ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

School of Engineering

Master's Degree in Computer Engineering

Design and Implementation of a Middleware for Efficient Publish/Subscribe in Embedded Systems

Thesis in

MOBILE SYSTEMS M

Advisor
Chiar.mo Prof. Paolo Bellavista

Presented by
Thomas Ambrogini

Academic Year 2022/2023

Dedication

Dedico questo spazio della tesi per esprimere la mia gratitudine a coloro che mi hanno sostenuto durante la sua realizzazione e nel mio percorso accademico e di vita.

Innanzitutto, estendo il mio sincero ringraziamento al mio relatore, il Prof. Bellavista, e al mio tutor aziendale, Simone Cilli, per le conoscenze trasmesse durante l'intero percorso, i loro preziosi consigli e la loro costante disponibilità. Sono profondamente grato ai miei genitori e a mia sorella, che mi hanno sostenuto in ogni aspetto della mia vita accademica e personale, e al resto della mia famiglia per il continuo supporto e affetto.

Un riconoscimento molto importante va ai miei amici, senza i quali non avrei mai raggiunto questo traguardo. Un grazie speciale a tutte le persone che ho conosciuto e mi hanno supportato durante la parte finale del mio percorso universitario. Esprimo infine la mia più profonda gratitudine a Datasensing, per avermi offerto la mia prima vera esperienza nel mondo del lavoro in un ambito per la quale nutro una grande passione.

Abstract

This master thesis explores the development and implementation of a Pub/Sub (Publish/Subscribe) platform customized for embedded devices. The initial phase of the project involves a comprehensive analysis of the chosen embedded board, going into its specifications and capabilities. The central focus lies in facilitating communication between heterogeneous processors, aiming to establish seamless connectivity and data exchange. The thesis also investigates the integration of real-time cores alongside Linux cores, balancing the need for responsive performance with the versatility of a Linux-based system. Through experimental evaluations, the thesis assesses the platform's performance in different scenarios.

Contents

List of Tables

List of Figures

Introduction

In the domain of industrial automation, Datasensing emerges as a key player, specializing in the development, manufacturing, and supply of advanced solutions in Machine Vision, Sensors, and Safety systems. Their extensive product line includes smart cameras, industrial cameras, industrial vision processors, software, and accessories for Machine Vision, as well as a range of sensors and safety features like light curtains, laser scanners, and control units. Focused on industrial automation, Datasensing offers services in different sectors, including automotive, electronics, food and beverage, pharmaceuticals, general manufacturing, and intralogistics.

This thesis centers around the development and implementation of a Pub/Sub (Publish/Subscribe) communication platform specifically designed for embedded systems. The primary goal is to design a Pub/Sub middleware that addresses the unique requirements of embedded systems in the automation industry, providing a robust and efficient platform for facilitating seamless information exchange among heterogeneous processors.

To achieve the goal of the project, the thesis is structured to provide a comprehensive exploration of embedded systems, focusing on the analysis of specific characteristics of the chosen embedded board and a detailed study of the platform Apache Kafka as a solution for publish/subscribe (Pub/Sub) communication. The last chapters of the thesis goes into the design and implementation of a Pub/Sub platform customized for embedded systems, incorporating insights gained from the analysis of both the embedded board and Apache Kafka. The conclusive chapter presents the experimental results, providing an evaluation of the proposed Pub/Sub platform's performance and effectiveness within the domain of embedded systems.

Chapter 1

Embedded Systems

Designing and writing software for embedded systems poses a different set of challenges than traditional high-level software development. This chapter gives an overview of these challenges.

Embedded systems are computing devices performing specific, dedicated tasks with no direct or continued user interaction. [21] Due to the variety of markets and technologies, these objects have different shapes and sizes, in general, they are small and resource constrained.

One of the characteristics for which embedded systems are used is the capacity of delivering real time processing. Real time processing refers to external events within precise and predictable timeframes. In manufacturing, robotics, and automative production, timely responses to changing conditions can mean the difference between operational success and failure. Embedded systems excel at this, ensuring that processes are executed with minimal latency, guaranteeing the coordination and synchronization of various components.

Embedded systems' ability to deliver real time processing capabilities is fundamental in automation industries. However, the importance of real time processing extends beyond the automation industry. In the healthcare sector, embedded systems can provide the delivery of life-saving treatments.

Most embedded systems are resource constrained, limiting memory and processing power. These constraints are a direct result of the need for compact, energy-efficient and cost-effective solutions. These limitations impose signif-

icant design challenges, forcing developers to optimize their code and data storage in order to respect those limits.

1.1 Microcontoller, Microprocessor and SoC

Embedded systems can rely on different integrated circuits (ICs) for executing applications, ranging from simpler, self-contained architectures to more advanced ones, which requires the integration of other components in order to work.

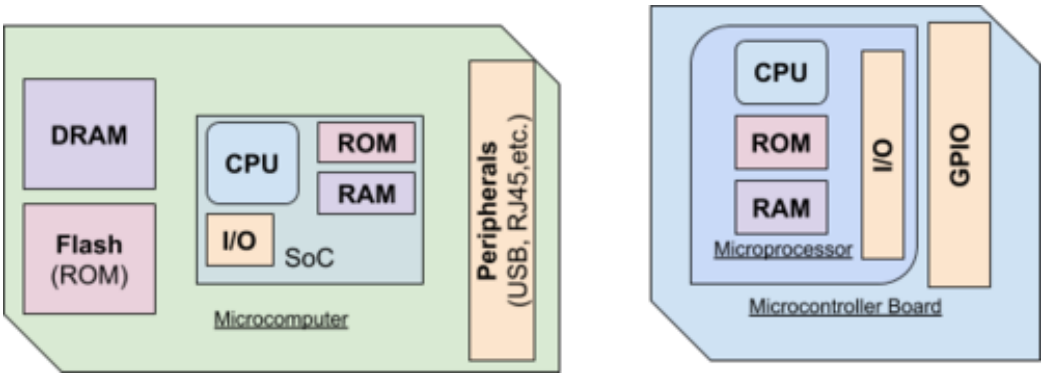

Figure 1.1: Microprocessor and microcontroller architectures

1.1.1 Microcontroller

Microcontrollers are compact, self-contained computing devices that find use in embedded systems. They are designed to perform specific tasks with great efficiency. A typical microcontroller incorporates a central processing unit (CPU), memory, input/output pins, and a range of peripherals within a single chip. This integration minimizes the need for external components, reducing cost, size, and power consumption. Microcontrollers excel in real-time control, making them suitable for applications where precise timing and responsiveness are crucial. Their versatility, energy efficiency, and cost-effectiveness have made them the preferred choice in an array of embedded systems, including robotics, automotive control systems, consumer electronics, and the Internet of Things (IoT).

1.1.2 Microprocessor

Microprocessors are the computational brains of modern computing systems. These general-purpose integrated circuits are designed to process data and execute instructions efficiently. Unlike microcontrollers, microprocessors do not integrate a wide array of peripherals, but they offer greater versatility in terms of the software and hardware components they can work with.

Microprocessors are commonly found in devices like personal computers, smartphones, and laptops, where their high processing power is a requirement. They excel in applications that demand data-intensive tasks, multitasking, and complex computation.

1.1.3 SoC

System-on-Chip (SoC) integrates multiple components into a single chip, just like microcontrollers. The main difference between SoCs and microcontrollers is that the latter can integrate also GPUs and various acceleraors, making them more complex and expensive. The integration of these elements into a single chip enhances efficiency and reduces power consumption, making SoC really valuable in the embedded systems world, especially for the automation industy, which could require substantial processing power for certain applications.

They provide an heterogeneous architecture with different processing units into the single IC (CPU, GPU, DSP) which can have great performances in multitask and multithread applications.

1.2 Toolchain

Developers rely on a set of tools to compile, link, execute and debug software to a specific target. Building the firmware images for an embedded system relies on a similar set of tools, called a toolchain.

In order to create an executable for an embedded system, the process is more complex than just creating the executable for a regular development environment. Since the architecture of the CPU is not the same of the host machine,

a compiler that generates machine code for the specific target architecture is needed. The process of creating executable for another architecture is called cross-compilation and the compiler needs some additional information to perform it.

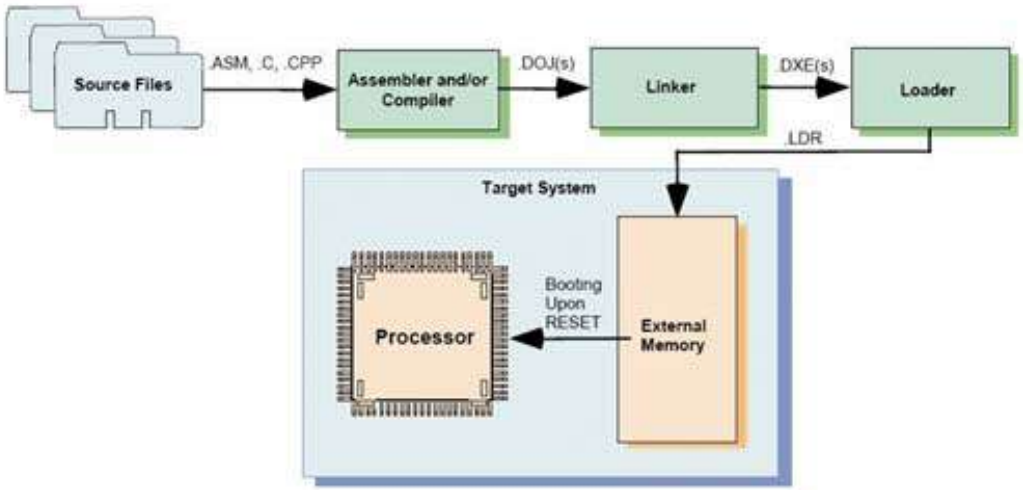

Figure 1.2: Cross Compilation process

1.2.1 Cross Compiler

The first element of the toolchain is the compiler, which is responsible for translating source code into machine code, which can be interpreted by a specific CPU. Each compiler can produce code for only one environment, since the source code is translated to machine specific instructions, which use the registers and the address model of a specific architecture.

