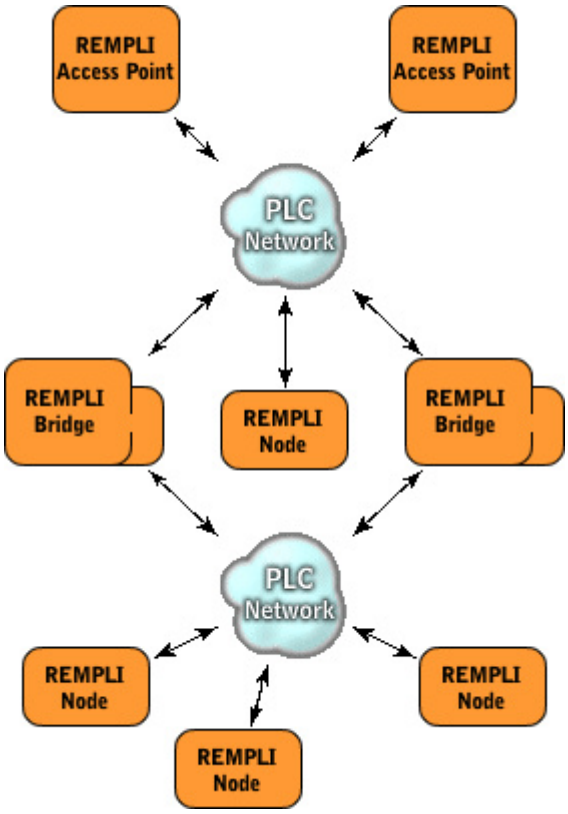

Figure 5.2: A sample simulation REMPLI network (graphic captured from OMNeT++).

Unique properties are assigned to each REMPLI device model, so it is possible to create a network model connecting several REMPLI devices, as depicted in Figure 5.2. These network models revealed themselves quite useful to test the behaviour of the three REMPLI Transport Layers and detect problems.

Further details about simulation description and results can be found in technical reports *"Characterizing the Timing Behaviour of Power-Line Communication by Means of Simulation"* [56] and *"REMPLI Discrete Event Simulation System"* [58].

5.1.2 Parallel development of Simulation and Implementation

When the implementation of the REMPLI Transport Layer started, the REMPLI system simulation was already in development and proving itself useful for the development and testing of the REMPLI Transport Layer protocols. Soon became clear that it would be useful if simulation code relating to TRM, QM, NLI and RCIM functionality – which was already subjected to testing – could be reused in the implementation of this layer, so time could be saved avoiding rewriting the code. Furthermore, it would be interesting if the files containing the code of the referred modules could simultaneously be used in the simulation and implementation. In this scenario, the functionality could be previously tested and debugged in the simulation environment and then directly transferred to the implementation. Later on, during integrations, the simulation environment proved more efficient to debug certain type of malfunction spotted on the lab testbed.

Despite the advantages indicated, there were some issues that had to be carefully addressed:

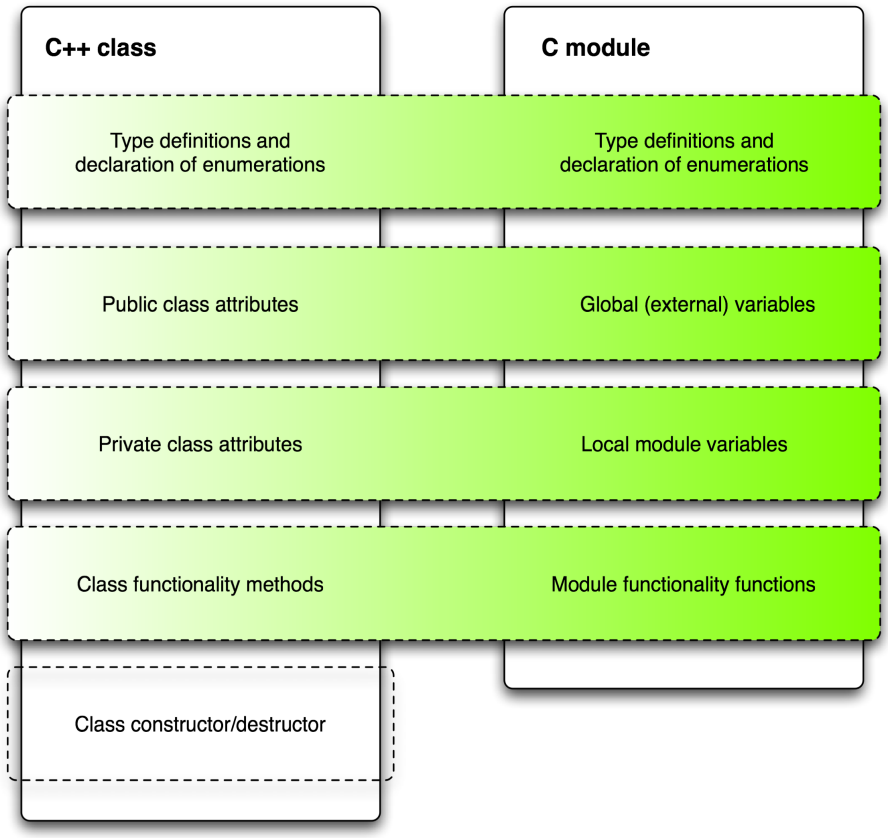

Figure 5.3: Mapping a TL component C++ class into a C module.

1. The simulation programming language is C++, while the implementation programming language is C. Even if C++ has evolved from the C language and the C language can be thought as a subset of the C++ language, C++ has significant differences from ANSI C.

2. The OMNeT++ simulation environment offers a uniform communication mechanism that allows the exchange of messages between model components. This mechanism API is available in the form of a C++ class and is the same for all components, inside or outside the Transport Layer. This mechanism is not reusable for the implementation, which must support diverse types of communication mechanisms between components, and between protocol layers.

3. The simulation environment simulates parallel execution of all Transport Layer modules, and a similar behaviour on the embedded system was expected.

In the simulation, each of the four main TL component modules – the TRM, the QM, the RCIM and the NLI – were coded as one C++ class. The functionality of a TL module completely enclosed in one C++ class, was translated to a C module, using the approach illustrated in Figure 5.3.

In this approach, type definitions and declarations of enumerations were kept unchanged because they follow a unique C/C++ syntax.

The behaviour of the TL was analysed so call attributes could be mapped as in Figure 5.3. On a REMPLI TL there is a single instance of each TL component module so, in the simulation, there was only a single instantiation of each component class on each device model. This characteristic allowed

to map class attributes into variables, because each class attribute would be unique during execution time. Private attributes were mapped as local variables, internal to the C module, while public attributes became global (`extern`) variables, accessible outside the C module. Special attention was given to assure that the same name was not shared by multiple public class attributes, so they would not be translated into multiple global variable definitions in the TL implementation.

Mapping class methods into module functions became straightforward, after observing the characteristics of the TL. In each class, there is one single method that is invoked from outside the class instance: the method that is supposed to handle a message received by the TL module. As class functionality is required, this method will call the other methods of the same class to achieve its objectives. This way, the methods in the simulation classes would just refer to methods and attributes inside the class instance and the resulting code was possible to write completely in C compatible syntax because the operation with other objects was not necessary, and was possible to map the methods into module functions[2]. The initialization procedure of each REMPLI TL component was coded into a method that was mapped into a module function. Although the constructor methods had not direct mapping into the respective implementation modules, each constructor calls the corresponding initialization method to set the initial values of the new component instance.

The resulting C modules represent the Object Oriented (OO) structural perspective, instead of the functional perspective typically found in C projects. This sort of code division as proven itself quite useful

This case supports that is it possible to write an application in C language, following an OO specification, despite the absence of useful properties such as class inheritance, polymorphism and the constructor mechanism. The OO specification of the simulation was found quite useful to design the architecture of the implementation of the TL, producing a clean and logical division of functionality over the modules. The following advantages were verified:

- Simplified design and development of each module, just taking in account the functionality of the respective module.

- Simplified testing and verification of the TL, because it was easier to identify which module was presenting an abnormal performance and, consequently, to track and trace the misconceptions present in the code.

5.1.3 Keeping source code simultaneously C and C++ compatible

When the Transport Layer implementation phase started, there was already several thousands of lines of C++ code being tested in the simulation. In the simulation, each one of the four main REMPLI TL modules was coded as a class with its code distributed in three files:

the Network Description file (*.ned) , required by OMNeT++ where the basic structure of a component is described;

the header file (*.h) , where the class is declared so it can be instantiated in other files;

the source file (*.cpp) , where the class functionality is defined.

[2]Some minor modifications in the header of each method was required to turn it simultaneously C and C++ compatible.

Network Description files

In a Network Description file it is defined the model structure (topology) of a component, stating which parameters, gates, submodules and connections a component has [55]. Parameters are values which are meaningful on the component such as an address, a serial number or a clock frequency. Modules are connected by gates and submodules inside a component can be interconnected too, by their gates. The Network Description File is of no interest to the Transport Layer implementation.

Header files

In the simulation, the header file of a TL module contains its class declaration; additionally, macros and structures can also be declared in this header file. Listing 5.1 is an extract of the original simulation TransmitRouteManager.h file, which illustrates the typical sections that can be found in a header file.

Listing 5.1: Extract of TransmitRouteManager.h.

```
1  #ifndef __TRANSMITROUTEMANAGER_H
2  #define __TRANSMITROUTEMANAGER_H
3
4  #include "tldefs.h"
5  #include "cx_msg.h"
6
7  #ifndef __PACKED__
8  # ifdef WIN32
9  #  pragma pack(1)
10 #  define __PACKED__
11 # else
12 #  define __PACKED__ __attribute__((packed))
13 # endif
14 #endif
15
16 #define TRM_COST_MULTIPLIER (10.0)
17 #define _TRM_SLAVE_RIT_MAX_INC (5)
18 #define _TRM_MASTER_RIT_MAX_INC (5)
19
20 // (...)
21
22 typedef enum trm_pdu_type {
23   TRM_PDU_MNL_ACCESS_TABLE_ADD = 0x01,
24   TRM_PDU_MNL_ACCESS_TABLE_REMOVE = 0x02,
25   TRM_PDU_BRIDGE_ROUTE_ADD = 0x03,
26   TRM_PDU_BRIDGE_ROUTE_REMOVE = 0x04,
27   TRM_PDU_STATUS_UPDATE = 0x05,
28   TRM_PDU_LINK_QUALITY_UPDATE = 0x06
29 } trm_pdu_type;
30
31 typedef struct trm_pdu_union_access_s {
32   rusn_t       rUSN __PACKED__;
33   rci_addr_t   rciAddr __PACKED__;
34 } trm_pdu_union_access;
35
36 // (...)
37
38 class TransmitRouteManager : public cSimpleModule
39 {
40   rusn_t    _tl_myRUSN;
```

```
41    tltype_t  _tl_tlType;
42    double    _tl_slotTime;
43    sint16_t  _tl_masterDataLength;
44    simtime_t _tl_readLLEntriesInterval;
45
46    // (...)
47
48    void _TRM_ApGetStatus(cx_msg_t recvMsg);
49    void _TRM_ApGetLiveList(cx_msg_t recvMsg) ;
50    void _TRM_ApGetLLEntry(cx_msg_t recvMsg) ;
51
52    void _TRM_NodeSetStatus(cx_msg_t recvMsg);
53    void _TRM_NodeRespTimes(cx_msg_t recvMsg) ;
54
55    // (...)
56
57    Module_Class_Members(TransmitRouteManager, cSimpleModule, 0);
58
59    virtual void initialize();
60    virtual void handleMessage(cMessage * recvMsg);
61    virtual void finish();
62  };
63
64  #endif  // __TRANSMITROUTEMANAGER_H
```

Listing 5.1: Extract of TransmitRouteManager.h.

In this example, the code before the class definition includes macro, struct and enum definitions. Most of these definitions are necessary for both the simulation and implementation functionalities and are C and C++ compliant.

However, the class definition is C++ exclusive and it will not be processed by a C compiler, so it is necessary to modify the source code and turn it C complaint. The most effective way to achieve this task it to use the C preprocessor conditional inclusion directives and the `__cplusplus` macro[3] to hide or substitute C++ code from the C compiler.

The class definition can be divided in three main sections:

- the class header,
- the definition of class attributes, and
- the declaration of class methods.

The class header, as seen in line 38 on Listing 5.1, is not processed by a C compiler. The solution was to hide it from the C compiler, containing it inside a `__cplusplus` macro conditional inclusion directive.

The definition of class attributes includes attributes that are only required for the simulation, and attributes that are fundamental to the functionality of the module, independently if the code is executing under simulation or implementation environments.

Simulation-exclusive attributes can be safely hidden inside a `__cplusplus` macro conditional inclusion directive, after thorough verification that their use is not required in the implementation.

Regarding the attributes that are used both in simulation and implementation, their definition in the header file means that they will became global variables in the context of the source files that

[3]The `__cplusplus` macro is generally defined by the C++ compilers, before calling the cpp.

include the header file. If several source files include one header file in which a particular variable is defined then, after compilation, we will obtain several object files, each containing a global variable with the same name. This fact raised a concern: how would the linker handle these object files when assembling the final executable file. Experimental tests[4] confirmed that for each variable defined in a header file, the linker would generate a single global variable, shared by all program modules. Although defining a global variable in a header file is considered poor programming practice, the development team has decided to maintain this approach while implementation is not detached from simulation for the following reasons:

- Splitting attributes/variables definitions could be troublesome, as future modifications in a variable definition would have to be performed in two places.
- This solution was working on the embedded boards.

Once the implementation become independent from the simulation, the definition of these variables would be done inside a source file and, if necessary, an `extern` variable declaration could exist in a header file.

Source files

Source files contain the definition of module functionality, in the form of class methods. The main task to keep C and C++ compatibility was to transform the methods in such a way that could be interpreted as functions by the C compiler, if possible without having to duplicate the source code or rewrite it from scratch. This task was, somehow, facilitated by the nature of already implemented simulation code, where methods do not call methods from outside the class and do not create new objects, looking much like procedural programming. Whenever specific simulation procedures were performed[5], they were hidden from the C compiler using the `__cplusplus` macro. The same macro was used in some procedures in which was totally impossible to keep C/C++ compatibility, and code had to be written specifically for C.