The compiler creates object files (.o) starting from the source code. These files contain compiled code for the target machine, which is still not executable, but can be linked to other object files to create the executable.

Object files contains instructions for the CPU and a symbol table, containing information about functions and variables of the program. These files have information about the functions and variables initial value.

1.2.2 Linker

The linker performs the creation of the executable file. It aggregates all the object files received as input created by the compiler and resolves all the meaning for every symbol and all the dependencies, finally producing the executable.

The standard executable format is called ELF (executable and linkable format) and it has been designed to represent programs on disk and other media. It can be executed by loading the information in RAM in specific addresses. ELF files are divided into sections, each one of them represents a different area of memory with information needed for the execution of the program. They also contain an header with a pointer to the different sections inside the file. The main sections are:

- .text: instructions of the program.
- .rodata: read only variables.
- .data: variables accessible in read and write.
- .bss: uninitialized variables which can be accessed in read or write mode.

Read-only information can be loaded directly from flash in an embedded system, while data that need to be modified has to be in RAM on modifiable areas of memory.

In regular development environment, much of the complexity of the linking step is hidden, but the linker is configured, by default, to rearrange the compiled symbols into sections which can be loaded into the process' virtual addresses by the operating system.

In an embedded system, the linking process becomes more complex as compiled symbols have to be placed in physical addresses specific to the system's architecture. In order to specify in which areas of memory the sections have to be placed, a custom linker file has to be created. This file defines the memory layout of the executable bare metal application. Additionally, in linker files, custom sections can be used if they are required by the target system.

1.2.3 Make

GNU Make is a tool which controls the generation of executable and other non-source files of a program from the program's source files. [12] Make gets its knowledge on how to build binary images from a file called the makefile, which lists how to create each of the non-source files.

The makefile operates on rules and targets. Rules are a series of commands which instruct make on how to produce a target, representing a non-source file.

One of the advantages of using a build automation tool is that some of the targets could have implicit dependencies from other intermediate components, that are automatically resolved at compile time. It also reduce the time required for the compilation by compiling targets only when an update to the dependencies has been done.

1.2.4 Debugger

One of the most powerful tools within a toolchain is the debugger. Debuggers allows developer to test the runtime functionalities of an application, ensuring it operates as intended or locating the source of errors.

In the host environment, debugging an application consists of launching the debugger tool with an ELF file as argument or attaching it to a running application. The standard tool for debugging is called GDB.

However, in embedded development, the situation is slightly different since the application runs on a different machine than the one used for debugging. To address this, a version of GDB to debug remote platforms has been developed. The remote debug session requires an intermediate component which can translate the GDB commands to CPU specific instructions.

The debug of remote embedded platforms often require a peripheral like JTAG.

Figure 1.3: Steps needed for the system boot

1.3 Boot

A bootloader is a piece of software that starts as soon as you power on the system. Its primary functionality is to initiate subsequent software components such as: an operating system, a bare metal application or in some cases another bootloader. In the embedded world, bootloaders are intimely tied to the underlying architecture of the SoC. They are typically securely stored in a non-volatile, protected on-chip memory, ensuring their availability at startup.

Upon execution, the bootloader executes a series of tasks. It begins by performing hardware checks to verify the integrity and functionality of the SoC's components. Then, it takes charge of initializing the processor and configuring essential system-on-chip registers.

Bootloaders are also important in ensuring security. They often serve as the starting point for establishing a Hardware Root-of-trust, which forms the foundation for securing the entire system.

1.3.1 Boot steps

There are several steps that has to be followed for the boot process in an embedded system.

1. Preinitialization: before the actual boot process begins, certain conditions must be met. Power, clock signals, and control connections need to be established. Additionaly, boot configuration pins should be set to their desired logical levels to configure the system's behavior.

2. Power, Clock, and Reset Ramp Sequence: the power-management chip typically govern the applicaton of a specific sequence for powering up the system. This sequence ensures that the system components are

brought ot life in the correct order and that power and clock sources stabilize gradually.

3. ROM Code: responsible for finding, for downloading, and for executing the initial software (SBL).

4. Initial Software (Secondary Bootloader - SBL): software that loads, prepares, and passes control to application software or the high-level operating system (HLOS).

5. High-Level Operating System or Bare-Metal Application: the final stage of the boot process involves the execution of the high-level operating system or a bare-metal application. This is the primary software that runs on the main processor of the system. In case of an operating system, it provides a platform for running applications. In a bare-metal system, it typically includes the main application code that performs specific tasks or functions.

The initial steps of the device initialization process focus on hardware setup, but they involve configuring system interface pins with software-configurable functionality. This configuration is an essential part of the chip configuration and is application dependent.

1.3.2 Multi-stage Bootloader

In advanced SoCs, it is not uncommon for the bootloader to load another bootloader, creating a multi-stage bootloading process. In this case the bootloaders serve different purposes.

The initial bootloader, called the first stage bootloader, is typically stored in a secure, read-only memory for enhanced security and to maintain a consistent, known state each time the system powers on. The first stage bootloader is intentionally kept simple and performs only the essential configuration tasks required to bring up the system.

The secondary bootloader, or second stage bootloader, takes on a more complex and configurable role to serve the specific needs of the application. The second stage bootloader can be updated more easily and frequently compared to the first stage bootloader, making it more adaptable to changes and enhancements as the system evolves or new features are added.

The multi-stage approach to bootloading exploit the first stage for security and configuration, while the second one actually initializes the system, providing modularity and flexibility for future updates.

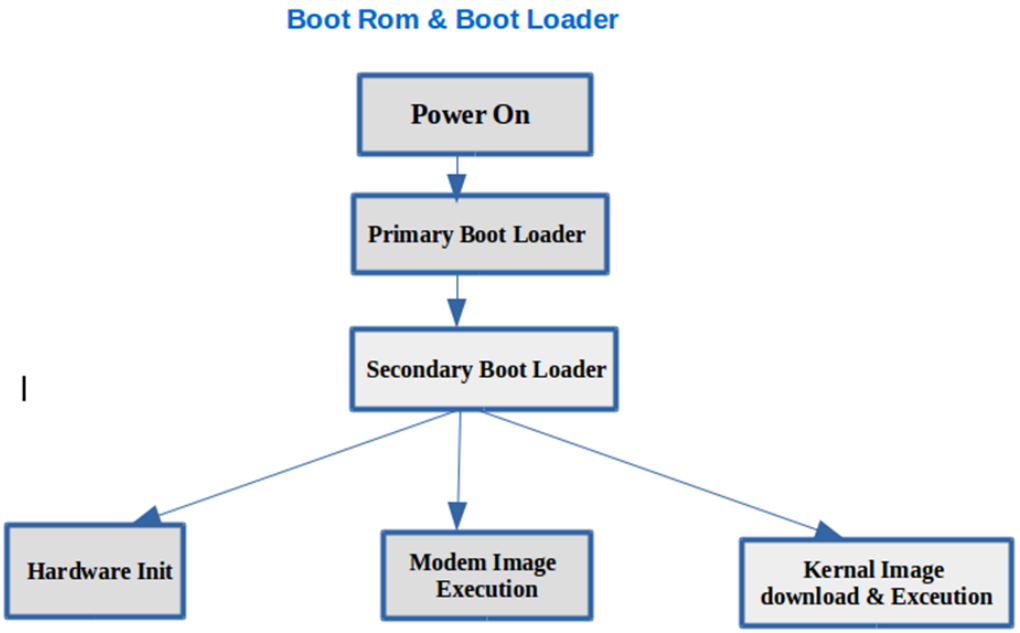

Figure 1.4: Multi stage bootloader

1.3.3 Multi core bootloading

Modern embedded systems have multi core architectures that must be initialized, and the process of bootloading must be carried out by multiple cores, working together.

Multi core bootloading can be seen as an extension of a multi-stage bootloader. In this scenario, the first stage bootloader may not even be aware of the presence of other cores. Its primary responsibility is to initiate the next-stage bootloader, wihch then takes charge of orchestrating the complex bootloading process across multiple cores.

The complexities which are introduced in this process are:

1. Image preparation: each processor core may have its own code and data

segments. The images for each core need to be prepared in a specific format that accounts for their individual requirements.

2. Image format: the images for every core could be concatenated into a single image for the second stage bootloader to load. The SBL must be aware of the format used, if only one image is used, to correctly parse it.

3. Coordination: coordinating the bootloading process across multiple cores involves setting up the execution context for each core, initializing hardware components, and synchronizing the start of execution. The SBL takes on the role of managing these tasks to ensure a smooth and reliable multi-core startup.

1.4 Operating Systems

In the embedded world there are different kind of operating systems, not only realized by different companies, but also with different needs as objective. Embedded devices, usually, needs to satisfy real time requirements, which makes hard using a general operating system, since they can not provide this feature.

The essence of real time is not that a computer has to respond fast, but that it has to respond reliably fast. The requirement of real time programming is being able to quickly handle aynchronous events. [18]

For many years, the only solution used for these devices was the bare metal approach. In this case there is not a real operating system which takes care of the management of fundamental operations. A bare metal application is considered to run directly on the hardware without respecting an external programming interface (e.g., the one given by an operating system), having direct access to CPU or microcontroller registers and without the security mechanisms of an OS.

Bare metal programming has the advantage of providing to the developers the highest grade of freedom, understanding exactly what every action will end up doing. Bare metal applications has the greatest possible degree of determinism and the resouce consumption is optimized for the specific case. On the other hand, it becomes hard to manage large project with different tasks and multiple operations to perform.

Other than the bare metal approach, there is the possibily of running a Real Time Operating System, which provides support for multiple tasks and device drivers between the hardware and the application, stack for the network and security.

1.4.1 Real Time Operating Systems

As the complexity of tasks continue to increase for embedded devices the necessity for a RTOS has increased and always more embedded devices go towards this solution, leaving behind the bare metal approach. These systems gives an efficient solution to meet hard real time requirements, particularly in safety critical applications that requires the management of different safety applications on the same device.

The scheduler in a RTOS, which has the job of deciding which thread have to execute on the core in any moment, is made in such a way to guarantee deterministic execution pattern. Real Time schedulers achieve determinism by assigning a priority to each task. In this way the scheduler can always run the task with the highest priority. [11]

FreeRTOS

FreeRTOS is a real-time operating system kernel for embedded devices. It is designed to be small and simple, built with an emphasis on reliability and ease of use. It is ideally suited for real-time applications, including a mix of both soft and hard real time requirements.

FreeRTOS allows applications to be organized as a collection of independent threads of execution. On a single core processor, only one thread can execute in any given time. The kernel decides which thread should be executing by examining the priority assigned to each threda by the application designer. Typically, threads with hard real-time requirements have higher priorities, ensuring their execution ahead of threads with soft real-time requirements.

In a single core system, the CPU must divide the tasks into time slices so that they can appear to run concurrently. The scheduler in an operating system is charged with figuring out which task to run each time slice. In FreeRTOS, the default time slice is 1 millisecond, referred to as "tick". A hardware

timer is configured to create an interrupt every 1 ms and the handler for that interrupt runs the scheduler, which chooses the task to run next. At eahch tick interrupt, the task with the highest priority is chosen to run. If the highest priority tasks have the same priority, they are executed in a round-robin fashion. If a task with an higher priority than the currently running task becomes available, then it will immediately run, withoug waiting for the next tick.

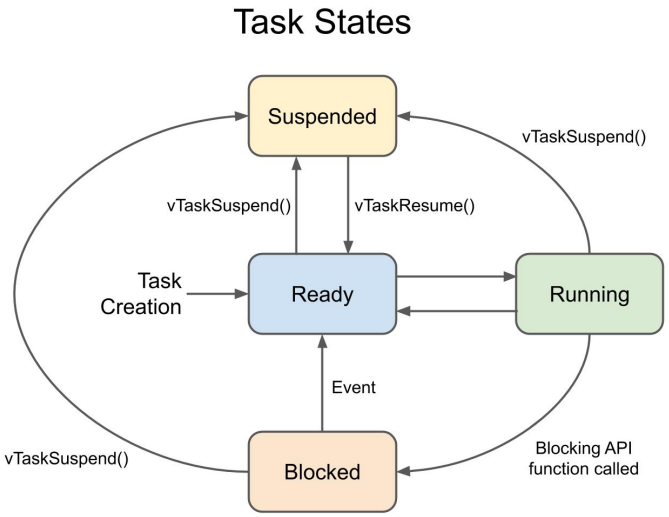

Figure 1.5: Task states

As soon as a task is created, it enters the Ready state, signaling their readiness to the scheduler for potential execution.

In multi core systems, the scheduler can schedule tasks based on the number of available cores in the system.

In addition, FreeRTOS provides features to simplify the embedded software development. It includes services for task scheduling, synchronization primitives, resource management, and interrupt handling.

1.4.2 Linux Embedded Systems

Embedded Linux is built on the same Linux kernel of every other linux systems. Other than the Linux kernel, embedded applications need additional

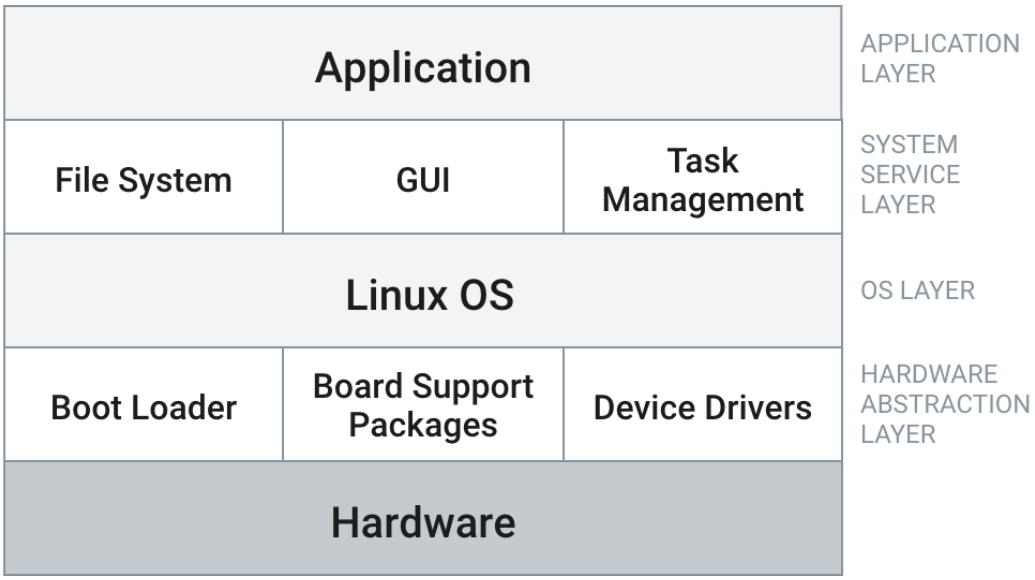

Figure 1.6: Embedded Linux Architecture

packages, which depends on the distribution and can be chosen based on the necessities of the developer. [5]

Although most of the devices are small in size, there are some of them that can run a version of the GNU/Linux operating system. To have this possibility the device must of a reasonable amount of RAM and the hardware must support MMU, Memory Management Unit, to provide different virtual address spaces for every process.

The main advantage of using a system with Linux is that the opportunity for a tailored soluion will be less, since the requirements, resource wise, for running Linux makes a system overkill for most of the applications. In this way, the software design can be simpler and it is possible to use existing solutions to embedded problems.

On the other hand, embedded devices have in many cases hard real time requirements, meaning that a series of operations must be accomplished in a short, measurable, predictable amount of time. To accomplish this Real-Time Operating Systems are used and it is almost impossible to achieve real time processing with embedded Linux as operating system, even if some patches to the kernel's scheduler have been applied to help meet these requirements.

Other application domains for embedded devices are low power consump-

tions, which could run on a battery or energy harvesting device. In this case, having an operating system increase the energy need of the device.

A reason for which linux could be a perfect candidate as an operating system for embedded devices is its modularity. Embedded developers have the possibility to customize their linux distribution, including only the necessary parts for their use case. [3]

Linux Embedded Real Time

One way to improve real time performances in an operating system is to add extra preemption points, where the OS can switch to critical operations. However, this process worsen the overall performances of the operating system in the general case, which is what general systems are optimized for, not taking into consideration the worst-case scenario, making the system non-deterministic.

The solution to the problem is to decouple the real time part of the system from the general purpose kernel. It is possible to optimize the real time OS to meet deadlines, having the best performance of general computing. [1]

Different projects to integrate real time into linux have been realized (e.g., RTLinux, RTAI). Although, they are maintained by different people, most of the functionalities are the same between different projects:

- A small real time core.
- One shot and periodic timer support.
- Real time scheduler.
- Real time threads.
- Real time FIFOs and shared memory.
- Real time interrupt handler.

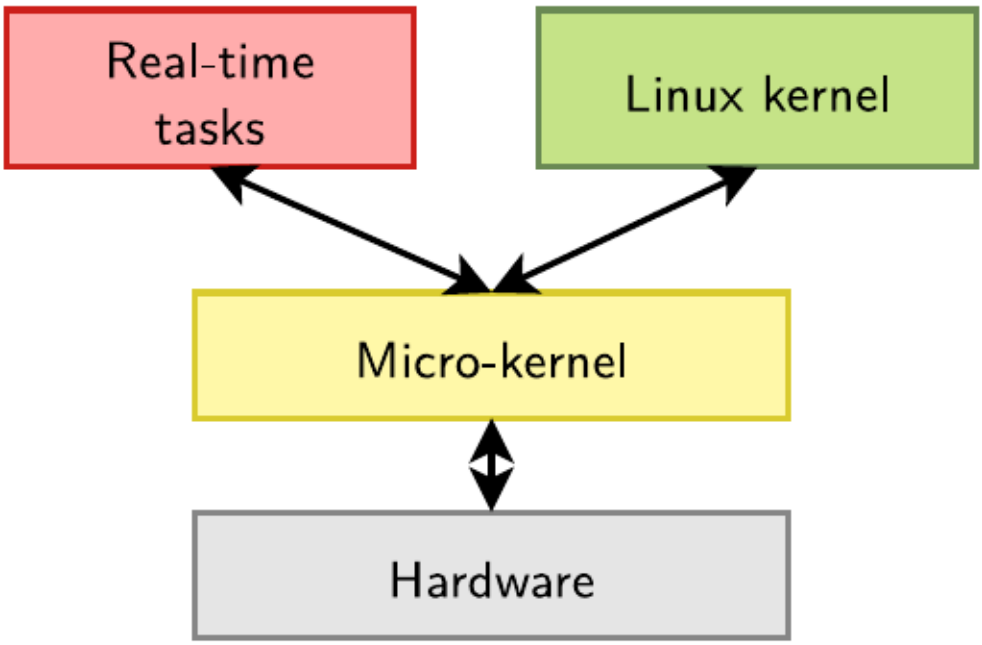

Figure 1.7: Architecture of Real Time Linux

Yocto

The Yocto Project is an open source project that helps developer create a custom Linux based systems independently from the hardware architecture. [16] Yocto is used to create operating system containing only the features needed by the embedded application. The Yocto project is more than a build system. It provides tools, processes, templates and methods so developers can rapidly create and deploy products for the embedded market.

Yocto makes possible to automize the creation of an embedded operating system. The needed steps to develop a system are:

1. Select and download the necessary packages and components.
2. Configure the downloaded packages.
3. Compile the configured packages.
4. Install the generated binary, libraries, etc.