The method prototypes were not C compatible because, besides the method name, they included the class scope and also the scope operator which had to be hidden from the C compiler. This was easily achieved generating a macro in each source file that was used before the method name, as in the example in Listing 5.2. This macro would be set empty for the C compiler, and would be set to a text containing the scope and scope operator for the C++ compiler.

Listing 5.2: Hiding the method scope from the C compiler.

```
1  // (...)
2  #ifdef __cplusplus
3    Define_Module( QueueManager );
4  # define _QM_FUNC_PREFIX_ QueueManager::
5  #else
6  # define _QM_FUNC_PREFIX_
7  #endif
8
9  // (...)
10
11 void _QM_FUNC_PREFIX_ _QM_initialize ()
12 {
```

[4] This issue was tested, using gcc in Linux and Mac OS X, and in the Hyperstone toolchain.

[5] These procedures were mainly for recording data for statistics.

```
13    // (...)
14  }
15
16  void _QM_FUNC_PREFIX_ _QM_cxHandleMessage(cx_msg_t recvMsg)
17  {
18    // (...)
19  }
20
21  // (...)
```

Listing 5.2: Hiding the method scope from the C compiler.

Application defined types shared the same name even if they have different structure definitions for C and C++, so method/function parameters could be kept unchanged in the method prototypes.

After these modifications have been introduced, a small set of rules that allowed the simultaneous development of simulation and implementation was defined, sharing the same four TL modules declaration and definition files.

5.2 Architecture of the implemented TL

As already described in previous sections, the main building blocks of the REMPLI TL are the QM, the TRM, the RCIM and the NLI. But besides the main modules, the architecture of the implemented TL also relies on additional modules that provide services required for the TL to operate as required. Services such as intermodule communications and direct interfacing with adjoining software layers were implemented as independent modules, so the extra functionality would not affect the main modules design.

Figure 5.4 presents the block diagram of the actual components present in the implemented TL and data flows with software blocks in adjacent layers. As already described in chapter 4, the four main modules perform, in brief, the following functions:

the RCIM switches messages between the Driver DeMux and the TL modules QM and TRM;

the QM fragments data to fit network packets and assembles network data chunks to offer the Driver DeMux a connection-oriented service;

the TRM gathers information about the network topology and status and instructs the QM to how to route the network packets;

the NLI switches messages between the NL and the TL modules QM and TRM.

The *DeMux processor* is the module that is responsible to deal directly with the Driver DeMux. This module manages the Notification channel and the Command channels and translates data between the RCI specification and the internal TL message formats, acting as an intermediate between the RCIM and the upper layer. This functionality was left outside the RCIM, to make the RCIM independent from the implementation of the RCI specification. Additionally, the RCIM is freed from managing the connections and concentrates only in switching messages between the TL components (QM and TRM) and the upper software layer.

On the opposite side of the TL, the *NL processor* performs a similar function to the DeMux processor, translating the data between the NL and the internal TL message formats, as an intermediate between the NLI and the lower protocol layer. Depending on the type of REMPLI device – Access

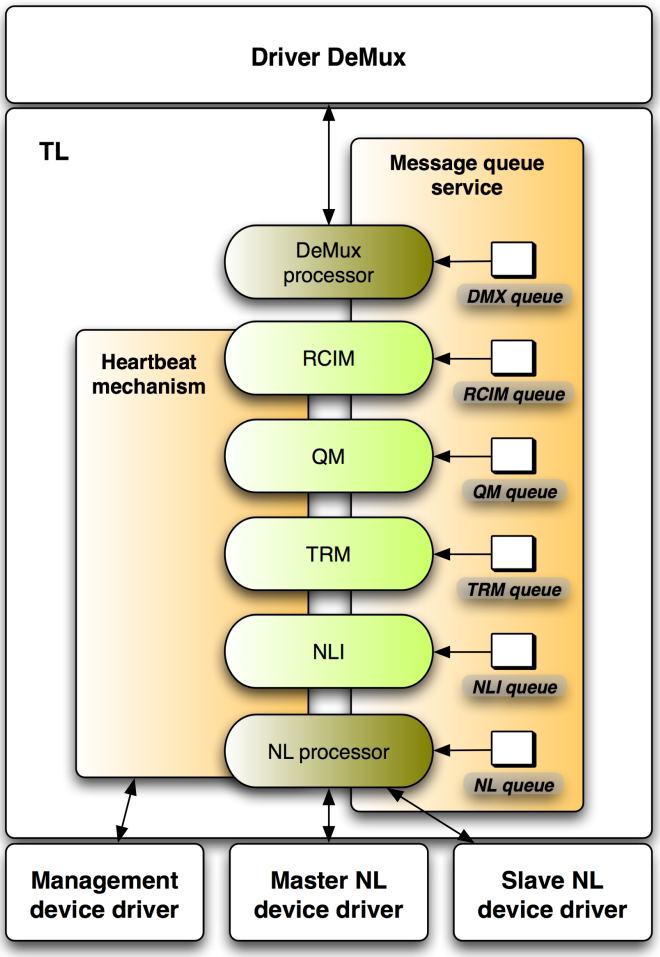

Figure 5.4: TL components and internal services.

Point, Bridge or Node – the NL processor must handle communications with Master and/or Slave NL device drivers. This module allows the NLI to complete focus on switching messages between the NL and the TL components (again, QM and TRM), totally unaware of the operations in progress with the NL.

The *Message queue service* is a mechanism that allows communications between the six above referred modules. This mechanism offers one mailbox to each module, in which all modules are allowed to insert messages. Each message is of a certain type, which identifies the processing to be carried out by the module; the message may also carry data necessary for the module to complete its task. Each mailbox contains a message queue and respective synchronisation mechanism.

Every REMPLI device has a software component called the *Management Driver*. The Management Driver controls several aspects of the system, from boot setup to software components monitoring. Every software component must periodically indicate the Management Driver that it is working in perfect condition, sending an indication, called *heartbeat*. If the heartbeat is not received, the Management Driver will make sure the non-responding process is terminated and then launches it again. The *Heartbeat mechanism* ensures that modules from RCIM down to NL processor are responding, before sending the heartbeat to the Management Driver, keeping this way the TL operating in proper conditions.

5.3 Concurrent processing

Communication protocol stacks are usually built entirely inside the kernel, up to the transport layer of the OSI. This tendency is normally justified by the processing performance necessary to match the time requirements that are specified for the transport layer. However, in the specific REMPLI case, there were several factors indicating that user-space was the more suited execution area for the transport layer.

One of the most evident factors was the different time requirements of the REMPLI network and transport layers. The REMPLI NL has very stringent time requirements because it must face the time-slot characteristics of the REMPLI communication medium. For every packet received, the network layer sends an acknowledge confirmation and if the network layer is not ready to send one packet on its time-slot, it will have to wait one cycle for the following time-slot, which can have a negative impact if this type of situation is recurring in a low bandwidth network such as the REMPLI. Contrarily to the REMPLI NL, the TL does not have such strict time requirements. When the TL processes a command that requests a reply from its counterpart, the first concern is to get a response that comes from the NL informing if the command was successfully sent over the network. After this, the timeout limit to get a reply from the addressee device can be set to several seconds and such reply will eventually arrive in the form of an event notification. This desynchronisation between the NL and TL resulting from the different time requirements suggests that there is not an imperative need to use the kernel space specific properties for the REMPLI TL.

Another relevant factor was the complex nature of the REMPLI TL functionality, specially on REMPLI Bridges. Having a lightweight kernel improves its performance, specially on small embedded systems and is normally a good policy to keep out of the kernel everything that is less demanding on performance requirements. Less complexity inside the kernel also means smaller probabilities of serious system instability due to programming misconceptions and errors.

Having the REMPLI TL outside the kernel also mean that it could profit the services the kernel offers, such as multithreading and socket communications, which were thought as probably useful tools for match with some of the requirements of the TL. It would also be easier to upgrade and

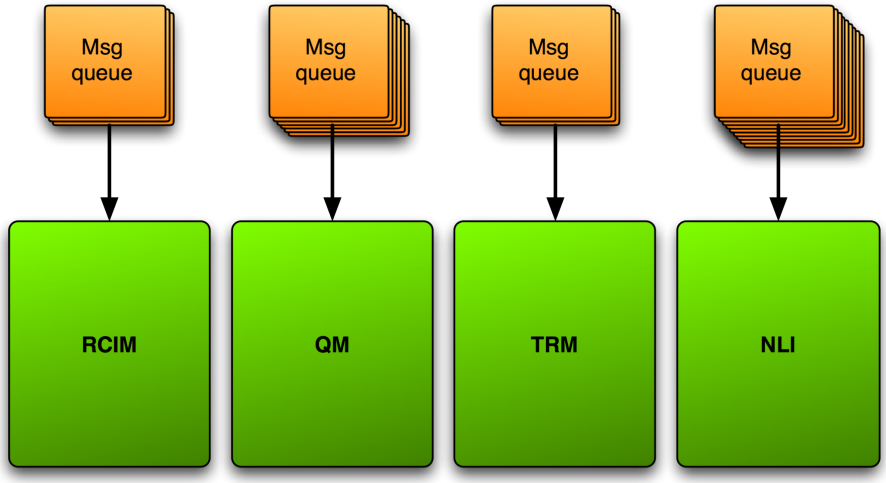

Figure 5.5: Each TL module receives tasks from its corresponding message queue.

integrate new functionality to the TL.

These factors were pondered during the design phase, and decided that the REMPLI TL would be an user-space application.

Each module has an associated First In, First Out (FIFO) message queue from where it fetches the messages sent by other modules, where it can find the the tasks to perform. As it can been seen in Figure 5.5, each module contains a mailbox to where other modules may place a message containing the command and related data for the module to process. One module is expected to process the messages sequentially and may, eventually, generate new messages destined for other modules. As message queues are independent from each others, some queues may present more activity than others, reflecting the stress over its related module. In the represented snapshot, the NLI message queue has more messages than, for instance, RCIM, which may indicate that the NLI is being more solicited than the remaining modules.

The timing behaviour of the message queues is unpredictable, once the system is working on the field, since the exogenous solicitations (either from the superior and inferior layers) are impossible to predict. This characteristic of the system makes impossible to determine the optimal order that modules should execute. Furthermore, the time taken for a module to process a message may vary, depending on the message type and the data carried. Thus, the four main modules of a REMPLI TL were specified to execute concurrently, in such a way that message queues would not suffer from starvation. Two execution mechanisms were considered: cyclic polling and multithreading.

5.3.1 Polling the message queues

In this mechanism, the four message queues are polled cyclically. If a message queue is not empty, the respective module is called to process the oldest message in the queue and once the tasks is completed, the polling cycle will head to the following message queue. Obviously, when a message queue is found empty, the respective module is not called, and the next message queue is polled.

This mechanism has its advantages in its simplicity:

- It offers equal opportunities to execute for the all the modules.

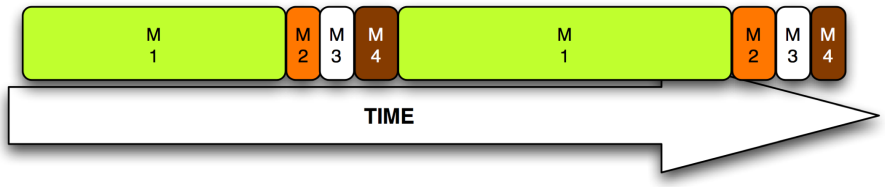

Figure 5.6: Module M1 uses more time than the remaining modules.

- The access to the message queues is always exclusive, since there is one single thread accessing them, so synchronization mechanisms are not necessary.
- It does not require special libraries[6] to be ported to the system's architecture.

Nonetheless, there are also some disadvantages:

- If one module usually takes longer times to process its messages, it may end up monopolizing the processor, as seen from the timing perspective, so the other modules will have to wait too long to process their messages.

  In the example depicted in Figure 5.6, module M1 is regularly taking much more time to perform one task, when compared with the other modules. Under this scenario, the fastest modules will always appear to process at the speed of the slowest modules.

- If message queues are empty most of the time, the TL process will never be idle, because it will continuously scan the message queues. As a result, an unnecessary excessive consumption of power and consequent heating of the embedded system may occur. It may also undermine the overall performance of the embedded system, as processor-time that could be allocated to other processes, if the TL process was known to be idle, will be spent on this scanning cycle.

5.3.2 Multithreading the modules

The four modules can also execute concurrently if they are set to run in independent threads, inside the same process. The OS kernel would be responsible to assign each module a time-slice for execution, temporally interleaving the threads. The modules are so allowed to start processing their own messages without waiting for any other module to complete its processing, which obviates the monopolization of process-time by one module and faster modules may see their throughput improved. Figure 5.7 depicts a situation in which module M1 is preemptively suspended twice while processing one task. During these two periods, modules M2 and M4 are allowed to process two small tasks. Meanwhile, module M3 processes one longer task that does not fit in one single time-slice, being also preemptively suspended.

Access to the queues to insert or retrieve a message on this multithread environment must be controlled by a synchronization mechanism, to avoid race conditions[59]. Modules can be ensured exclusive access to one message queue by means of one mutex associated to the queue, preventing that simultaneous operations on the queue can be performed simultaneously.

Comparing with the polling mechanism described in section 5.3.1, there is an increase of complexity in the access to the message queues, because of the additional mutual exclusive control mechanism. But, besides the added complexity, there is also a positive secondary effect: if one message

[6] Like Portable Operating System Interface (POSIX) threads, for instance.

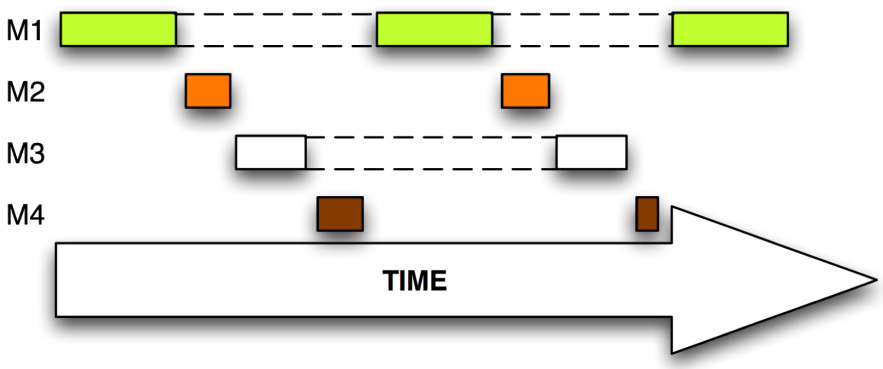

Figure 5.7: Multithread perspective of modules' executions.

queue is empty, the thread of the respective reader module can be blocked, leaving processor-time to other threads that are active. And during low activity periods, if all message queues are empty, the whole process may block and release processor-time for other processes or reduce processor power consumption and heating.