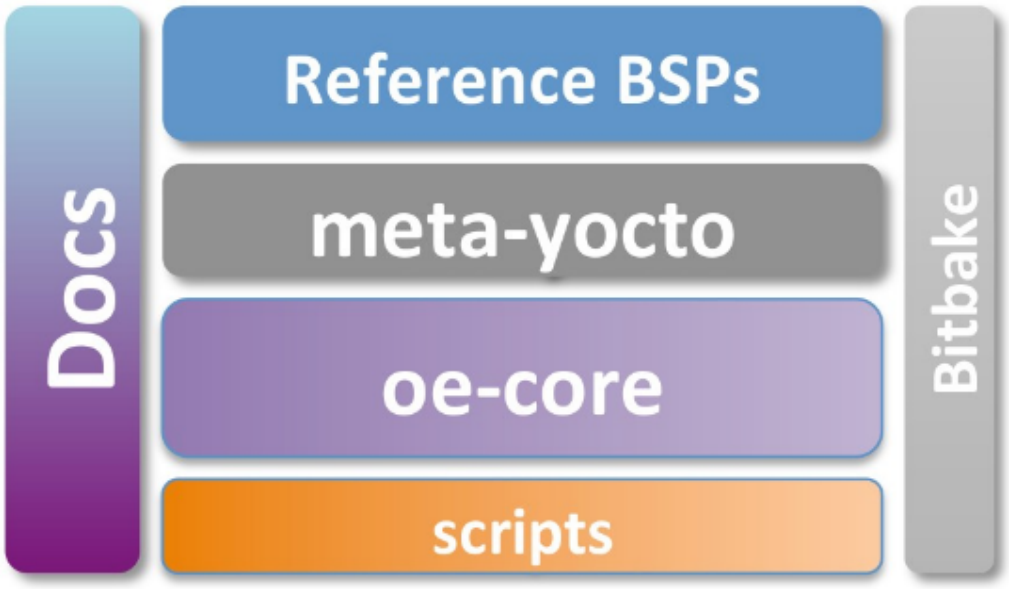

Figure 1.8: Yocto Stack

5. Generate the final deployable format.

Depending on the number of packages needed, these steps could become really complex to manage, making preferable to use an automated system.

Yocto is characterized by a layer model which grants modularity and the possibility for different developers to work on different part of the system at the same time.

Layers are a set of instructions that tells the build system what to do and they are used to logically separate information in a specific build.

Yocto is formed by two main components: bitbake and OE-Core (Open Embedded core). They are combined to realize the poky host build.

The OpenEmbedded-Core is a collection of information, such as configuration files and recipes used as a common layer to create the custom distributions. It is the starting point from which every distribution is built.

Bitbake is a build engine, which analyze files called recipes, and build an image from them. Inside the recipes there will be all the instruction the engine has to perform to create the image.

The project's strenght lies in its ability to offer customization, cross-compilation support, reporducibiliy and a community ecosystem, making it a fundamental tool for crafting optimized Linux-based solutions for embedded applications.

Poky

Poky is the Yocto project reference distribution and is composed of a collection of tools and metadata. It contains the OpenEmbedded Build system: BitBake and OpenEmbedded Core. It also include a large set of recipes, organized in hierarchical layers. It provices the mechanism to build and combine thousands of distributed open source projects to form a fully customizable, complete and coherent Linux software stack. [23]

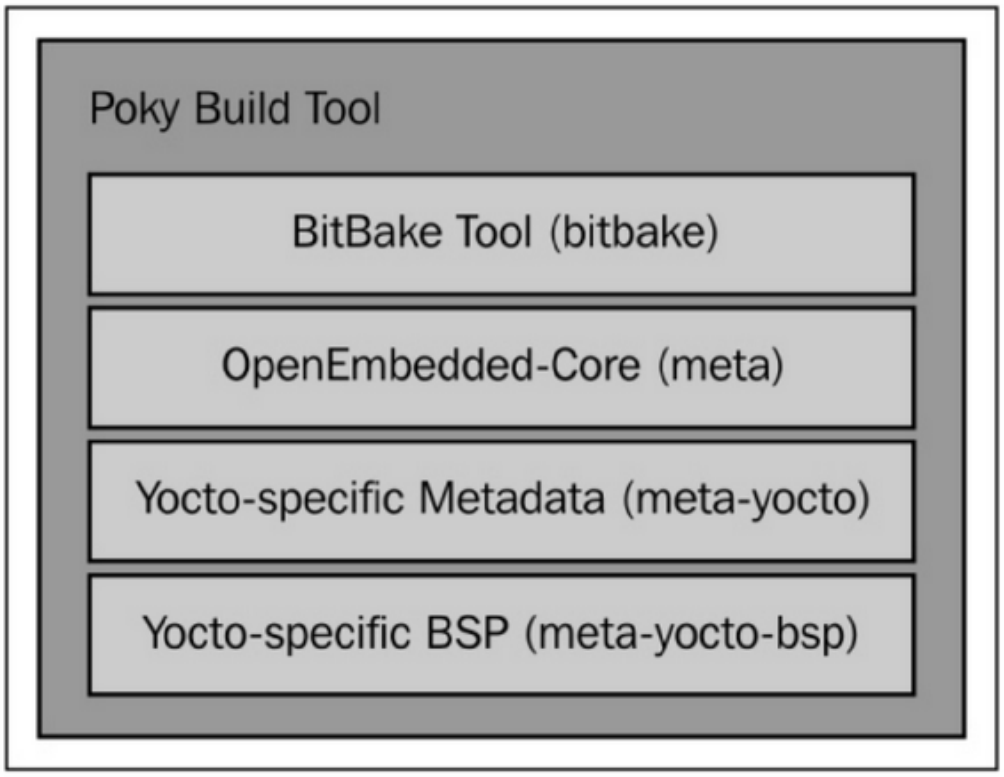

Figure 1.9: Poky build tool stack

The main concept on which the Poky Build System is constructed is that every aspect of a build is controlled by metadata, grouped into package recipes.

A recipe is then used by BitBake to set variables or define additional tasks to be performed during the build.

Chapter 2

Board Characteristics

This chapter goes into an in-depth exploration of the AM64x board, showing the major features exploited during the realization of the project. The chapter begins with the reasons behind the selection of the board. Next, the chapter explores through a comprehensive analysis of the distinctive processor characteristics of the AM64x board: the Arm Cortex-A53, Cortex-R5F, Cortex M4 and PRU cores. The analysis of the processors shows their individual attributes and how they can work together to solve more complex tasks. Furthermore, the discussion will explore the mechanisms for interprocessor communication, which is a core concept for the coordination among the heterogeneous cores.

2.1 Board Selection

The first task done during the internship was the selection of the board to use during the project. Different boards were analyzed during this process, each with its own set of characteristics. The boards analyzed were: Texas Instruments AM64x [9], featuring scalable ARM Cortex-A53 architecture; Renesas RZ/G2 MPUs [14], known for Linux support and versatile connectivity; Xilinx Zynq UltraScale+ MPSoCs [17], combining FPGA fabric with powerful ARM Cortex-A53 cores; Ingenic X2xx Series [15], offering solutions by Ingenic Semiconductor; and Rockchip RK3xx Series [13], known for its versatile applications and capabilities in embedded systems.

Board Selection					
Feature	Renesas RZ	Texas Instruments AM64x	Xilinx Zynq	Ingenic X2xx	Rockchip RK3xx
High-Performance MPUs	v	v	v	x	v
Heterogeneous Processors	v	v	v	v	v
Industrial ethernet protocols support	v	v	v	x	x
Multi-core Memory controller	v	v	v	v	v
Safety features availability	x	v	v	x	x

Figure 2.1: Board analyzed

The AM64x was the only board which met all the criteria, making it the perfect candidate for this project.

The first important element was the availability of different heterogeneous cores. The AM64x board has heterogenous multi-core architecure, offering a spectrum of processing cores like: Arm Cortex-A53, Cortex-R5F, Cortex-M4F and PRU cores. This diversity in processing capabilities enables the implementation of complex algorithms and tasks across different computational requirements.

Another important factor is the board's native support for Ethernet communication, including Gigabit Ethernet ports and protocols to guarantee seamless data exchange between the embedded system and external devices, facilitating efficient communication with regular PCs.

Additionaly, the board's robust hardware features, such as: large connectivity options, extensive peripheral support, and integrated security features, complement the application's demands in orchestrating reliable and secure communication in heterogeneous environments.

2.2 Introduction to AM64x

AM64x is an extension of the Sitara industrial-grade family of heterogeneous Arm processors. [8] AM64x is designed for applications in industrial automation, industrial communication, and other embedded systems application which require a unique combination of real-time processing and com-

munications with application processing.

AM64x combines two instances of Sitara's gigabit TSN-enabled PRU-ICSSG, two Arm Cortex-A53 cores, four Cortex-R5F MCUs, and a Cortex-M4F MCU domain.