In the early stages of the implementation phase, the POSIX threads were not implemented in the kernel which would prevent the use of this technique. Once the POSIX threads were successfully ported to the embedded system's architecture, the multithreaded TL approach was selected because the advantages were in consonance with the desired system behaviour.

5.4 Intermodule communications

The component modules of the REMPLI TL use messages queues to communicate between each other. Each module has its own mailbox which the module is the exclusive reader. It is from its mailbox that the module fetches the messages containing the indication (command) of the task to perform and possible data to process.

The message queues are implemented as singly-linked lists [60], where messages are fetched from the front of the queue and new messages are inserted at the tail of the queue. Some special messages, such as timer or heartbeat messages are inserted at the front of the queue, because they take priority over regular messages due to more strict timing restrictions.

Operations on the lists are supported by an array of queue control structures (one control structure for each linked list). Figure 5.8 represents one message queue, with its nodes pointing to the next node, along with its respective queue control structure.

5.4.1 Queue Control Structure

The Queue Control Structure contains the following elements:

- pointers to the extreme nodes of the linked list of messages,
- one mutex to assure exclusive access to queue operations,
- one conditional variable to signal the reader module.

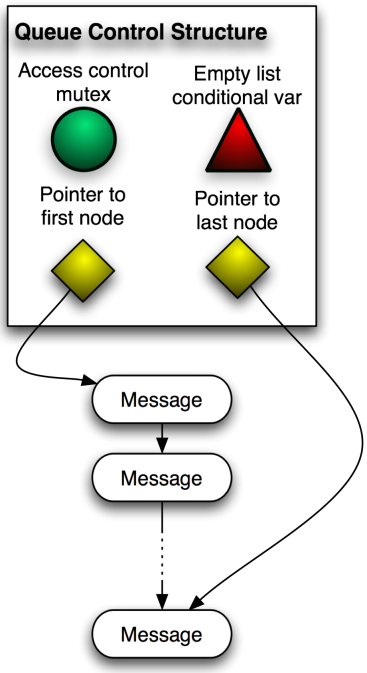

Figure 5.8: Message queue and respective Queue Control Structure.

The pointer to the first node of the list is used to locate the next message to be processed by the reader module, while the pointer to the last node is used to insert a new node in the list. The pointer to the last node is not necessary on a singly-linked list like this one in the example, since the last node can be reached crossing the full list, starting from the first node, but it can decrease significantly the time required to insert a new node, if the list contains several messages, because it offers direct access to the end of the queue, paying only a 32-bit memory overhead on the control structure.

Operations on the message queue – to insert or retrieve a message – involve multiple pointer value updates that may lead to race conditions in a multithreaded concurrent environment. One solution to avoid this problem is to assure that all pointer manipulations in one operation are all performed as one single atomic transaction [61]. To avoid conflicting changes of pointers values, all threads have to obtain the queue mutex to enter this critical section and then release it once the pointers are updated; if one thread does not immediately obtain the mutex, it will wait until the mutex is free.

The activity of these queues is a variant of the classic producer-consumer problem [62], in which there is one consumer which is only allowed to read if the queue is not empty and multiple producers that may insert messages in an boundless queue. In this scenario, once a producer obtain the mutex, it can seamlessly insert the message into the queue, but when is the consumer to obtain the mutex, the thread must first verify if the queue is not empty. If the queue is empty, the consumer thread should release the mutex – allowing the producers to feed the queue – and wait until a message is received. The conditional variable in the control structure acts in conjunction with the mutex [61], so the consumer thread may atomically block on the conditional variable and release the mutex, suspending its execution while the message queue contains no messages. Every time a producer thread inserts a message into the queue, it signals the conditional variable before releasing the mutex, waking up the possibly blocked consumer thread which may, in turn, recover the mutex and fetch the message from the list.

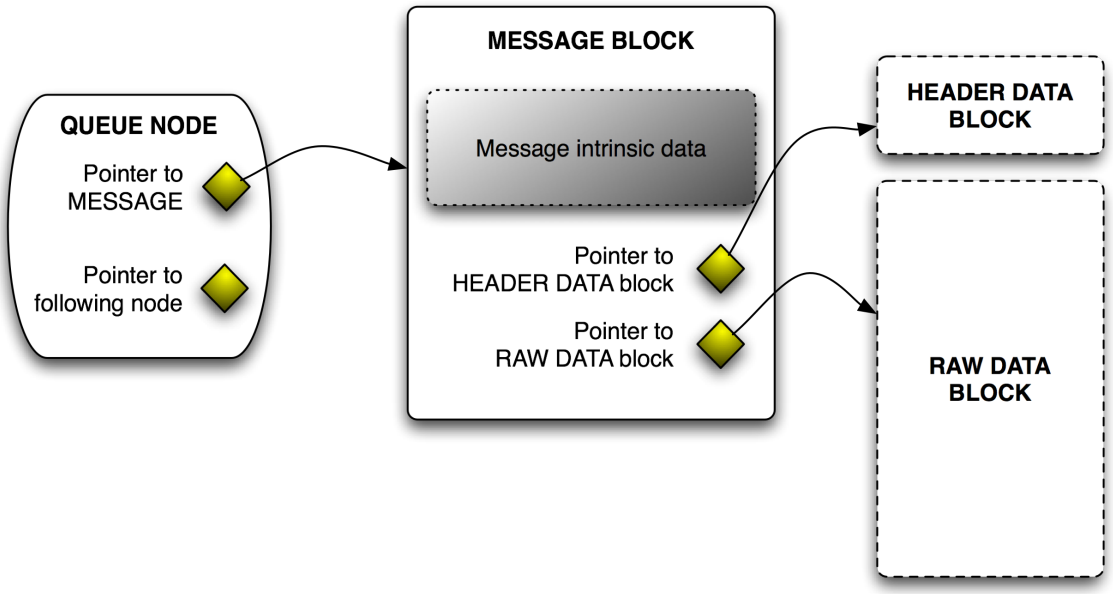

Figure 5.9: Message component building blocks.

The critical section of queue operations (insertion and retrieval) is kept to the minimum necessary instructions (almost only pointer values manipulations), in order to give the highest availability to queue access. When multiple threads try to simultaneously access the same queue, the threads are placed in a waiting list and operate one after another. These waiting lists were expected to not grow long and that time required for a module to operate one queue was kept short, because the pressure over one queue should not be heavy due to the following factors:

- the time required to process one message is supposed to be significant longer than the time required to insert a message in a queue and
- the operation to insert a message is blocking, so one module will take some time before it tries to insert another message in the same queue;
- the time required for a queue operation is independent of the size of data carried by the message, as only some pointer manipulations are performed (there are no memory block copies), and is kept as short as possible.

5.4.2 Message queue nodes

Messages are created on demand in dynamically allocated memory, and are composed by one mandatory block – the MESSAGE block – and two optional – the HEADER DATA and the RAW DATA blocks – depicted in Figure 5.9.

When a message is inserted in a queue, it is generated in dynamically allocated memory a NODE block , which has one single purpose: to point to the MESSAGE block while it waits in the queue. Once the message is retrieved from the queue, the NODE block is immediately destroyed.

Contrarily to the ephemeral existence of NODE blocks, one MESSAGE block may live long enough to be processed sequentially by multiple modules before being discarded. The MESSAGE

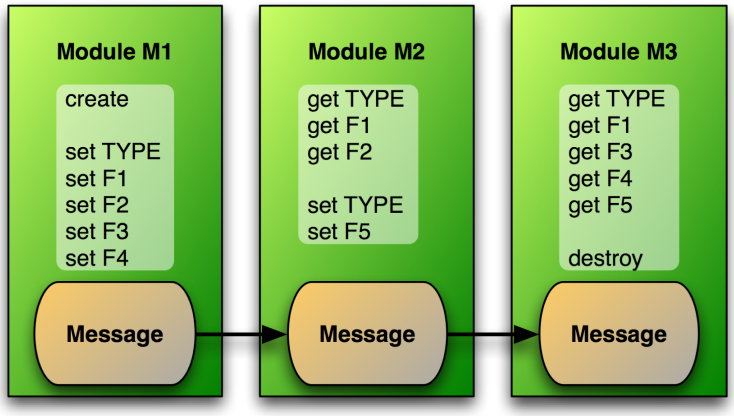

Figure 5.10: Sequence of operations with persistent ”fits-all” message.

block is a fixed-sized data structure, a collection of 20 data fields required by all the TL modules, for all possible module commands. The module that creates one message sets all relevant fields, leaving the remaining fields unset, and passes the message to the recipient module. The recipient module will process the data in the message accordingly to the message type, and may modify or add data to the message before sending to a subsequent recipient module, or discard the message if no further processing is required.

At first sight, this single ”fits-all” structure that is pushed from module-to-module like a product in an assembly line exhibits a clear memory usage overhead, since it has a larger memory footprint than any optimized structure for each one of the 200 message types available. Nevertheless, taking in account the TL characteristics, this memory overhead can be compensated with time savings in memory operations. Many of the events that force one module to generate a new message will involve the activity of 2 or more subsequent modules. Considering the simple example[7] in Figure 5.10, module M1 generates a new message and sets the message type and fields F1 to F4 before sending it to module M2. For the received message type, module M2 needs to know the data in fields F1 and F2 and, after processing, it sets the result into field F5, updates the message type and forwards it to module 3. For the received message type, module M3 requires the data in fields F1, F3 and (set by module M1) and in field F5 (set by module M2) and after processing, it destroys the message. Comparing the same sequence of operations using optimized structures and non-persistent messages, as depicted in Figure 5.11, it is clearly observable that there will be more time consumed on memory allocation and selective attribute memory copy operations. For the sake of simplicity and less time consumed on memory operations, the ”universal” message structure was selected for the internal TL communications.

The HEADER DATA block contains data that is exclusively commuted between the Driver DeMux and the TL, as part of the Parameter Block, a segment of the messages exchanged between these two software layers. According to the RCI Protocol [32, 51], all messages exchanged between the Driver DeMux and the TL can be split into two segments: the Header and the Parameter Block. The Header is a fixed structure containing four fields, namely the Message Type (8-bit char), the Command Code (8-bit unsigned), the IPC Transaction ID (32-bit-unsigned) and, depending on the Message Type, the Length of Parameter Block or Error Code (32-bit unsigned) [8]. Depending on a combination of

[7]Note that in real application, modules work with more data fields than represented in this example.

[8]The Error Code applies to Error messages, while the Length of Parameter Block applies to Command and Response

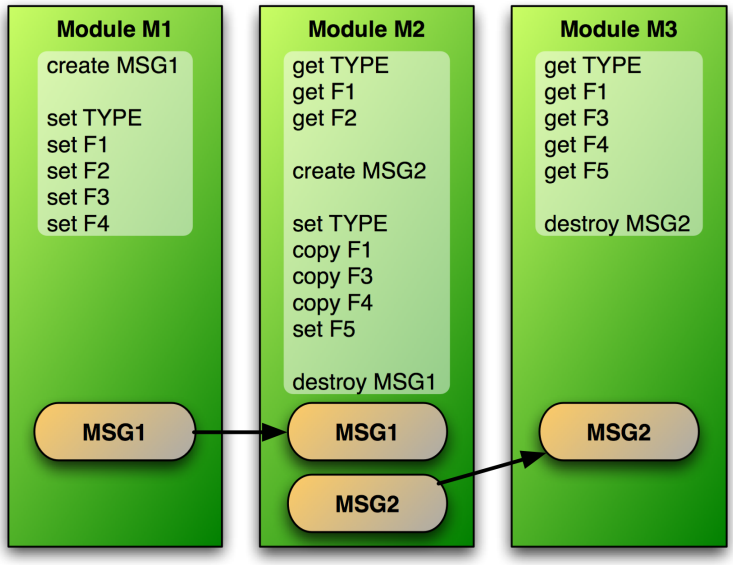

Figure 5.11: Same sequence as Figure 5.10 with optimized-structure messages.

Message Type and Command Code, the message may carry or not a Parameter Block. The Parameter Block may contain well defined parameters which are usable by the QM and Driver DeMux to process the packet, followed by PDU data, an opaque variable-sized field containing binary data oriented to the drivers/applications. The HEADER DATA block is an image of the segment of visible parameters inside the Parameter Block.

The RAW DATA block is an opaque container for binary data which constitutes the payload oriented to the drivers/applications. This payload is not supposed to be understandable by the TL, but is subject of processing in the QM:

- the payload must be inserted in a transmission queue and, possibly, fragmented into small chunks, to fit in network packets, before being delivered to the NL;
- on reception, the QM must assemble the chunks and then send the complete payload to the Driver DeMux.

5.4.3 Scheduled Messages Service

A service of scheduled messages was developed, in which one module sets one message to be inserted in its own message queue, after a defined period of time. This service is used when a module, in presence of a certain event, needs to be reminded to perform a specific task in the near future. When this mechanism is solicited, a new message is set by the module and handed to a new born thread which will, in turn, sleep for a determined time and then insert the message at the end of the message queue of the same module, before finishing its execution. The module will then fetch the message from its queue and process it accordingly to its type[9]. This mechanism only assures that the message is received after the determined period of time.

messages.

[9]The message may be destroyed at the end of processing or may be updated and forwarded to another module.

5.4.4 Timer Service messages

The Timer Service was implemented as an optimization of the Scheduled Messages Service, to serve in cases of cyclic periodic tasks. When a module starts this service, it generates and sets a new message, and passes it to a new thread. This thread will repeatedly sleep for the determined time period and, after waking up, insert the message on the head of the message queue, so the module will fetch the message as soon as possible reducing the associated delay. Then, the module performs the routines associated to the message type, never destroying the message: the message is kept in memory so the timer thread can reuse it endlessly, avoiding unnecessary and repetitive memory allocation/release operations.

5.5 Processing communications with the Driver DeMux

According to the RCI specification [51], communications between the REMPLI TL and the Driver DeMux are established by means of connection-oriented internet sockets[10], enabling two types of channels – Notifications and Commands – as explained in detail in chapter 4 (section 4.1). The interface between these two layers allows one single Notification channel and multiple Communication channels. For both types of channels, the Driver DeMux acts like a client, always taking the initiative to ask for a new connection while the TL is listening for new connection requests in specified ports.

The Command channels support request/response transactions: the Driver DeMux issues commands and receives responses or error indications from the TL. The command and respective response/error share the same IPC transaction ID, a number generated inside the Driver DeMux, so it is possible to issue multiple commands without waiting for responses (see Figure 4.2) and later pair the responses/errors with the issued requests. IPC transaction IDs are generated independently in different Command channels and may overlap, so it is important that the TL knows for certain to which channel the response/error should be sent.

The Notification channel is used to inform the Driver DeMux about events that occured in the lower layers, so messages always travel from the TL to the Driver DeMux. Such events may be, for instance, data received from the network, a network status event or an alarm. In this channel, data always travel from the TL to the Driver DeMux. The messages sent to the Driver DeMux carry an IPC transaction ID set to 0 (zero).

The RCI protocol at the Access Points is comprised of 8 commands with respective success or error responses and 4 notifications, while at the Nodes is comprised of 5 commands with respective success/error responses and 3 notifications. Since above the TL, Bridges are seen as a common Node[11], they share the same RCI protocol implementation as the Nodes. The RCI protocol implementation for Access Points is exclusive for this type of REMPLI device.

It was apparently natural that the DeMux Processor would be composed by two main components: one component to handle the Notification channel operations and a second component to manage the Command channels. An additional component routes the messages received from the TL to the Command or Notification components. Therefore, a solution in which multiple threads would perform simple tasks, instead of one single thread that would include all the module functionality, was designed and implemented. To address several issues such as channel status detection and availability, this module resulted more complex, as seen in a general view in Figure 5.12.

[10] Also known as stream sockets.

[11] Above the TL, the Bridge has the same kind of drivers and applications as the Nodes, because they are slaves of Access Points. The Bridge may be seen as a Node that additionally can perform data transfer between two REMPLI network segments. This added functionality goes up to the TL level and is transparent for the Driver DeMux.

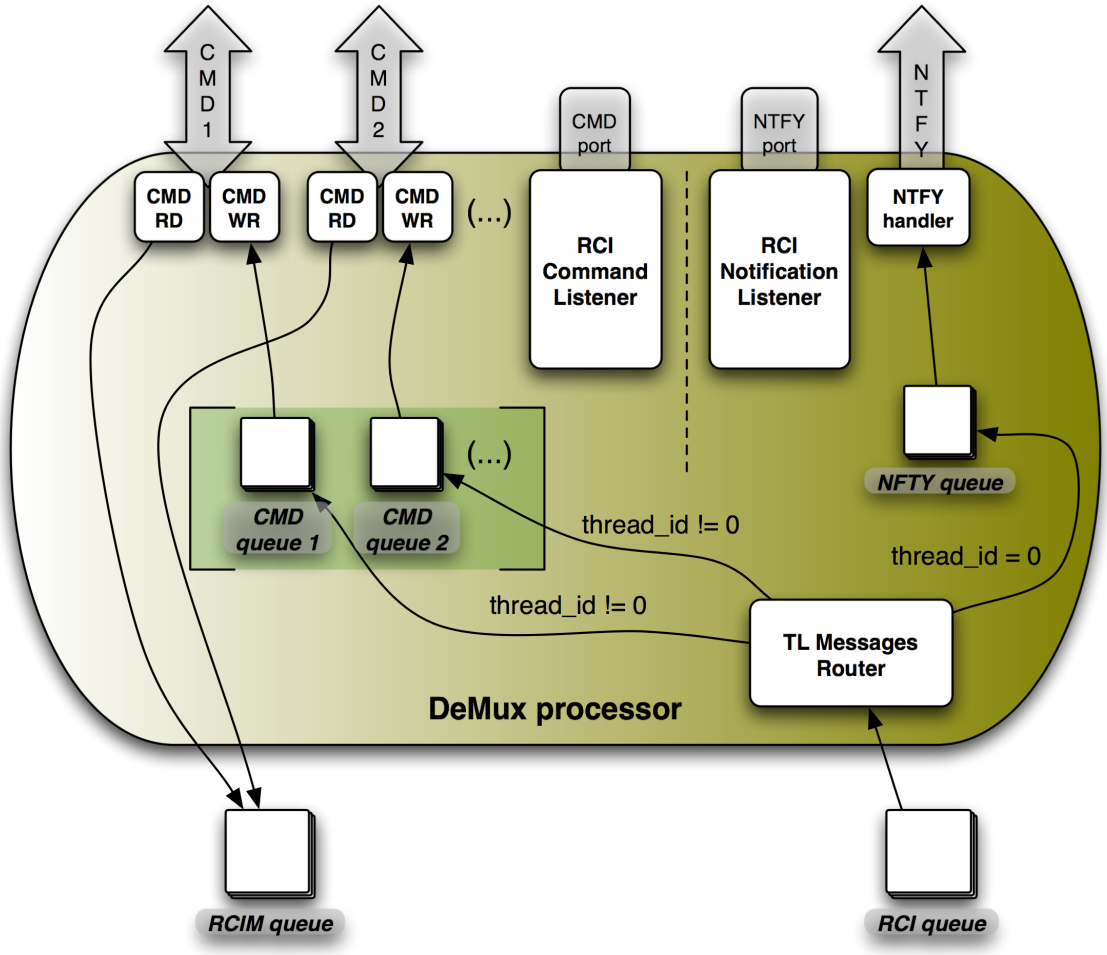

Figure 5.12: DeMux processor general view.

The TL has to be receptive to accept new connection requests from the Driver DeMux and so, for each type of channel one thread is set to listen on the specified target socket ports. Therefore, the *RCI Command Listener* listens for requests at the Command port while the *RCI Notification Listener* waits for connections at the Notification port.

5.5.1 Handling the Command channels

When a connection request arrives at the Command port, the RCI Command Listener accepts the call and generates a new thread which will be in charge of receiving the commands from the DeMux: the *Command Reader*, depicted as CMD RD. This thread creates one message queue (depicted as "CMD queues") where messages from the TL containing responses/errors from previously issued commands will be buffered before being sent to the DeMux, and identifies it with its thread id. Once the queue is set, the Command Reader generates a new thread which will act as its counterpart, and write data to the DeMux through the same Command channel – the *Command Writer*. The Command Reader will then enter in its processing cycle, reading commands from the associated channel. Every command read generates a new TL message which is inserted into the RCIM queue, to be routed to the destination module.

On the other hand, the Command Writer, depicted as CMD WR, receives the socket file descriptor corresponding to the command channel and the location of the message queue generated by the Command Reader. The task of the Command Writer is to collect responses and error indications from the associated command queue and send them to the Driver DeMux through the same command channel from where the commands were received.

Since the same value of IPC transaction ID may arrive to the TL from different command channels, an extra value is required to pair the issued commands to the respective response/error indication, so information about the status of one command is sent through the same channel. The Command Reader thread id identifies which reader has received the command and, therefore, the associated channel. This is the reason why every command queue is identified by the thread id of the Command Reader that has generated it, so response/error messages can be inserted in the right queue and be processed by the appropriate Command Writer.

Figure 5.13 presents one *Command Queue*, where the queue nodes point to the TL messages that carry the data to be sent over the command channel. The *CMD Queue* structure has a mutex to assure exclusive access for the insert and remove node operations and a conditional variable that indicates if the queue is empty. the conditional variable is used for by the Command Writer thread to block when there are no messages destined to the Driver DeMux. For simplicity of the figure, only one pointer was depicted, but actually there are two pointers, pointing to the first and last nodes of the queue, providing faster insert and removal node operations. The memory location of this CMD Queue structure is listed in the *CMD Queue Locator Board*. The location of a CMD Queue is posted in this "queue of queues" by the respective Command Reader, so other threads may find where the Command Writer will fetch its messages for processing.

Once this pair of threads is created, they become completely independent, so they must be aware of the command channel status. When the channel becomes unusable during their operation, the RCI specification defines that there is no alternative way to reach the addressee, so the associated CMD Queue and all of its messages are discarded, before both threads cancel their execution themselves.

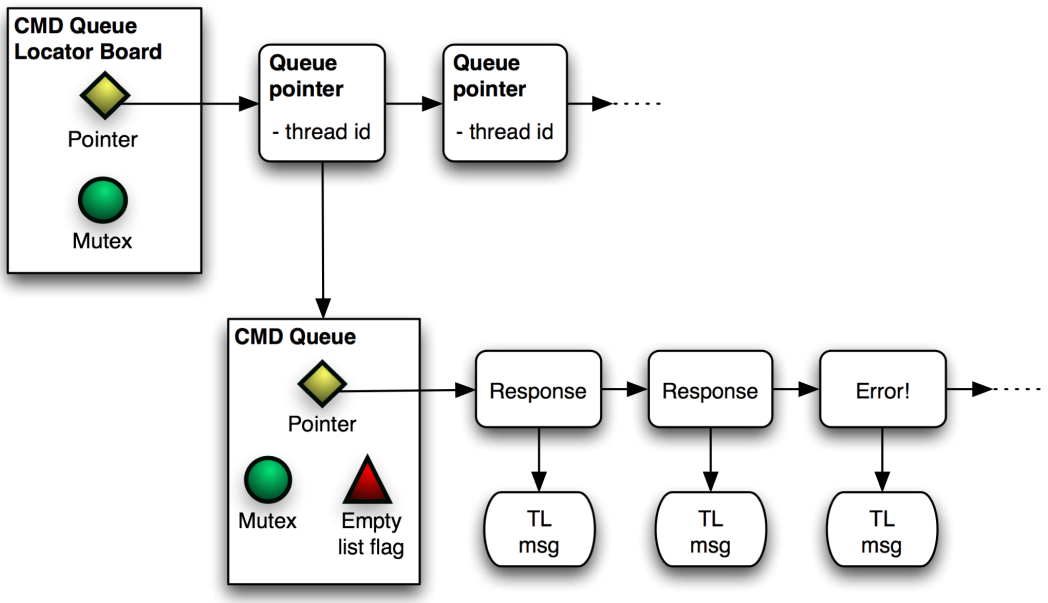

Figure 5.13: Mechanism to locate Command Queues.

5.5.2 Serving the Notification channel

According to the RCI specification, the Notification channel must be unique, so once the RCI Notification Listener accepts one connection with the Driver DeMux, it must assure that no other Notification channel will be established while the actual is still active. This requirement also means that the TL must be aware of the status of this connection: if the channel is lost by some reason, it must be closed and the RCI Notification Listener should be quickly ready to accept a new connection.

It is also important to note that data takes a unidirectional route across the Notification channel, from the TL to the Driver DeMux.

Being so, once the RCI Notification Listener accepts one connection from the Driver DeMux, it generates a new thread, the *Notification Handler* (depicted in Figure 5.12 as NTFY handler) which will be in charge of fetching the event notifications from the *Notification Queue* (depicted as NTFY queue) and send them to the Driver DeMux. During the Notification Handler operation, the RCI Notification Listener is blocked, aware of the status of the Notification channel and if it notices some abnormal situation, it will cancel the Notification Handler execution and will wait for a new connection request, to repeat the same process again.

The Notification Queue is a publicly known queue where event notifications are buffered. As seen in Figure 5.14, the Notification Queue is a linked list with a control structure. The control structure contains a mutex that prevents simultaneous insert/retrieval operations in the queue, and a conditional variable that keeps the Notification Handler blocked when queue is empty. Two pointers (only one is represented for figure simplicity) locate the first and last nodes of the queue for faster insert and retrieval operations. The queue nodes point to TL messages that contain the event notifications to be sent to the Driver DeMux.

This Notification Queue is independent from the Notification Handler, so if this thread for some reason is terminated, the queue is able to maintain the pending messages. Although the RCI specification allows the TL to discard the notifications that are pendent when the Notification channel is

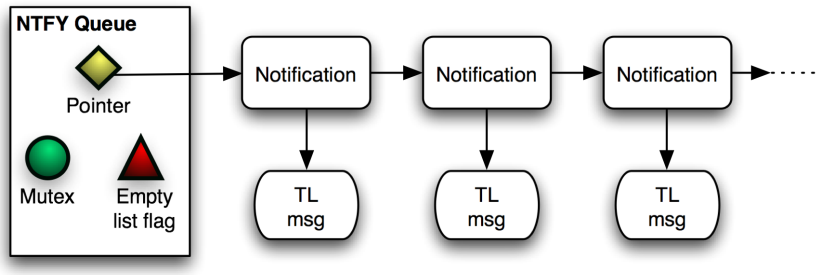

Figure 5.14: Perspective of the NTFY Queue.

down, it was decided that the messages would not be destroyed and, once the Notification channel is recovered, these messages will be processed by the new Notification Handler.

5.5.3 Distributing TL messages to channel handlers

Messages that are supposed to be delivered to the Driver DeMux are buffered in the RCI Queue. These messages can be notifications or responses to issued commands and have to be distributed to the right channel handlers. The *TL Messages Router* analyses both the message type and the thread id in each TL message, and takes one of the following actions:

thread id is ZERO. The message is intended to be processed by the Notification Handler. The message is simply inserted in the Notification Queue.

thread id is not ZERO. The message is addressed to one Command Handler. The TL Messages Router will search the Command Queue that is identified by the same thread id at the CMD Queue Locator Board. If the queue is found the message is inserted in the queue, but if the queue does not exist, it means that the channel is no longer active, and the message is discarded.

5.6 Processing communications with the NL

The REMPLI NL is implemented inside the kernel and offers an interface to its services in the form of two different Linux device drivers: the Master device and the Slave device, as already illustrated in Figure 4.1. Both Linux device drivers are only available to REMPLI Bridges, while Access Points only have the Master device available, and Nodes have one Slave device, just as depicted in Figure 4.3.

Data flows between the NL device drivers and the TL are unpredictable so, even if it is not complex for the TL to know when data must be sent to the NL, it must be simultaneously aware of possible data arriving from the lower layer. Profiting from the multithreaded environment, the module that processes communications with the NL is divided into 3 components, running in separate threads. Figure 5.15 illustrates in detail this module and interactions with other software components.