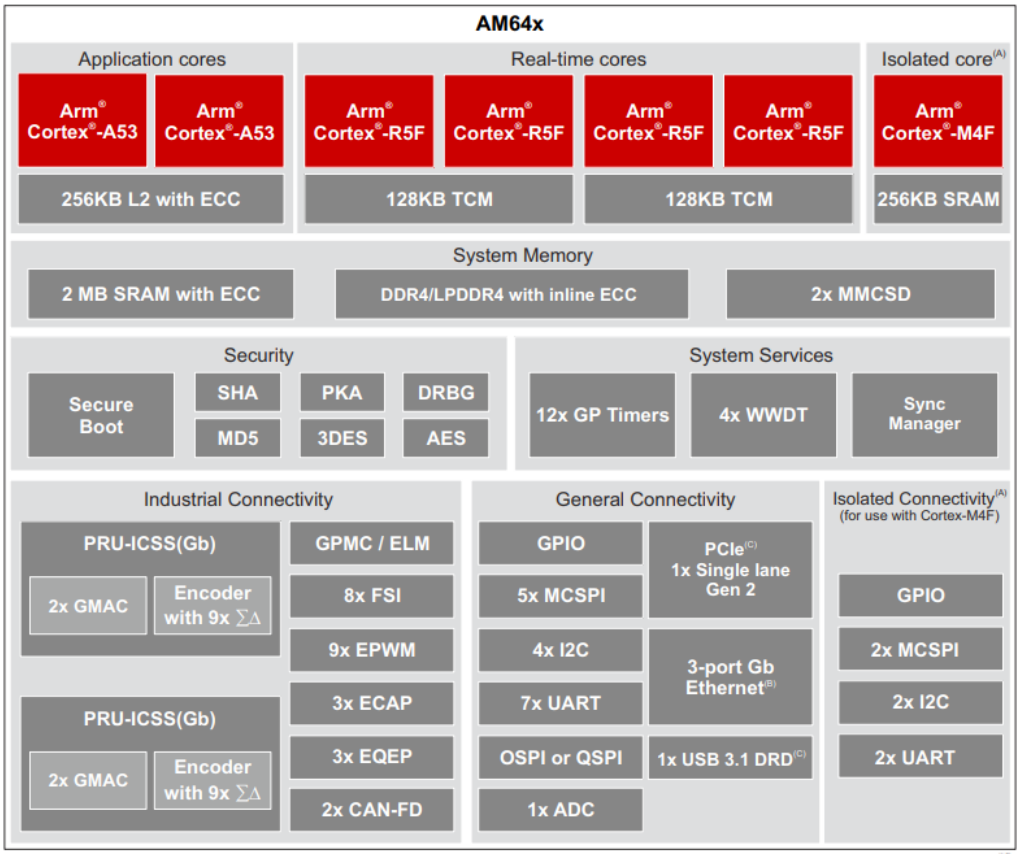

Figure 2.2: AM64x architecture

The Arm Cortex-A53 Cores are specifically optimized for higher-level processing tasks. They offer good performance for general-purpose computing and serving as a popular choice for running operating systems, such as embedded Linux. Their computational power enables the fusion of the Linux environment with the real-time capabilities offered by the other cores within the AM64x. The AM64x offers configurable memory partitioning, facilitating the division of memory allocation between the Linux cores and the real-time cores. Particularly, the Cortex-A53s can exclusively utilize DDR memory, while the internal SRAM can be flexibly allocated in various sizes to acco-

modate the needs of the Cortex-R5fs, either collectively or individually.

Within an embedded system, the most important objective is to provide real-time computation, and this is done through the high performance R5Fs, in the AM64x. Arm Cortex-R5F cores offers a robust set of real-time processing capabilities wihch are fundamental for deterministic and time-critical tasks. These cores are specifically designed to handle real-time operations, ensuring precise timing, reliability, and responsiveness in applications within embedded systems and industrial environments. The functionalities best suited for the R5Fs are those requiring deterministic behavior, such as control systems, motor control, and real-time monitoring. Their architecture is also optimized for safety operations, guaranteeing reliability and predictability in execution. As highlighted in the introduction, this is one of the most important aspect in the industrial automation applications. Moreover, the inclusion of quad-core Cortex-R5Fs in the AM64x provides the possibility of parallel processing, allowing for efficient handling of multiple real-time tasks simultaneously. The multi-core setup enhances the system's ability to manage complex control algorithms or handle several concurrent real-time processes without compromising performance or reliability.

The integrated Cortex-M4F core, coupled with dedicated peripherals within the AM64x series, enables the implementation of functional safety features, which is a crucial characteristic in many industries. The possibility of isolating these safety-critical elements from the rest of the SoC guarantees enhanced security and integrity.

The M4F core is developed to address digital signal control markets that demand an efficient, easy-to-use blend of control and signal processing capabilities. [2] The combination of high-efficiency signal processing functionality with the low-power, low cost and ease-of-use benefits of the Cortex-M family of processors satisfies markets.

Finally, in the architecture of the AM64x can be found an additional co-processor called PRU-ICCSG (Programmable Real-Time Unit for Industrial Communication SubSystem Gigabit). This subsystem is used for industrial communication, being able to perform real-time industrial communication. PRU cores are primarily sed for industrial communication, but they can also be used for other applications such as motor control and custom interfaces. The PRU-ICSSG frees up the main ARM cores in the device for other functions, such as control and data processing.

2.2.1 A53 Subsystem

The SoC implements one cluster of dual-core Arm Cortex-A53 MPCore, which is a multi-core variant of the A53 core, where each core can execute code independently from the other. It is based on the symmetric multiprocessor (SMP) architecture, where all cores have equal access to system resources like memory and peripherals.

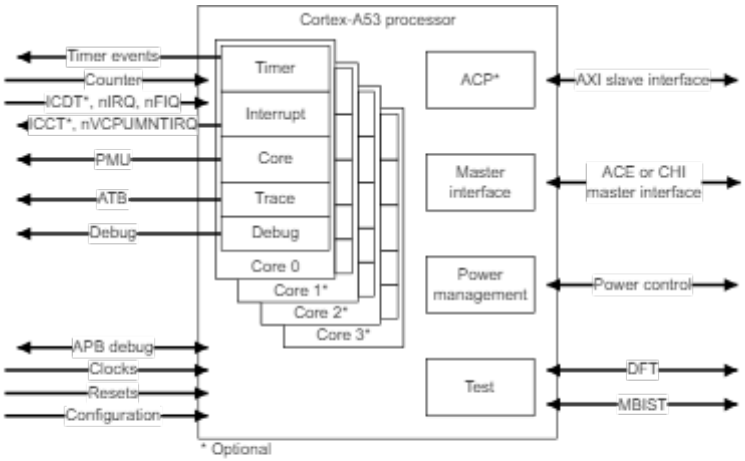

Figure 2.3: Cortex-A53 MPCore architecture with 4 cores

The Cortex-A53 processor is a high efficiency processor that implements the Armv8-A architecture. While maintaining the A53's energy efficiency, the Cortex-A53 MPCore enhances overall system performances through parallel processing.

The A53 CPU has the ability to execute 64-bit applications with the AArch64 execution state. It also has the possibility to execute 32-bit applications with the AArch32 state for retrocompatibility with the existing ArmV7-A applications.

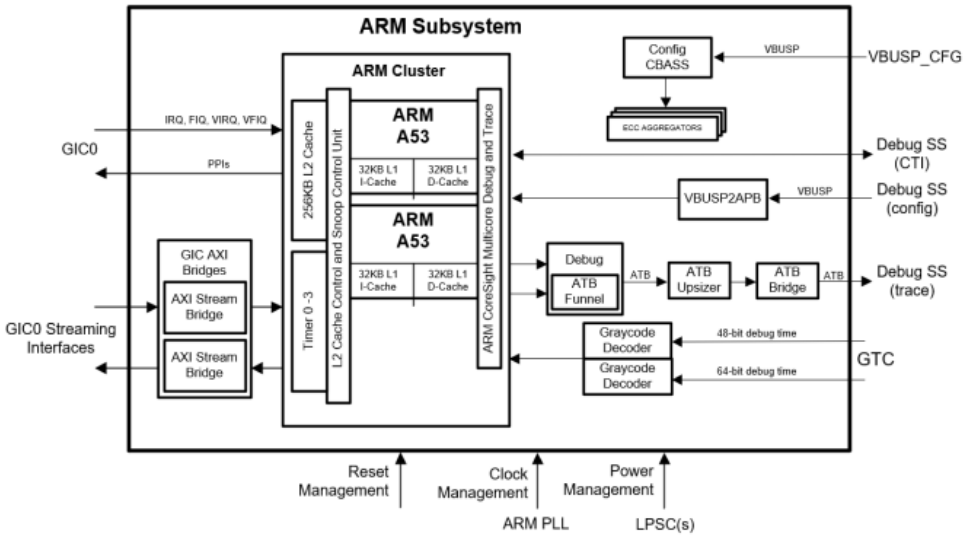

Figure 2.4: A53SS Block Diagram

Each core within the subsystem is equipped with 32KB L1 instruction and 32KB L1 data caches, complemented by a shared 256KB L2 cache, as illustrated in Figure 2.4. These cache configurations significantly contribute to optimizing data access and enhancing overall processing efficiencty.

Given the processing power of the Cortex-A53 core, it is usually used to run embedded Linux and combine the features offered by Linux with the real-time operations offered by the other cores. In this idea, the A53 can be used for general purpose computing and interact with the other cores to execute certain services which require to satisfy real-time contraints.

2.2.2 R5F Subsystem

The board selected for the project has two R5F subsystems, a dual-core implementation of the R5F processor, which is a version of the R5 processor with the floating point unit extension.

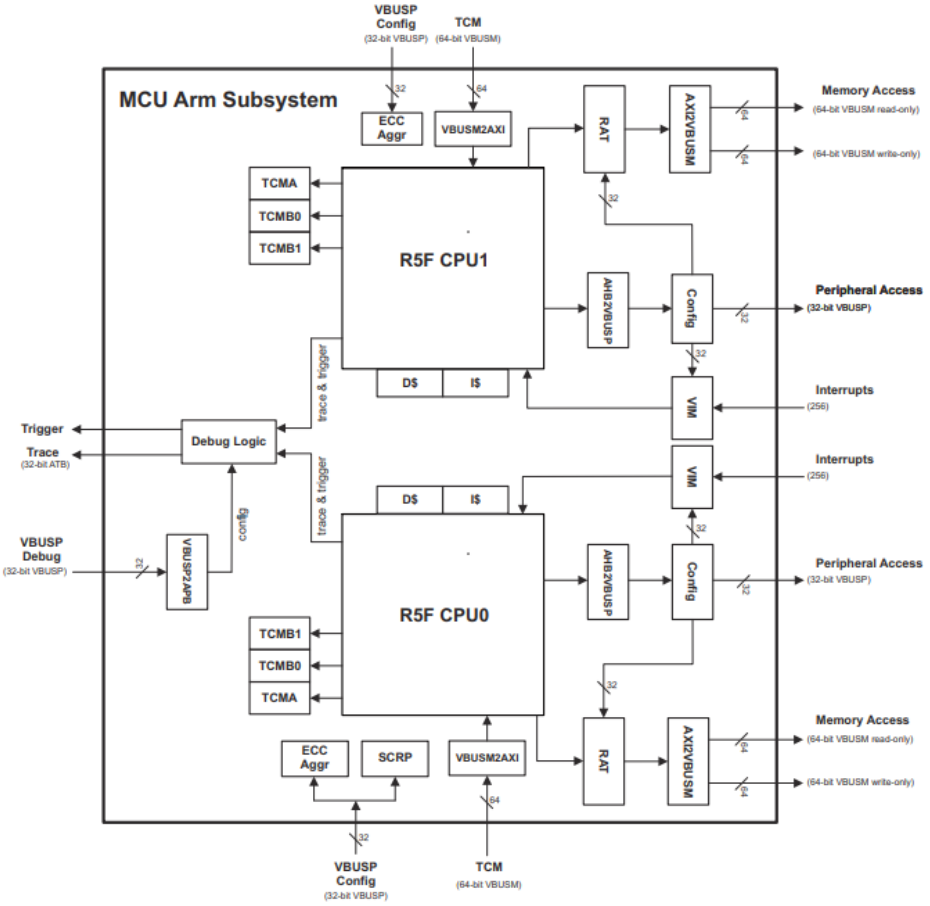

Figure 2.5: R5FSS Block Diagram

The architecture of the R5F core is the Armv7-R, which is specifically designed for real-time applications. The most important features are: deterministic behavior, safety and reliability.