The *NL Writer* is the thread responsible to send data to the NL device. This component fetches messages from the NL queue that carry data destined to the lower layer. Based on the message type, the NL Writer decides which operation should be invoked from the NL device driver, in order to accomplish the task. The message types clearly indicate to which type of network device (Master/Slave) the message is destined, so this component performs unambiguously in REMPLI Bridges, where the message handler routine contains a segment for Master-destined messages – named as Master processor – and another for Slave-destined messages – named as Slave Processor. Obviously, the Master

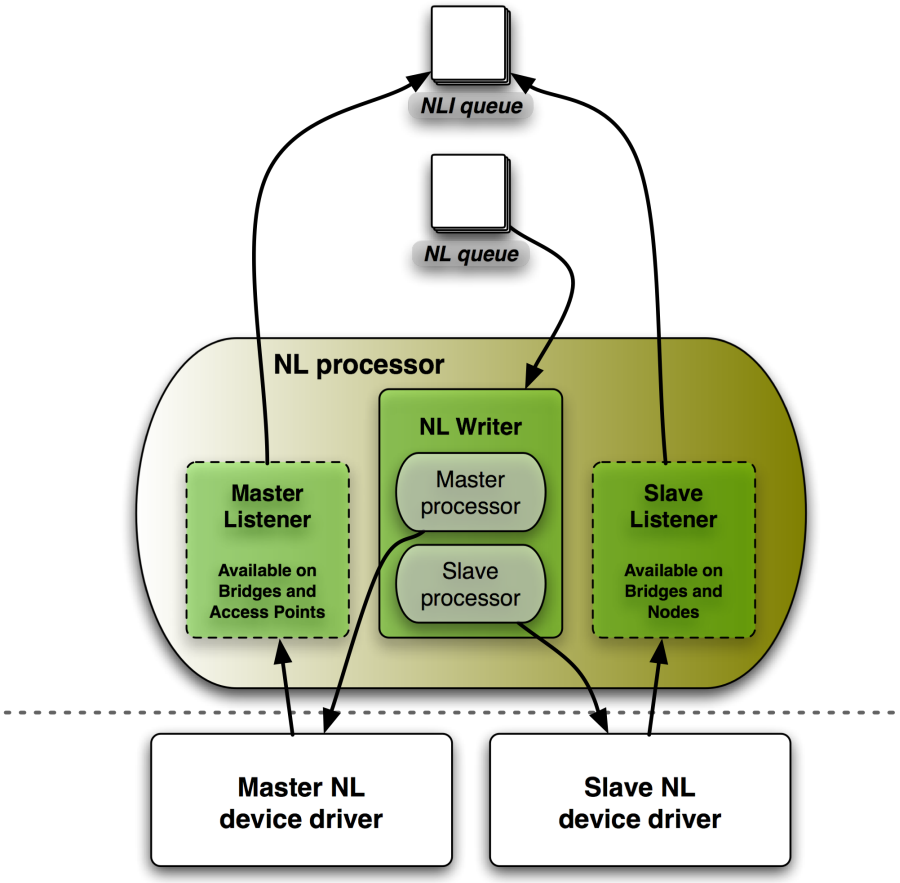

Figure 5.15: NL processor details.

processor is suppressed from the REMPLI Nodes and, conversely, the Slave processor is not present in REMPLI Access Points, as these REMPLI devices do not contain a Master device and a Slave device, respectively.

Along with each of the network devices available in the REMPLI device, there is a corresponding listener thread – *Master listener* and *Slave listener* – that will collect the data arriving from the network device and make it available for the TL modules. These listening components are normally waiting to be signalled by the network device, to start their operation. Once there is data available, the listener thread will generate a new TL message block and will fill its fields and set the message type according to the event related to the data received. The message is then inserted into the NLI queue, so the NLI may switch it to the appropriate module.

5.7 Heartbeat

In a REMPLI device, every software component must indicate to the Management Driver that it was launched and executing correctly. This means that every software component must check itself if it is functioning properly, before sending an indication to the Management Driver – the *Heartbeat*. When the Management Driver verifies that one registered component is not issuing heartbeats, it will kill the process and launch a new process of the same program. The Management Driver sets a 120 seconds time window for the TL to cyclically confirm that is still operating properly.

The REMPLI TL verifies the state of its core components – the TRM, the QM, the RCIM and the NLI – and also the NL processor, circulating a periodic message which only purpose is to find if the modules are processing messages from their queues. This periodic message is supported by the Timer Service (refer to section 5.4.4), which is in charge of placing one message at the head of the RCIM queue, every 30 seconds. When the RCIM receives this timed message, it generates a new message that will cross the multiple TL modules. If working properly, every module is supposed to forward the received message to the next module, inserting at the end of each queue just like any regular message. It is required that the message can cross all modules in less than the remaining 90 seconds, following this sequence:

1. RCIM forwards to QM;
2. QM forwards to TRM;
3. TRM forwards to NLI;
4. NLI forwards to NL processor;

Once the NL processor receives the message, it calls the heartbeat function call from the Management Driver, informing that the TL is in proper condition and saving the TL from an unnecessary process termination.

Chapter 6

Analysis and evaluation

After the development of the REMPLI Transport Layer and its integration in the communication stack, its timing behaviour was tested and the results analysed. The software was tested on actual REMPLI devices which constituted the physical REMPLI model testbed, located at the iAd labs in Großhabersdorf, Germany, but remotely accessible over Internet. Figure 6.1 is a diagram of the devices and connections that constituted the testbed. The testbed included two of each REMPLI devices, being all of them reachable by the remaining devices. Signal attenuators of 60 dB were inserted between each REMPLI AP and the nearest REMPLI Bridge to simulate long distance lines, and both REMPLI AP had a direct path with 180 dB attenuation. REMPLI Bridges and REMPLI Nodes were interconnected by a meshed segment without signal attenuators, and the connection quality was very similar in all possible connections. The boot up configuration of the REMPLI Bridges could be modified to load the REMPLI AP or REMPLI Node software instead, making the REMPLI Bridge act as a REMPLI AP (with the slave network device turned off) or a REMPLI Node (with the master network device turned off) respectively. This operation allowed to break the testbed network into two independent networks comprising exclusively REMPLI AP and Nodes, offering two parallel testing environments. The results of tests presented in this chapter were obtained using a subnetwork that included Bridge 1 (acting as a REMPLI AP) and Nodes 1 and 2. Although the number of REMPLI devices was quite restrict as compared with a practical field deployment, the testbed was a useful tool to analyse the performance of the hard and software components.

To verify the proper functioning of the REMPLI Transport Layer, a set of tests was defined to collect data for a timing behaviour analysis. The REMPLI TL was set to work over the board OS and it would use the functionality provided by the Network Layer and Management Driver. On top of the TL was executing a application that would simulate a DeMux Driver. This application was programable, allowing the definition of a set of actions that should be carried out, in order to observe how the TL and, consequently, the whole underlying system would behave.

The TL was set to log timestamps of certain events that permit to trace the sequence of operations and also measure the time each module uses to perform specific tasks. The times required to process a message and to insert a message in a queue are two significant items for analysis, since the operation of each module is message driven. Although inserting a message in a queue is not the main functionality of a module, it could be a bottlenecking factor. If after processing one message the module needs to forward it to another module, it will block until the message is placed in the respective queue before it fetches the following message to process. Times for any specific task are subject of variation due to several factors, being the most relevant:

1. the concurrent processing under a multithreaded environment in a uniprocessor (single core)

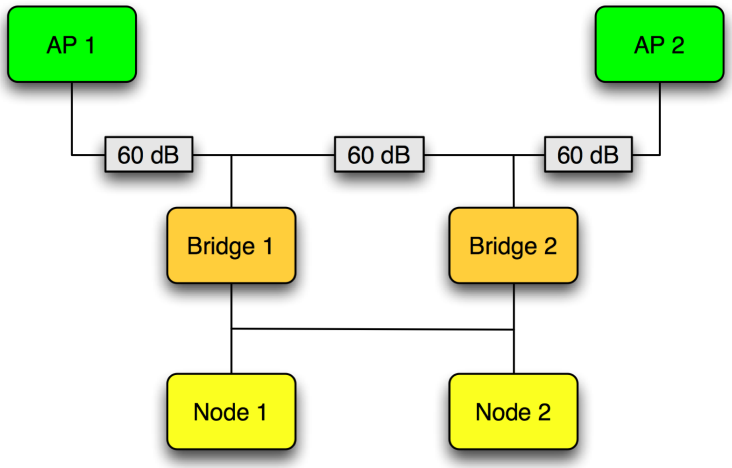

Figure 6.1: Diagram of the REMPLI testbed.

system, and

2. the diverse size of data to process, which is relevant in fragmentation/assembling and transmission procedures.

To verify the performance of the REMPLI TL, one test was planned in which one *REMPLI AP test application* would send a RESPONSE REQUEST containing payload data to a *REMPLI NODE test application* which, in turn, should reply the AP with the same payload data. The payload data is randomly generated and carries a Cyclic Redundancy Check (CRC), which allows the REMPLI AP test application to verify if the payload data was not corrupted during the request-response cycle. This sequence, which is illustrated in Figure 6.2, was then repeated several times in order to observe if the performance of the REMPLI TL would present high variance. Although the REMPLI TL allows concurrent processing of application commands, this characteristic was not used during tests: one command would only be issued after the emission of the previous command was considered concluded either by success or error, so there would be no interference between commands in the times recorded. Nevertheless, during the processing of an application command, the REMPLI TL could be simultaneously processing an incoming RESPONSE relative to a previously issued RESPONSE REQUEST.

Although the size of one network packet is not fixed and is set according to the communications medium characteristics, the size available to applications data is expected to be around 50 bytes on field conditions. The test applications allowed to program the size of the payload data, and two sizes were used in performance tests:

20 bytes. This size was short enough to fit in one network packet, so there was no need of data fragmentation in the QM.

2000 bytes. Payload of this size is too long, so the QM is always required to perform fragmentation to send application data on several network packets.

The use of fixed size of payload data on tests, instead of random sizes, allows to observe the variance of the TL performance under uniform circumstances, discarding the variance that could be injected by varying payload data size.

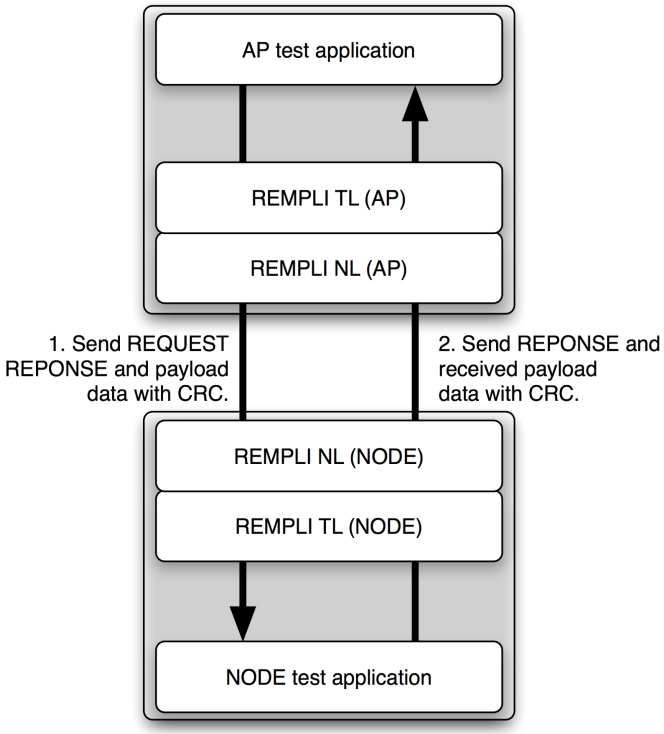

Figure 6.2: TL performance test scheme.

Three types of time periods were recorded, in order to inspect the TL performance:

Message processing time. The time taken by a TL module to process each message it received in its mailbox was measured. The measure of this time started once the module successfully retrieved a message from the message queue[1], and stopped once the processing of the message was considered concluded.

Message insertion time. The time one module took to insert one message in a message queue was measured. Since the modules insert messages in queues while they are processing one message, this time was naturally included in the *message processing time*.

TL response time. The time it took since the DeMux emulator application issued one command to the TL, and the instant the TL replied with an success/error response was measured.

6.1 Message queue operations

TL modules communicate by exchanging messages that are inserted in message queues associated to particular modules. In order to keep the consistency of message queues, operations on each queue are mutually exclusive, as already described in chapter 5 (section 5.4). An activity which access is mutually exclusive can become a bottleneck of the system if the time a module is blocked becomes regularly significant, so the considerations regarding short times to insert/retrieve messages as compared with the times required to process the same messages had to be verified (and confirm that queue

[1]The time a module was blocked by the mutex, waiting to retrieve the message from the queue was not measured.

Module	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\Delta t_{min}(\mu s)$	$\Delta t_{max}(\mu s)$	$\Delta t < 1000\mu s$	$\Delta t > 10000\mu s$
DMX proc	467.8	328.1	258	1078	90.48%	0.00%
RCIM	487.7	315.5	185	1054	97.01%	0.00%
QM	1564.1	1951.0	185	22361	56.59%	1.10%
TRM	274.3	168.4	184	1244	98.91%	0.00%
NLI	597.3	1190.4	185	15099	94.69%	0.41%
NL proc	255.6	58.0	184	917	100.00%	0.00%

Table 6.1: Insertion times for AP, 20 bytes SENDREQRESP.

Module	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\Delta t_{min}(\mu s)$	$\Delta t_{max}(\mu s)$	$\Delta t < 1000\mu s$	$\Delta t > 10000\mu s$
DMX proc	569.1	407.2	258	1203	66.67%	0.00%
RCIM	329.2	201.4	257	953	100%	0.00%
QM	1579.0	4272.7	184	117726	58.99%	0.52%
TRM	239.2	226.5	184	3730	98.98%	0.00%
NLI	538.2	1810.3	184	88015	93.88%	0.12%
NL proc	1629.9	23517.9	184	672749	88.84%	0.44%

Table 6.2: Insertion times for AP, 2000 bytes SENDREQRESP.

operations did not negatively influence the REMPLI TL performance). Nevertheless, the time required to insert one message in a queue is also dependent on the thread scheduling therefore, having these concepts in mind, the tests should reveal short times with some small acceptable variation to an average value.