It features an Harvard memory architecture, which separates the cache for instructions and the cache for data. Each core has a total of 32KB of L1 cache: 16KB for instruction and 16KB for data.

To enable fast memory access the R5F has two tightly-coupled memories (TCMs), which are low latency, tightly integrated memories that can be used for instructions or data. The total TCM available for each core is of 32KB. It has similar performance to accessing instructions or data in cache. [4]

One of the most important features of the TCMs is the possibility to be accessed from external sources. This makes possbile to preload data in the memory, such as instructions before they are needed from the core. It is also possible to process data and save it on the TCMs, and external sources can directly access the data, without communicating with the core.

The TCM can be used to hold time-critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache is undesirable. In addition, you can use it to hold ordinary variables, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks. [7]

The subsystem can operate in one of two modes: split or single-core mode, which has to be decided during bootstrap. In split mode, each core of the subsystem is considered as a standalone processor, working completely independent from the other, and each of them has dedicated RAMs and interfaces. In single-core mode, only one of the cores is operating, but it has available the TCM of the second core, which could be offered advantages in performance if the processing power of two cores is not needed, but a lot of data has to be processed.

2.2.3 M4F Subsystem

The last core of the system is an Arm Cortex-M4F running safety processing. Similar to the R5F, also the M4F has the extension for floating point unit.

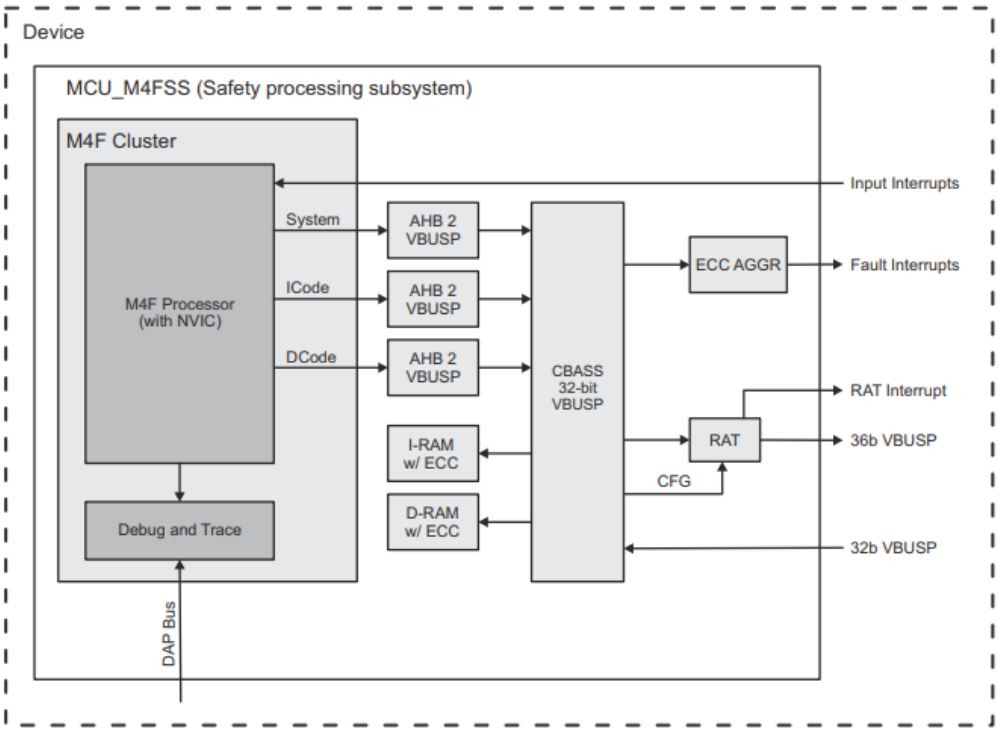

Figure 2.6: M4FSS Block Diagram

The Cortex-M4F processor is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. This processor is intended for deeply embedded applications that require fast interrupt response features. [6] The processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design. The M4 processor implements a version of the Thumb instruction set based on Thumb-2 technology, ensuring high code density and reduced program memory requirements.

The M4FSS has a total of 256Kb of SRAM divided into two banks: 192KB of I-RAM, and 64KB of D-RAM. The I-RAM meory is intended mainly for M4F's instruction code, and D-RAM for M4F's data. The M4F allows concurrent fetch for instruction code and data via dedicated buses. The M4FSS supports unified memory for both banks, which means that instruction code and data can be placed in any bank, although, it is recommended that the data saved is the intended one for efficiency purposes.

The RAM also has Error-correcting code (ECC), which is used to detect

and correct data corruption which occurs in memory. To enhance safety, the core also has a Memory Protection Unit (MPU), giving the possibility to privileged software to define memory regions and assign memory access permission and attributes to each of them.

The M4FSS incorporates a dedicated local reset input. This makes possible to preload code in the M4F RAM and then release the reset, in this way the processor will have the instructions it needs when it starts.

During the boot sequence, another processor will start the M4F, from that point onward the M4F will be isolated form other cores, running safety processing. This isolation from other cores on the board is important to maintain a highly secure and predictable compuational environment, essential for safety-critical applications.

2.2.4 PRU-ICSSG

Industrial communication within Sitara proessors and microcontrollers is efficiently managed by the Programmable Real-Time Unit Industral Communication Subsystem (PRU-ICSS). The PRU-ICSS is a co-processor subsystem that houses the PRU cores and Ethernet media access controllers (EMACs), responsible for executing low-level industrial Ethernet and fieldbus protocols via firmware. Higher-level protocol stack components are implemented in software running on the Arm cores.

PRU cores are mainly dedicated to industrial communication but can be harnessed for other custom applications, freeing the main Arm cores for more general control and data processing.

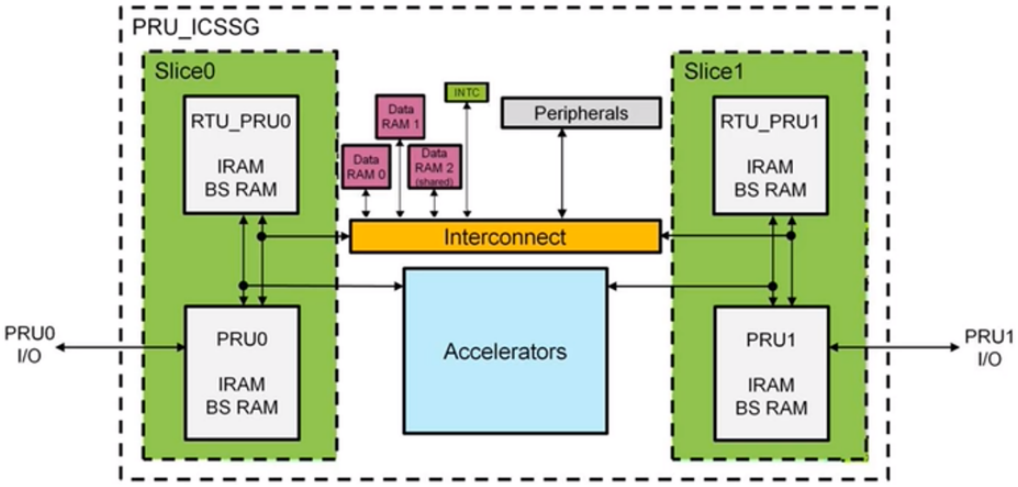

Figure 2.7: PRUICSSG Architecture

PRU-ICSSG represent the next generation of PRU-ICSS, building on the capabilities introduced in the predecessor (PRU-ICCS). At its core, each PRU-ICSSG has 4 32-bit RISC cores:

1. 2 PRU cores
2. 2 Auxiliary Programmable Real-Time Units known as RTU-PRUs

A notable enhancement in the PRU-ICSSG, compared to its predecessor, is the inclusion of internal accelerators. These accelerators play a crucial role in data processing and movement, contributing to the achievement of both real-time and Gigabit-level speeds.

PRU-ICSSG Cores

PRUs, or Programmable Real-Time Units, are specialized processing units embedded in the Sitara family of microcontrollers. PRUs are designed for real-time applications that require low-latency and deterministic performance. They offer a dual-core architecture, with high programmability, enabling custom firmware to be executed in real time. PRUs are especially suitable for tasks such as industrial control, motor control, communication interfaces,

and custom I/O control. They provide low-latency, direct access to processor pins, making them efficient for interfacing with external devices. PRUs can be used alongside the main ARM CPU, sharing resources as needed.

Within the subsystem there are two type of cores: PRUs and RTUs. The key distinction between them lies in their I/O access. PRUs are versatile, capable of I/O control and data processing, while RTUs are exclusively designated for data processing. Both, however, share access to PRU_ICSSG and SoC resources.

The PRU-ICSSG is divided in two symmetrical slices. Each of them containing one PRU core and one RTU_PRU core. The content within each slice is only accessible to the PRU and RTU cores within that slice, whereas content outside the slices can be accessed by any core. Each core has its dedicated instructions and broadside RAM, along with a RAM dedicated to each slice, shared by the cores belonging to that slice.