During the test phase, the time required for a module to insert a message in a queue – from the instant it tries to obtain the mutex till it releases the same mutex – was recorded. With the collected data is possible to determine which module inserted the message, the time required, which queue was operated and the type of message.

The results of batch tests on a REMPLI AP are presented in Table 6.1 where the REMPLI AP test application injected SENDREQRESP[2] commands with 20 bytes of payload data, and Table 6.2 corresponds to the same procedure, but payload data of 2000 bytes size. On both tests, the respective REMPLI Node replied with the same payload data sent by the REMPLI AP, acting as a echo device.

From Tables 6.1 and 6.2 it is possible to infer that the minimum time required to insert a message in a queue is about 184 μs. This value should correspond to the case in which the thread obtains immediately the mutex and the critical section is performed without being preemptively suspended by the kernel, before the mutex is released.

On both types of tests, the average time to insert one message in a queue is lesser than 10 times the minimum time recorded, which means that waiting to obtain the mutex is common during operation. It can be seen in Figure 6.3 that the average times do not suffer a pronounced variation in accordance with the size of the payload data.

However, the standard deviation and, consequently, the variance is higher than expected, specially

[2]According to the RCI specification, the SENDREQRESP is a command in which the REMPLI AP expects one REMPLI NODE to reply, receiving the answer in the form of a Notification.

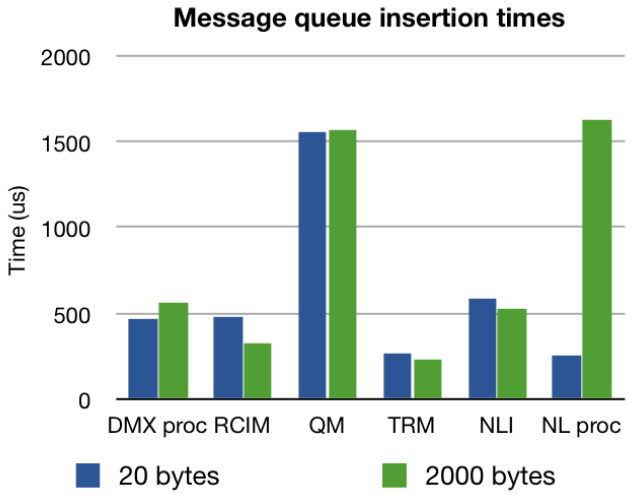

Figure 6.3: AP average message queue insertion times.

Module	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\Delta t_{min}(\mu s)$	$\Delta t_{max}(\mu s)$	$\Delta t < 1000\mu s$	$\Delta t > 10000\mu s$
DMX proc	940.8	587.4	258	2406	65.00%	0.00%
RCIM	601.0	1137.2	185	10419	93.98%	1.20%
QM	1141.1	1218.2	185	19376	69.35%	0.38%
TRM	267.3	126.8	184	917	100.00%	0.00%
NLI	424.0	298.1	185	1088	98.05%	0.00%
NL proc	389.1	351.1	184	1965	96.23%	0.00%

Table 6.3: Insertion times for Node, 20 bytes SENDRESP.

for the QM and NLI on both test runs, and on the NL in the larger payload data size. These high values of standard deviation are due to some spurious but seriously high values of time required to insert a message in the queue. It is possible to state that these values are spurious, because the recorded data shows that most of the insert operations take less than 1000 μs, which is about 5 times the minimum time, and only a very small residual insert operations take longer than 10000 μs, which is approximately 54 times the minimum time.

The reasons for such high variance, may be related with the scheduling mechanism of the threads, in which if one thread does not obtain immediately the mutex it will be suspended, and the cycle to wake this thread may sometime take long if other threads are using intensively the processor. This is the case in the test with 2000 bytes payload data size in which the NL experiences severely longer waiting times, and the QM has to assemble the data received from the network.

Regarding the corresponding times recorded for the REMPLI Nodes in the same test runs, the statistics are presented in Table 6.3 and Table 6.4. Again, the average times are under 10 times the minimum measured value – 184 μs – and the majority of the insert operations take less than 1000 μs and only a residual percentage of operations take more than 10000 μs. In Figure 6.4 it is possible to observe that, again, the average times do not correlate with the size of the payload data. The results are similar to the ones found in the REMPLI AP tests, and the same considerations apply.

Module	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\Delta t_{min}(\mu s)$	$\Delta t_{max}(\mu s)$	$\Delta t < 1000\mu s$	$\Delta t > 10000\mu s$
DMX proc	1356.0	3045.4	258	19951	55.00%	2.50%
RCIM	382.0	295.0	257	1493	92.22%	0.00%
QM	1109.7	1191.9	184	29671	68.20%	0.11%
TRM	310.5	725.3	184	10427	98.51%	0.46%
NLI	666.4	3878.9	184	112475	95.32%	0.12%
NL proc	337.6	243.1	184	3882	98.74%	0.00%

Table 6.4: Insertion times for Node, 2000 bytes SENDRESP.

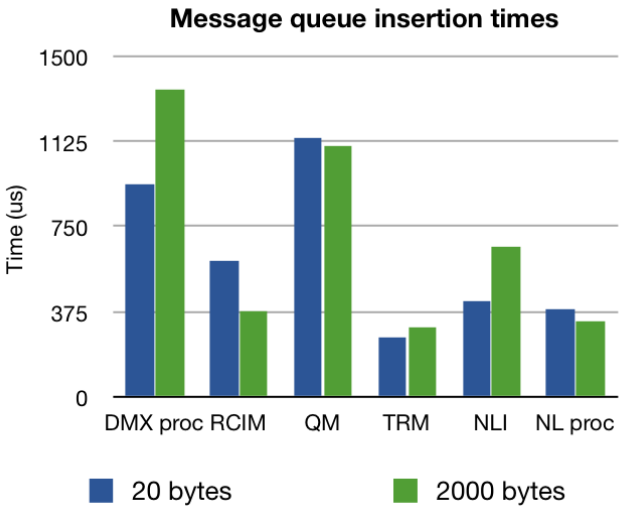

Figure 6.4: Node average message queue insertion times.

Module	$\overline{\Delta t}(\mu s)$		$\sigma_{\Delta t}(\mu s)$		$\Delta t_{min}(\mu s)$		$\Delta t_{max}(\mu s)$	
DMX proc	358.6	(358.6)	396.8	(396.8)	260	(260)	2858	(2858)
RCIM	879.8	(392.1)	344.4	(152.2)	543	(297)	1519	(1061)
QM	3570.0	(3021.4)	6073.1	(5696.9)	87	(87)	56000	(54943)
TRM	1013.9	(926.0)	4679.2	(4657.5)	41	(41)	65251	(64992)
NLI	1358.7	(762.7)	3377.9	(3192.2)	493	(295)	40642	(40382)
NL proc	987.8	(970.3)	4331.6	(4333.4)	189	(189)	36358	(36358)

Table 6.5: Processing times for AP, 20 bytes SENDREQRESP.

6.2 Message processing

Each module operates in an infinite cycle, in which the module has to retrieve a message from its message queue and consequently process it. There are several types of messages for each module and each type requires a specific procedure to be performed.

The times recorded during test runs include the operations that were performed since one message was successfully retrieved from the message queue until the processing was considered complete. Several procedures require that data is sent to another module (via a TL message) and, naturally, the processing time includes the time spent to insert a message in one other module message queue, since this operation is part of the message processing procedure.

Regarding the activity of the interfacing modules – *NL processor* and *DeMux processor* – the recorded message processing times do not include all the operations performed by these components. For the *DeMux processor*, the time this component took since it received a TL message from the RCIM and switch it to the *Notification handler* queue or the proper *Command Writer* queue was recorded; however, the time the Command and Notification handlers take to send the data over the proper socket channel was not recorded. Similarly, the time necessary to process an incoming command from the Driver DeMux was not also recorded because, once the Command Reader receives a command, it immediately tries to inform the RCIM through a TL message and this time was already considered in the message queue insertion times. The statistics on the *NL processor* message processing activity relates only to the operations in which data is sent to the network device: the *NL processor* receives TL messages asking to send data over the network, so this is the processing the NL processor is expected to do. This way, the activity in which the NL processor receives data from the network device and immediately forwards it to the NLI was not recorded, being these values already considered in the message queue insertion times.

Tables 6.5 and 6.6 present the average times spent by each module in a REMPLI Access Point to process one message. The values used in these statistics include the time taken by the module to insert a message in a queue, whenever the message processing procedure requires so. Although the operations on message queues are part of the message processing, the analysis of the processing times without the variance added by the random times introduced by the mutex locks – which, has already observed, could have an appreciable effect – was also interesting. To eliminate this source of variance in such cases, the insertion time was subtracted from the processing time, being the statistics calculated with these values presented inside parentheses, next to the total values.

From both tables, it is possible to observe that an increase in size of the payload data of an application does not affect the processing times of most modules, with the exception of the QM. In fact, only the QM presents a significant increase in the average time and, more specially, in the standard

Module	$\overline{\Delta t}(\mu s)$		$\sigma_{\Delta t}(\mu s)$		$\Delta t_{min}(\mu s)$		$\Delta t_{max}(\mu s)$	
DMX proc	293.6	(293.6)	34.3	(34.3)	259	(259)	331	(331)
RCIM	722.2	(393.0)	292.3	(133.8)	555	(297)	1692	(1182)
QM	6478.7	(5898.4)	61354.1	(61261.8)	87	(87)	872618	(871705)
TRM	576.3	(477.4)	1262.9	(1224.4)	41	(41)	43595	(43336)
NLI	998.4	(460.2)	2795.6	(2126.8)	482	(295)	88381	(68128)
NL proc	425.6	(413.9)	1218.1	(1212.4)	189	(189)	35911	(35911)

Table 6.6: Processing times for AP, 2000 bytes SENDREQRESP.

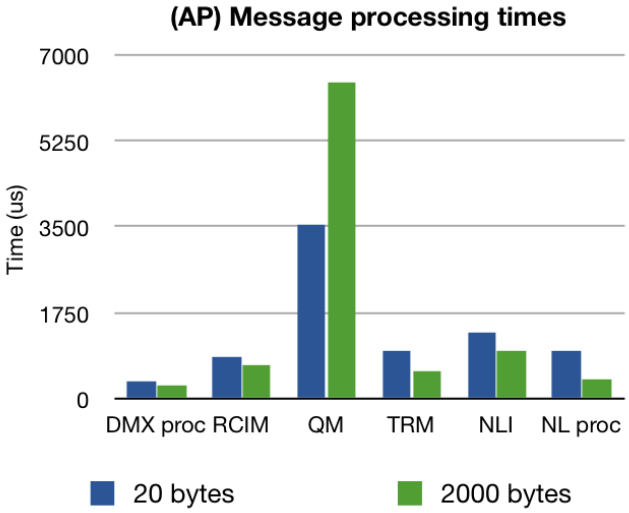

Figure 6.5: AP average message processing times.

deviation and maximum Δt value. This can be justified with the fact that most modules do not operate the payload data, except the QM which must fragment the data into smaller chunks to fit the network packet size, and also the opposite procedure. Even the network device write operations performed by the NL processor are independent from the size of the command payload data: the larger the data, the more write operations of small size packets are performed by the NL processor, but this does not change the average value.

Considering the size of payload data on both test runs, the proportionality of 100× is not reflected in the average processing time of the QM, which only increases 1.8×, as can be seen in Figure 6.5. Since fragmentation of payload is only one of several tasks performed by the QM, this smaller proportion could only be caused by the effect of other lesser time-consuming operations performed by this module in the calculation of the average value. The times required for the QM to process the messages which have to fragment the command payload data were analysed separately and the results are shown in Table 6.7. Obviously, the average times to fragment the command data are longer than the general average times for the QM to process one message. Interesting is to observe that the QM takes about 52.1× longer to generate a queue of data chunks for a 100× bigger data payload. Removing the variance due to mutex operations when message queue operations are performed, this proportionality raises to 55.8×.

	Complete procedure		Not considering queue insertion	
	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$
20 bytes payload	16055.5	2696.0	14960.4	2661.0
2000 bytes payload	836176.3	20753.1	835255.2	20757.6
Multiplication factor	52.1×		55.8×	

Table 6.7: Access Point QM times to fragment SENDREQRESP data.

Module	$\overline{\Delta t}(\mu s)$		$\sigma_{\Delta t}(\mu s)$		$\Delta t_{min}(\mu s)$		$\Delta t_{max}(\mu s)$	
DMX proc	499.4	(499.4)	704.5	(704.5)	259	(259)	3011	(3011)
RCIM	1060.7	(459.7)	1262.8	(343.6)	541	(293)	11639	(2176)
QM	1896.7	(1485.1)	2481.7	(2152.1)	87	(87)	23260	(22344)
TRM	531.8	(455.9)	410.3	(290.3)	41	(41)	1641	(1282)
NLI	781.5	(356.4)	361.8	(204.2)	483	(297)	2806	(2547)
NL proc	1063.3	(1063.3)	4261.7	(4261.7)	190	(190)	29501	(29501)

Table 6.8: Processing times for Node, 20 bytes SENDRESP.

Observing the Node statistics, presented in Tables 6.8 and 6.9, we can observe that the size of the payload data has practically effect on the average values of time required to process a message. There is an increase in maximum times observed in the QM, TRM and NLI modules, which may be the cause for the increase in the the standard deviation of the late two modules.

On the QM, an increase of the average processing time values was expected, as fragmentation and assembling operations were being performed by this module. But the increase of the size of payload data by 100× only produced an increase of 1.1× in the average time, as visually illustrated in Figure 6.6. Even taking exclusively in account the times relating to payload fragmentation procedure, as presented in Table 6.10, there is only an increase in the proportion of 2.2× in the average time, when the proportion of the size of payload data is 100×. It is also very interesting to observe that the average time required by the AP to process a command with 20 bytes of payload data (3570.0 μs) is longer than the time taken by the Node for the counterpart operation with 2000 bytes of payload data (2064.5 μs). While on the QM of the AP the longer times recorded consistently correspond to command

Module	$\overline{\Delta t}(\mu s)$		$\sigma_{\Delta t}(\mu s)$		$\Delta t_{min}(\mu s)$		$\Delta t_{max}(\mu s)$	
DMX proc	418.6	(418.6)	561.9	(8561.9)	259	(259)	3712	(3712)
RCIM	839.9	(457.9)	431.2	(290.9)	553	(293)	2058	(1800)
QM	2064.5	(1533.2)	2949.7	(2720.5)	87	(87)	68772	(67855)
TRM	721.2	(591.6)	2754.6	(2658.9)	41	(41)	65485	(65299)
NLI	1152.1	(485.6)	4392.9	(2064.5)	482	(296)	112840	(39994)
NL proc	435.4	(435.4)	1242.9	(1242.9)	189	(189)	33690	(33690)

Table 6.9: Processing times for Node, 2000 bytes SENDRESP.