Each core can operate independently or in coordination with ARM cores or other PRU/RTU cores.

PRU cores are designed as non-pipelined, single-cycle execution engines. This means that all instructoins, except for memory reads and writes, are completed within a signle cycle. As there is no pipelining, PRU cores offer full determinism, ensuring that instructions are not rearranged during execution.

2.3 Memory Map Layout

The memory map of an embedded device provides a layout of the memory space, indicating how different types of memory are organized and used within the device.

The two most important types in the AM64x are: MSRAM and DDR memory layout.

MSRAM

MSRAM, or SRAM (Static Random-Access Memory), stands out of its high speed, low power consumption when idle, reliability, and lack of a need for

data refreshing. However, it's relatively expensive and offers lower storage densite compared to DDR SDRAM.

The AM64x SoC offers a total of 2MB of MSRAM, divided into eight individual banks, each with a size of 256KB.

Each of the four R5s cores has access to dedicated 256 KB private bank for exclusive use, providing isolation and dedicated memory space for each core.

In addition to core-specific banks, there are shared banks accessible by all cores, facilitating inter-core communication, shared data storage, and support for applications requiring more memory than their private banks can offer. Shared banks play an important role in enabling communication and data sharing between different cores within the SoC, providing collaborative processing capabilities.

This memory layout offers flexibility for applications. When using only one core, applications can utilize the private banks reserved for other cores as needed.

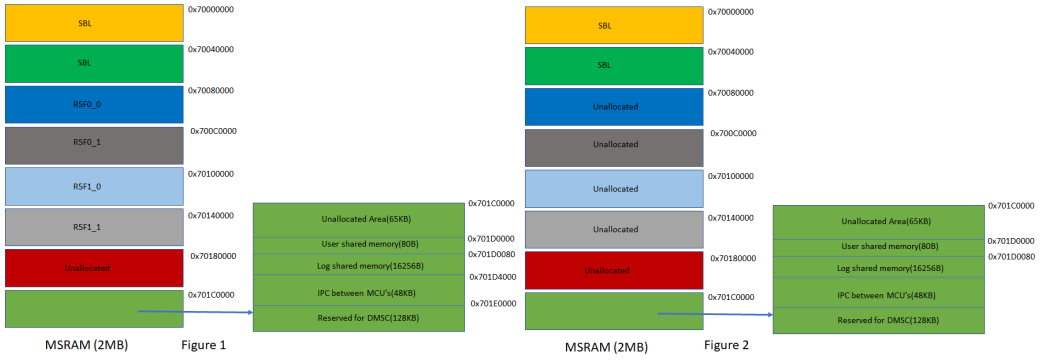

Figure 2.8: MSRAM layout

The first 512 KB of MSRAM is set aside for a specific purpose, it's reserved for the System Bootloader (SBL). The SBL is responsible for initializing the system, and it contains essential elements such as the System Controller Firmware (SYSFW) and SYSFW Board Configuration data.

The reason this memory is reserved is to create a dedicated space for the combined bootloader image. This image ensures that SYSFW is properly loaded into the DMSC Cortex M3 processor. Any attempt to modify or use tihs reserved memory for other purposes in the linker file should be avoided.

If the reserved area is accidentally overlapped with an application's memory space, it can lead to problems during the boot process. Specifically, during the boot process, the SBL loads the core image, and if the reserved memory is not respected, there's a risk of overwriting the SBL itself. This could result in a malfunctioning or unstable system. To prevent such issues, the SBL has built-in checks that will flag an error if it detects any section of an application falling within the reserved memory region. These checks are in place to ensure the system's integrity and reliable boot-up.

DDR

DDR is known for its high data throughput, cost-effectiveness, synchrnous operation with the system clock, and data transfer on both edges of the clock signal. This double data rate transfer significantly increases bandwidth and is ideal for memory-intensive applications. DDR SDRAM also provides greater memory density compared to MSRAM.

The layout of the DDR differs, depending on which Operating System is running on the cores.

If embedded Linux is running on the A53 core, the DDR is divided into sections, one for every core, which has to be used in the RTOS application.

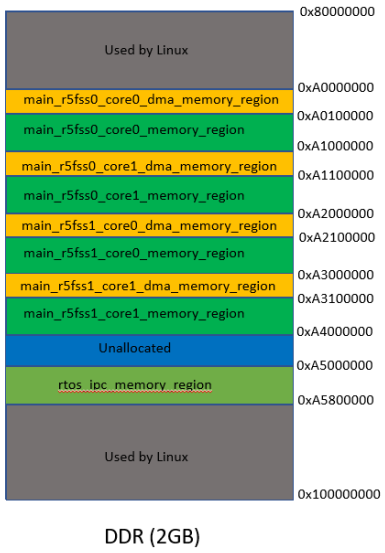

Figure 2.9: DDR layout with linux running on A53 core

If an RTOS or a bare-metal application is running on the A53 core, the application needs to be loaded into DDR for booting. The DDR is not allocated in this case, and it can be used by the all the cores, specifying it in the linker file.

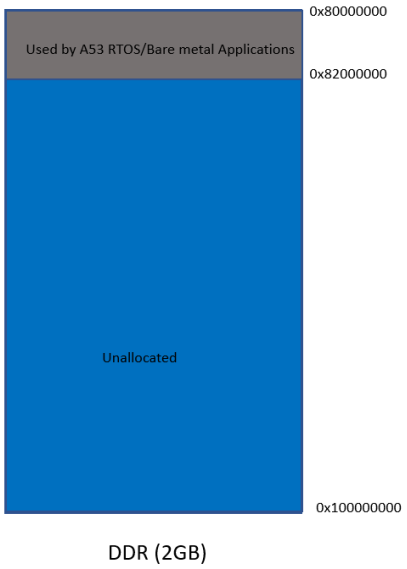

Figure 2.10: DDR layout with RTOS or baremetal application on A53 core

2.4 Networking

Advances in the embedded systems have led to new functionalities to be required. The ability to communicate over the internet with other devices.

Networking is a broad term used to cover Ethernet (IEEE 802.3), EtherCAT Profinet and other ethernet-like communication protocols used in industrial, automative and other general use cases.

Networking is supported using the following hardware peripherals:

- Common Port Switch (CPSW): CPSW subsystem provides IEEE 802.3 standard Ethernet gigabit speed packet communication for the device and can also be configured as an Ethernet switch. CPSW supports RGMII and RMII interfaces.

- PRU-ICSSG: PRU-ICSSG is firmware programmable and can take on various personalities like Industrial Communication Protocol Switch (for protocols like EtherCAT, Profine, EtherNet/IP), Ethernet Switch, Ethernet MAC, Industrial Drives, etc. PRU-ICSSG supports RGMII and MII modes.

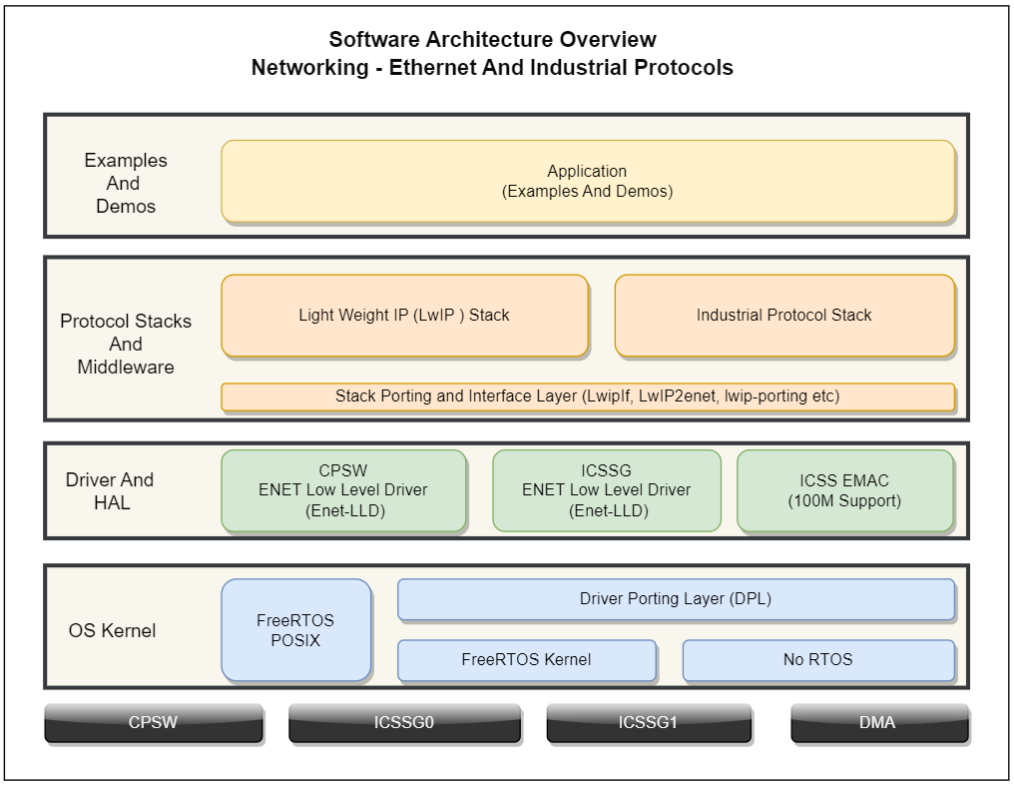

Figure 2.11: Networking Software Stack

2.5 Inter Processor Communication

Inter-Processor Communication (IPC) is a fundamental aspect of multi-core systems like the AM64x SoC. In such systems, multiple CPUs or processor cores works together to achieve specific tasks or run various applications. These cores may need to communicate and share daa with each other to realize a larger system-level application or to coordinate their activities. IPC mechanisms facilitate this communication.