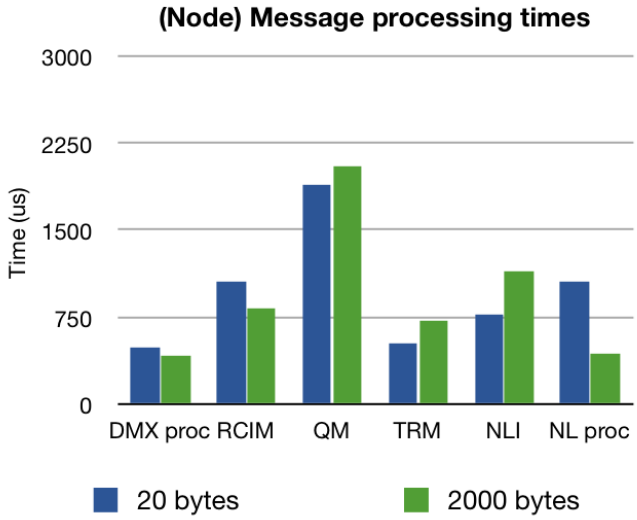

Figure 6.6: Node average message processing times.

	With queue insertion		Without queue insertion	
	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$	$\overline{\Delta t}(\mu s)$	$\sigma_{\Delta t}(\mu s)$
20 bytes payload	5796.9	191.3	4877.0	142.8
2000 bytes payload	12489.9	278.7	11559.3	270.1
Multiplication factor	2.2×		2.4×	

Table 6.10: Node QM times to fragment SENDRESP data.

payload fragmentation of data of same size, on the QM of the Node the longer times correspond to message processing that sporadically takes longer, probably due to thread scheduling. This indicates that processing an application command on an REMPLI AP is heavier than on an REMPLI Node.

6.3 TL performance

The previous sections have presented indicators about the performance of the major TL modules, but it is also important to observe the performance of the REMPLI TL, as an integrated system. The REMPLI TL offers services to the layer above, the Drivers DeMux, which issues commands and expects a response informing if the command was successful or not. In the tests that were carried out, the application that emulated the Drivers DeMux recorded the timestamps when it issued a command and when it received the respective response from the REMPLI TL. Table 6.11 presents the average results and the standard deviation for commands issued with 20 bytes and 2000 bytes of payload data.

From the low standard deviation values, it is possible to infer that the REMPLI TL presents a regular behaviour under similar requests, on both types of devices.

The REMPLI AP and the REMPLI Node present very similar response times when the data to be sent through the network has a small size. However, the REMPLI Node responds faster when the size of data increases. In fact, an increase of 100× on the size of payload data leads to an increase of the

Size of payload data	$\overline{\Delta t}(s)$	$\sigma_{\Delta t}(s)$	$\Delta t_{min}(s)$	$\Delta t_{max}(s)$	Goodput (bps)
REMPLI AP					
20 bytes	1.096044	0.100964	0.909153	1.231499	145.98
2000 bytes	12.005622	1.094176	7.520724	12.701801	1332.71
REMPLI NODE					
20 bytes	0.909914	0.107126	0.759636	1.114349	175.84
2000 bytes	6.085396	0.287488	5.718992	6.574816	2629.24

Table 6.11: TL response times on a REMPLI AP.

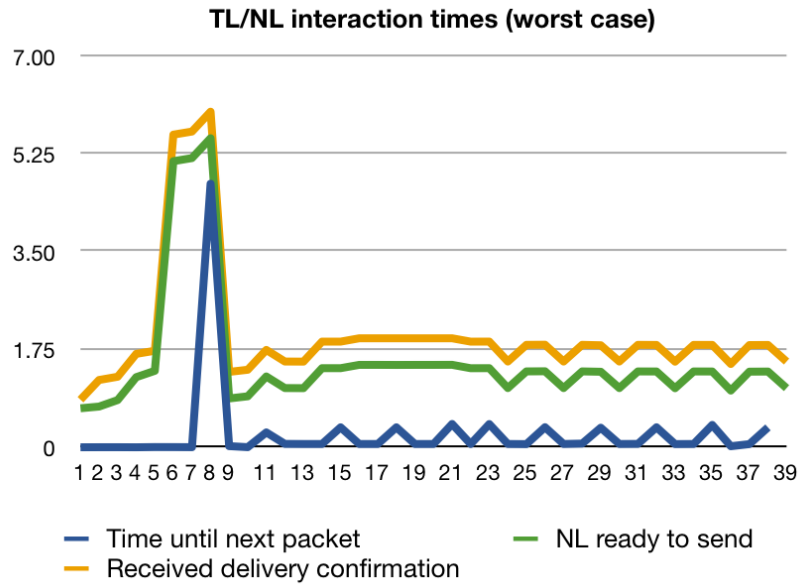

Figure 6.7: NL packet time responsiveness (worst case recorded).

average response time in 6.7× on the REMPLI Node and 11.0× on the REMPLI AP.

To verify if the Transport Layer could cope with the network capacity, two practical cases – the worst and the best cases recorded – were studied, in which the Access Point tries to send a SENDREQRESP with 2000 bytes of payload. For this study the selection relied on the Access Point because its transport layer presented longer response times than the Node. In the worst case, the REMPLI TL took 12.701801 seconds to accept the command and payload data, and reply with a success response. From the time records of the activity between the TL and the NL we can observe the facts listed below.

- The TL took 9.454430 s to write the 39 network packets that carried the command and data to the NL. The average time between write operations was 0.248503 s.
- The NL took 10.514470 s since it received the first packet until it informed the TL that was about to send the last packet to the network. The average time between reading a packet from the TL until get it ready to send it to the network was 1.547704 s.

- The time elapsed since the first packet was written to the NL until the TL received the confirmation that the last packet was received by the Node was 10.990460 s.

Figure 6.7 graphically presents the times recorded for the referred case. It is possible to observe that for every packet, the time the TL takes to perform the next consecutive write operation to the NL[3] is continuously inferior than the time the packet is buffered in the NL before being ready to be inserted in the network[4]. During the 4.72 seconds between packets number 8 and number 9, the NL device was busy receiving 36 packets carrying part of a Node response to the immediately previous issued AP request. Once the NL load decreased, the TL resumed writing packets with short time intervals. This behaviour indicates that the REMPLI TL is not inserting prejudicial delays and has a margin to cope with the network performance.

From the same graph it is also interesting to observe that the time between the NL being ready to insert the packet in the network and the confirmation of the packet reception by the Node is practically constant – being the average value 0.461382 s – which indicates the predictable behaviour of the network due to the stringent NL time constraints.

During the best case recorded, the NL did not received any data packets from the Node. The REMPLI TL took 7.520724 seconds to accept the command and payload data, and reply with a success response. An analysis of the time records of the activity between the TL and the NL revealed the facts listed below.

- The TL took 4.577190 s to write the 39 network packets that carried the command and data to the NL. The average time between write operations was 0.120443 s.
- The NL took 5.878630 s since it received the first packet until it informed the TL that was about to send the last packet to the network. The average time between reading a packet from the TL until get it ready to send it to the network was 1.082942 s.
- The time elapsed since the first packet was written to the NL until the TL received the confirmation that the last packet was received by the Node was 6.352190 s.

Figure 6.8 graphically presents the times recorded for the best recorded case. In this graph, it is visible that the TL does not delay the NL operation, even when the NL is under light load.

The same graph shows once more that the time between the NL being ready to insert the packet in the network and the confirmation of the packet reception is practically constant, being the average value 0.476046 s.

[3]Represented by the series caption 'Time until next packet'.
[4]Represented by the series caption 'NL ready to send'.

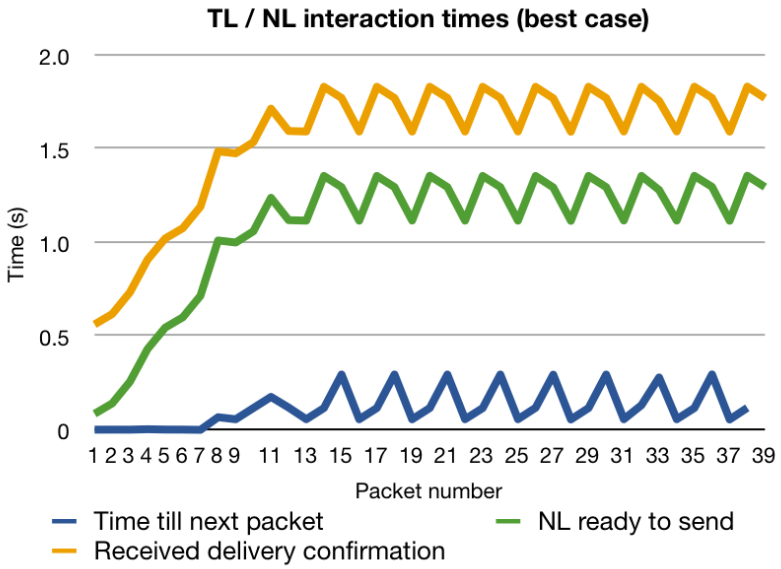

Figure 6.8: NL packet time responsiveness (best case recorded).

Chapter 7

Conclusions and future work

7.1 Summary of this dissertation

The use of power-lines as a transmission medium is currently standardised in Europe by the CENELEC EN 50065 standard [44]. The CENELEC A-band, ranging from 3 kHz to 95 kHz, is restricted to electricity suppliers and their licensees and is actually subject of research and development of applications for the utility companies. The liberalisation of the energy markets and the availability of multiple electricity producers and energy sources is pushing the need of real-time knowledge and control of the power network with a finer detail than was common some years ago. Embedded systems can be installed in key locations to perform metering and SCADA, as long as they can be remotely accessed by application servers through some communications medium. The power distribution infrastructure is already deployed and reaches all electricity clients which – for the electricity distributor companies – constitutes an economical alternative to other communications media such as dedicated telephone lines or wireless links.

The REMPLI project was thus motivated by the current market needs and opportunities. The principal aim of this project is to develop a communications system that can transparently exchange data between digital equipment (meters and SCADA Remote Terminal Units) that follows industrial bus open standards and is already available in the market, through the electric power distribution grid. Additional or upcoming protocols can be supported by the REMPLI system requiring only the development of an application driver to realise the adaptation to the PLC medium.

The REMPLI system is composed by three types of devices: Access Points, Bridges and Nodes. Access Points provide Application Servers the access to the REMPLI system, performing the translation between the IP-based networks – where the Application Servers are connected – and the PLC network where REMPLI packets are commuted. Bridges are employed to exchange data between the medium-voltage and the low-voltage segments at transformer stations. Since transmission characteristics are certainly different in the referred distribution segments, different modulation parameters can be set to optimise the overall throughput of the REMPLI network, resulting in different maximum packet sizes. In this way, Bridges may have to assemble and refragment data that has to cross the boundary between the medium- and low voltage segments. Nodes interfaces the meters and/or SCADA Remote Terminal Units with the REMPLI network.

The REMPLI communication protocol follows the master/slave model, in which the Access Points are the masters and Bridges and Nodes are the slaves. During regular operation, Access Points are solicited by the Application Servers to issue requests to one or multiple slave devices which may have to respond back or not. Generally, masters take the initiative of communication, but slaves have the

possibility to issue alarm messages to inform a master about some relevant event.

The possibility of multiple masters and slaves in a real-time meshed network and the level of throughput poses new challenges to efficient routing and scheduling algorithms. The complexity inherent in the REMPLI Transport Layer requirements was an important factor in the decision of implementing the REMPLI Transport Layer as a user-space application. The REMPLI Transport Layer is composed of four major modules: the Transport Route Manager, the Queue Manager, the REMPLI Communications Interface Manager and the Network Layer Manager. The Transport Route Manager is in charge of collecting data about the network status and decide which is the best route for each packet and when is the right moment to deliver a packet to the network. The Queue Manager is responsible to fragment the outgoing data in order to fit the network packets, and to reassemble the data received in the incoming network packets. The REMPLI Communication Interface Manager and the Network Layer Manager have to switch messages between the adjacent protocol layers and the Queue and Transport Route managers.

An embedded platform was specifically developed for this project. The hardware platform relies on the Hyperstone's hyNet 32XS microprocessor and iAd's DLC-2C/CA PLC chipset. The system kernel of this hardware is a uClinux port to the hyNet 32XS, a derivative of the Linux kernel for devices without Memory Management Unit. A toolchain was created to build software (either kernel modules or user-space applications), including the GNU C-Compiler, the GNU Debugger and GNU binutils. The uClibc was ported to the hyNet 32XS, so compiled applications can link it and make use of this source of object code.

The four modules of the REMPLI Transport Layer were developed and tested in a discrete event simulation system and the object-oriented source code served as basis for the implementation of this protocol layer in the referred embedded platform. According to the specification, the REMPLI Transport Layer should have its modules working concurrently and so this application became a multi-threaded process, being each module executed as a single thread. A message queue service, based in linked lists, allows each module to issue operation requests and associated data to other modules; the message queue operations constitute a variation of the classic producer/consumer problem and the exclusive access to queue operations is assured by an associated mutex. Two additional modules were created to operate directly with the adjacent layers: the Driver DeMux processor and the Network Layer Processor. The Driver DeMux processor listens for connection requests issued by the Driver DeMux and takes care of communications performed with the upper layer. Communications with the Driver DeMux occur in two types of channels: the Notification Channel, a unique unidirectional socket where the Transport Layer sends event notifications to the Driver DeMux, and the Command Channels, from where the Transport Layer receives application commands and sends the command status (success/error) response. The Network Layer processor has to handle packets that are received from the network and also prepare data to be sent over the network adapter. Access Points and Nodes have one single network adapter so the operation of the Network Layer processor is quite straightforward, Bridges have two network adapters and the NL processor must select, based on the data properties, to which adapter each packet must be sent.

The REMPLI Transport Layer was tested in laboratory, to verify its reliability and responsiveness. The complexity of the algorithms and the operations controlled by mutexes could pose a serious risk to the responsiveness of this layer, furthermore when the network offers a narrow bandwidth and latency generated by the protocol stack is not desirable. The tests were performed in a testbed with a reduced number of REMPLI devices, and above the REMPLI Transport Layer there was an application that emulated the Driver DeMux, issuing sets of pre-programmed commands, and the times required to perform several operations were recorded for statistical analysis.

7.2 Main contributions

7.2.1 Characterisation of the REMPLI Transport Layer architecture

This document provides a detailed characterisation of the REMPLI Transport Layer architecture. According to the required functionality, the REMPLI TL integrates new packet routing and scheduling algorithms that were validated in a discrete event simulation system and applied in the implementation of the REMPLI Transport Layer.

7.2.2 Implementation of a prototype of the REMPLI Transport Layer

During the project timeline, the REMPLI Transport Layer was implemented for the embedded platform. The implementation resulted in three editions of the Transport Layer, one for each type of device (Access Point, Bridge and Node). These editions were tested in laboratory and then on a field test in Sofia, Bulgaria. The Access Point and Node editions functioned properly in both scenarios, showing a full integration in the protocol stack. During the field tests at Sofia, Bridges were set to act as repeaters. This solution was possible because the network was operating with one single Access Point and all segments were set to carry packets of same maximum size. In this scenario, the use of repeaters resulted in increased payload data throughput compared with the use of bridges, but if the network settings became more complex (multiple Access Points and different modulation parameters in the medium- and low-voltage segments) a degradation of performance was expected.

In the field test of Sofia, multiple remote meter readings were flawlessly performed and SCADA Remote Terminal Units executed remotely issued commands [63]. The timing analysis results are consonant with the settings of the network.

7.2.3 Efficiency and predictability analysis of the REMPLI TL

A set of tests was performed in laboratory, in order to verify three aspects of the REMPLI TL behaviour:

1. The amount of time each module is blocked by a mutex, when the module has to insert a message in a message queue.

2. The amount of time each module takes to process one received message.

3. The overall responsiveness of the TL, seen by its client – the Driver DeMux.

The mechanism of modules intercommunication based on messages queues is effective and scalable, but there was the concern that the indeterministic access to queue operations controlled by mutexes could insert delays that compromise the overall TL performance. The results showed that the minimum value for each operation was less than 200 μs and most of the blocking periods were less than 1000 μs, independently of the size of the payload data carried by the message, which denotes a predictable behaviour. Nevertheless, when one module was using the processor more intensively, the blocking periods could sporadically grow up to tens or (more rarely) hundreds of milliseconds, due to thread scheduling strategy which benefits the most active thread.

The statistics related to the time taken by a module to process one message revealed that the size of the payload data only affects significantly the Queue Manager. This is justified by two reasons:

1. During the tests, the times taken when data is exchanged between the DeMux processor and the Driver DeMux were not recorded. The elapsed time during socket read and write operations obviously vary with the size of data exchanged.

2. The Queue Manager is the only that operates on payload data, performing data fragmentation and assembling operations.

The statistics also revealed that the message queue insert operations (that involve mutex locks) has an effect in the average times and variance of the processing times, which is obvious since the time to acquire a mutex is not deterministic. Nevertheless, the impact is not harmful even when (which very sporadically) the period the thread is blocked is in the order of hundreds of milliseconds.

Finally, the overall performance analysis of the REMPLI TL revealed a regular behaviour under equivalent solicitations. This analysis also demonstrated that the REMPLI TL is not degrading the REMPLI system and performs faster than the network capacity. Also, the implementation of the REMPLI TL as an user space application had no bottleneck effect in the system.

7.3 Future work

This dissertation covers the implementation of a fully functional prototype of the REMPLI Transport Layer for the embedded platform also developed within the REMPLI project. In the field test at Sofia, the Bridges acted as repeaters. This alternative solution is functional, but to take advantage of the REMPLI system design, the Bridge software integration should be validated.

The three types REMPLI devices share the same processor board, but in a REMPLI system, the Nodes outnumber the Access Points and Bridges together. The functionality of the REMPLI Node is less complex than the Bridges and Access Points, therefore there is the intention to develop a new REMPLI Node based on the 8051 microcontroller architecture to lower the costs of installation of a REMPLI system. This includes the adaptation of the Node TL to this architecture.

Also an interesting area of future work, would be the analysis of different scheduling approaches, in order to reduce the variability in the behaviour (in some small cases) of the REMPLI TL.

Bibliography

[1] M. Lobashov, G. Pratl, and T. Sauter. Implications of power-line communications on distributed data acquisition and control system. In *EFTA'03 Emerging Technologies and Factory Automation*, pages 607–613, Lisbon, Portugal, September 2003.

[2] A. Treytl, T. Sauter, and G. Bumiller. Real-time energy management over power-lines and internet. In *8th International Symposium on Power-Line Communications and Its Applications (ISPLC'04)*, Zaragoza, Spain, March 2004.

[3] REMPLI web site [online]. Available from: `http://www.rempli.org` [cited 14th March 2008].

[4] A. Treytl, M. Lobashov, YQ Song, , G. Bumiller, and F. Pacheco. REMPLI Deliverable 1.1 - Applications Requirements Report. Technical report, The REMPLI Consortium, 2003.

[5] G. Bumiller, Liping Lu, and YeQiong Song. Analytic performance comparison of routing protocols in master-slave PLC networks. *International Symposium on Power Line Communications and Its Applications, 2005 (ISPLC'05)*, pages 116–120, April 2005.

[6] F. Pacheco, M. Lobashov, M. Pinho, and G. Pratl. A power line communication stack for metering, SCADA and large-scale domotic applications. In *9th International Symposium on Power-Line Communications and Its Applications (ISPLC'05)*, 2005.

[7] Leslie Pack Kaelbling. *Learning in Embedded Systems*. MIT Press, 1993.

[8] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

[9] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems. *Real-Time Systems*, 2(4):247–254, November 1990.

[10] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT 2004)*, pages 203–210, Pisa, Italy, September 2004. ACM.

[11] B. Graaf, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE SOFTWARE*, 20(6):61–69, Nov.-Dec. 2003.

[12] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research And Development*. Springer, 2005.

[13] A. Sangiovanni-Vincentelli and G. Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE DESIGN & TEST OF COMPUTERS*, 18(6):23–33, Nov/Dec 2001.

[14] S. Gumbrich. Embedded systems overhaul: It's time to tune up for the future of the automotive industry. *IBM Institute for Business Value*, December 2004. Available from: `http://www-935.ibm.com/services/us/imc/pdf/g510-3987-embedded-systems-overhaul.pdf` [cited 15th August 2008].

[15] AUTOSAR - AUTomotive Open System ARchitecture [online]. Available from: `http://www.autosar.org` [cited 18th August 2008].

[16] OSEK - Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug [online]. Available from: `http://www.osek-vdx.org` [cited 18th August 2008].

[17] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.

[18] J. Hyysalo, P. Parviainen, and M. Tihinen. Collaborative Embedded Systems Development: Survey of State of the Practice. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 130–138, Potsdam, Germany, March 2006. IEEE Computer Society.

[19] Device details – Nokia N82 [online]. Available from: `http://www.forum.nokia.com/devices/N82` [cited 15th August 2008].

[20] High performance - OMAP 2420 [online]. Available from: `http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=11990&contentId=4671` [cited 18th August 2008].

[21] Klaus Grimm. Software technology in an automotive company - major challenges. In *25th International Conference on Software Engineering (ICSE'03)*, pages 498–503, Portland, USA, May 2003. IEEE Computer Society.

[22] *MPC561/MPC563 Product Brief*. Freescale Semiconductor, Inc., 2003. Available from: `http://www.freescale.com/files/microcontrollers/doc/prod_brief/MPC561PB.pdf?fpsp=1&WT_TYPE=Data%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation` [cited 18th August 2008].

[23] *M58BW016DB datasheet*. ST Microelectronics, 2002. Available from: `http://www.datasheetcatalog.org/datasheet/stmicroelectronics/7822.pdf` [cited 18th August 2008].

[24] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996.*, volume 1, pages 293–307, Aspen, USA, Feb 1996.

[25] *REMPLI System Overview - White Paper*. REMPLI Consortium, 2004.

[26] *IEC TC 57, IEC 60870-5-x Telecontrol equipment and systems - Part 5: Transmission protocols*. IEC, 1990.

[27] G. Pratl, M. Lobashov, and T. Sauter. Highly modular gateway architecture for fieldbus/internet connections. In *Feldbustagung FeT'01*, page 293, Nancy, France, November 2001.

[28] T. Sauter, M. Lobashov, and G. Pratl. Lessons learnt from internet access to fieldbus gateways. In *IECON'02 The 28th Annual Conference of the IEEE Industrial Electronics Society*, Sevilla, Spain, November 2002.

[29] G. Pratl and M. Lobashov. Remote access to power-line networked nodes: Digging the tunnel. In *IEEE International Workshop on Factory Communication Systems (WFCS2004)*, pages 323–332, Vienna, Austria, 2004.

[30] A. Bratoukhine and G. Pratl. REMPLI Deliverable 6.3 - Fieldbus data representation model. Technical report, The REMPLI Consortium, 2004.

[31] *ISO/IEC 7498-1 Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. ISO, 2nd edition, 1994.

[32] M. Lobashov, F. Pacheco, and L. Marques. REMPLI Deliverable 1.4 - Gateway Specification. Technical report, The REMPLI Consortium, 2004.

[33] P. Maas, G. Bumiller, M. Lobashov, and G. Pratl. REMPLI Deliverable 1.3 - Node Software Specification. Technical report, The REMPLI Consortium, 2003.

[34] G. Bumiller, M. Sebeck, E. Böhme, M. Deinzer, F. Pacheco, and R. Brito. REMPLI Deliverable 1.2 - PLC System Specification. Technical report, The REMPLI Consortium, 2003.

[35] G. Bumiller and M. Sebeck. Complete power-line narrow band system for urban-wide communication. In *Symposium of Power-Line Communication and its Applications*, Malmö, Sweden, 2001.

[36] A. Treytl, P. Maas, A. van Gordon, and K. Knebel. REMPLI Deliverable 4.5 - PLC Node Application Software. Technical report, The REMPLI Consortium, 2005.

[37] *hyNet 32XS Network Processor (brochure)*. Hyperstone AG, 2007. Available from: `http://www.iad-de.com/products/asics/hynet/hyNet_32XS_Flyer.pdf` [cited 27th July 2007].

[38] *ISO 11898 Road vehicles - Controller area network (CAN)*. ISO, 2003.

[39] *ITU-T I.430 Integrated Services Digital Network (ISDN) - Basic user-network interface - Layer 1 specification*. ITU, November 1995.

[40] R. Finlayson. *RFC 906: Bootstrap Loading using TFTP*. IETF, June 1984.

[41] *RFC 1094: NFS: Network File System Protocol Specification*. IETF, March 1989.

[42] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813: NFS Version 3 Protocol Specification*. IETF, June 1995.

[43] S. Shepler, B. Callaghan, and et al. *RFC 3530: Network File System (NFS) version 4 Protocol*. IETF, April 2003.

[44] *EN 50065 Signalling on low-voltage electrical installations in the frequency range 3 kHz to 148,5 kHz*. CENELEC, 2001.

[45] *IEC 61000-3-x Electromagnetic compatibility (EMC) - Part 3: Limits*. IEC, 3rd edition, November 2005.

[46] uClinux – Embedded Linux/Microcontroller Project [online]. Available from: `http://www.uclinux.org` [cited 14th March 2008].

[47] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, February 2005.

[48] uClibc – A C library for Embedded Linux [online]. Available from: `http://www.uclibc.org` [cited 14th March 2008].

[49] *hyNet 32XS Linux-Tools (brochure)*. iAd GmbH, 2007. Available from: `http://www.iad-de.com/products/asics/hynet/hyNet_32XS_Flyer.pdf` [cited 14th March 2008].

[50] A. Barros, F. Pacheco, and L.M. Pinho. A complex protocol layer as a linux user-space process. In *International Symposium on Industrial Embedded Systems, 2006 (IES '06)*, Juan-les-Pins, France, Oct. 2006. IEEE.

[51] M. Lobashov. REMPLI Deliverable 6.1 - Gateway Core Implementation Report. Technical report, The REMPLI Consortium, 2005.

[52] R. Droms. *RFC 2131: Dynamic Host Configuration Protocol*. IETF, March 1997.

[53] OMNeT++ Discrete Event Simulation System [online]. Available from: `http://www.omnetpp.org` [cited 14th March 2008].

[54] András Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001.

[55] András Varga. Parametrized topologies for simulation programs. In *Proceedings of the Western Multiconference on Simulation (WMC'98) / Communication Networks and Distributed Systems (CNDS'98)*, San Diego, USA, January 1998.

[56] Luís Marques, Filipe Pacheco, and Luís Miguel Pinho. Characterizing the timing behaviour of power-line communication by means of simulation. Technical report, IPP-HURRAY! Research Group, June 2004.

[57] G. Bumiller. Power-line physical layer emulator for protocol development. In *Symposium of Power-Line Communication and its Applications*, Zaragoza, Spain, March 2004.

[58] Luis Miguel Marques and Filipe Pacheco. REMPLI discrete event simulation system. Technical report, IPP-HURRAY! Research Group, September 2007. Available from: `http://www.hurray.isep.ipp.pt/asp/download_doc2a.asp?id=401` [cited 14th September 2008].

[59] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software*. Linux Documentation Project, 3rd edition, March 2003. Available from: `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html` [cited 21st May 2008].

[60] Donald E. Knuth. *Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 3rd edition, July 1997.

[61] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[62] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, December 2001.

[63] A. Treytl, P. Maas, G. Kosel, M. Lobashov, and M. Sebeck. REMPLI final report. Technical report, The REMPLI consortium, April 2008. Available from: `http://www.iad-de.com/technology/rempli/download/REMPLI_final_report.pdf` [cited 14th September 2008].