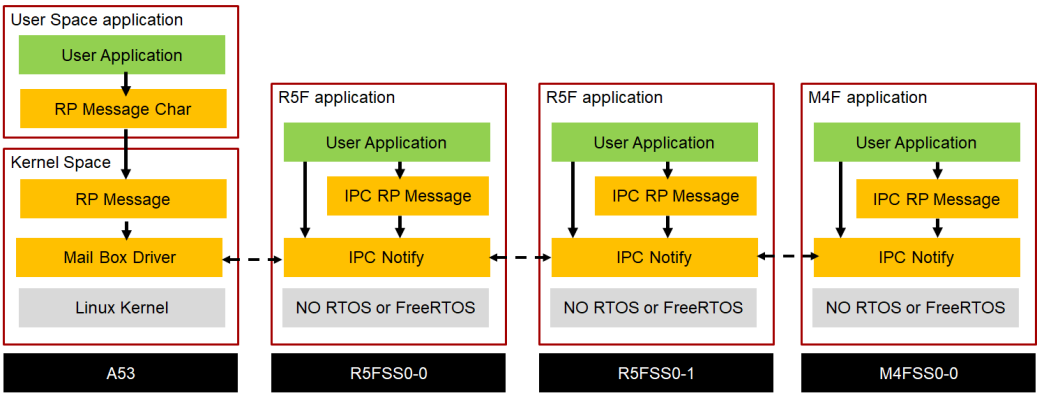

Figure 2.12: IPC mechanisms

There are two IPC mechanisms to realize communication between real-time cores:

1. IPC Notify
2. RPMessage

Developers can leverage IPC Notify alongside RPMessage, to fine-tune their applications based on their specific needs and requirements.

2.5.1 IPC Notify

IPC Notify defines APIs for enabling low-latency inter-processor communication between different CPU cores. These APIs prioritize speed and efficiency and are therefore constrained in terms of features.

To achieve low latency, the underlying implementation employs hardware mechanisms to interrupt the receiving CPU cores. Additionaly, it utilizes hardware FIFOs when available, and in cases where hardware FIFOs are not present, it resorts to using software FIFOs based on fast internal RAM to transport message values.

In the case of AM64x, IPC notify utilizes hardware mailbox-based hardware FIFOs to transport messages and interrpt the receiving CPU core. This ap-

proach ensure that messages are exchangesd with minimal delay and high efficiency, making it suitable for scenarios where speed is a primary consideration.

The first limitation to have all the advantages offered by IPC notify is the size of the message: IPC Notify combines the message content and client ID into a single 32-bit value, limiting the content of the message to less than a byte.

2.5.2 RPMessage

RPMessage, or Remote Processor Message, is a communication method that enables CPUs to exchange messages in the form of packet buffers. These messages are directed to specific logical endpoints on another CPU, facilitating efficient inter-processor communication.

RPMessage exploits shared memory between cores to exchange the messages. The sender places a packet in a buffer inside the shared memory. After that, it has to notify the receiving core that there is a new message to process. This notification is done using a hardware interrupt mechanism, similar to IPC Notify.

In contrary with IPC Notify, RPMessage allows for variable packet sizes. When both ends are using real-time operating systems, the minimum packet size is 4 bytes, but it can be configured by the developer to use bigger sizes, although 512 bytes should be taken as upper limit. If Linux is involved in the communication the size of the packet is fixed to 512 bytes, configured in the Linux kernel.

Every core can establish multiple logical endpoints, where the message can be delivered to, allowing for multiple communication channels between CPUs.

2.5.3 Linux IPC

The RPMsg char driver serves as an interace for user-space processes to access RPMsg endpoints. These endpoints are used for communication between different applications and can be uniquely accessed by requesting specific interactions with the remote service. What is important to note is that

the RPMsg char driver supports the creation of multiple endpoints for each RPMsg char devicce that has been initialized. This means that a single RPMsg char device can be used for different instances or applications.

When these endpoints are created, they appear as individual character devices within the /dev directory. Underlying this setup, the RPMsg bus operates on top of the VirtIO bus. Each time a VirtIO name service announcement message is sent, a new RPMsg device is generated, and it is intended to be bound to an RPMsg driver. These RPMsg devices are dynamically generated, initiated by an announcement message from the remote processor. This message contains information about the service's name, source, and destination addresses.

The handling of this announcement message is the responsibility of the RPMsg bus, which dynamically creates and registers an RPMsg device that represents the remote service. Once a relevant RPMsg driver is registered, it is immediately probed by the bus, and this allows both sides to begin exchanging messages.

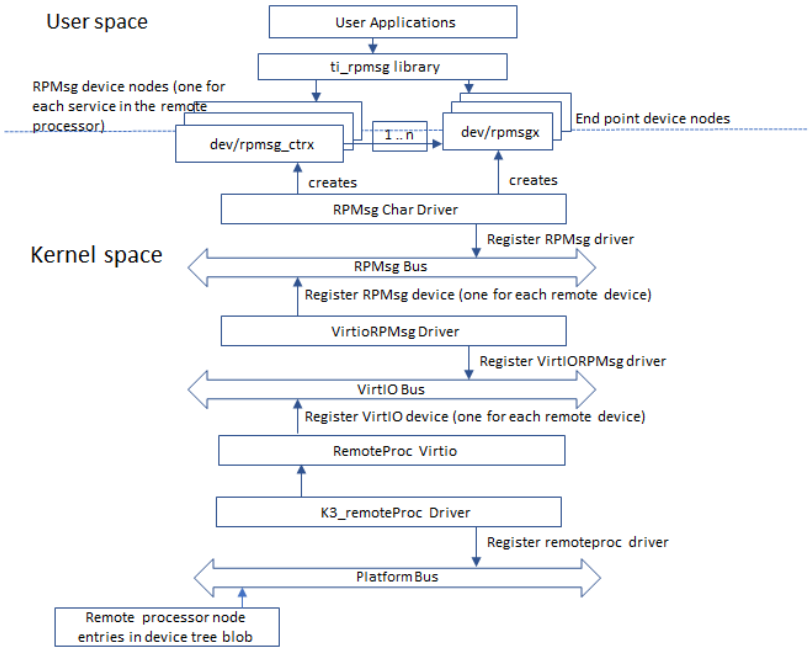

Figure 2.13: RPMsg library stack

2.5.4 PRU IPC

Given that PRUs can offload tasks from the primary cores, establishing communication between the PRU cores and the ARM cores becomes essential.

PRU IPC module provides APIs for communicating with the PRU cores with low latency. The communication is done in the form of blocks of data transfer in one exchange. Options are available for configuring multiple buffers and blocks per buffer.

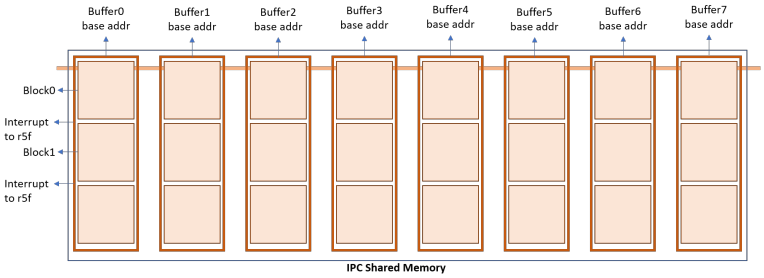

Figure 2.14: PRU IPC

PRU write to the shared memory and creates an interrupt even once a certain number of samples, which are equal to the block size, have been written into IPC Shared Memory. The PRU writes one block of data for all the buffers at a time, and then moves on to the buffer after that.

After writing the block of data, the PRU sends an interrupt to the destination of the message, which can read it from the shared memory.

Communication between Linux and PRUs

There are two vrings provided per PRU core, one vring is used of rmessages passed to the Linux core and other one is used for receiving messages. System level mailboxes are used to notify cores (ARM or PRU) when new messages are waiting in the shared buffers.

On the ARM Linux side, RPMsg messages are received in kernel space. An interface module is provided (rpmsg_pru) that creates a character device in user space so that users can perform operations (read or write) on a character device in the file system to communicate with PRU cores.

On the PRU side, an RPMsg library is provided in the Software Support Package that aims to abstract the communication, providing developers with send/receive APIs for simple communication with other cores.

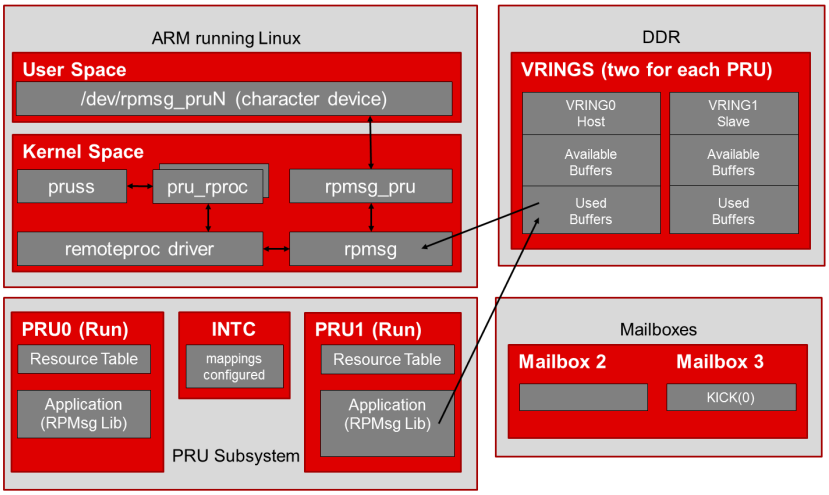

Figure 2.15: Components for Linux and PRU IPC

ARM to PRU

Steps for the ARM core:

1. Allocate or Retrieve Buffer: the ARM host starts by either allocating or acquiring a used buffer from the slave Vring.
2. Data copy: the data to be transferred is copied into the chosen buffer.
3. Add to Available List: The filled buffer is then added to the available list in the slave Vring.
4. Notify the PRU: to signal the PRU, the ARM host kicks the slave Vring by sending its index via a message in Mailbox 2.

Steps for the PRU core:

1. Detect Kick: The PRU monitors Mailbox 2 for a kick with the index of the relevant Vring, indicating that data is ready for reception.

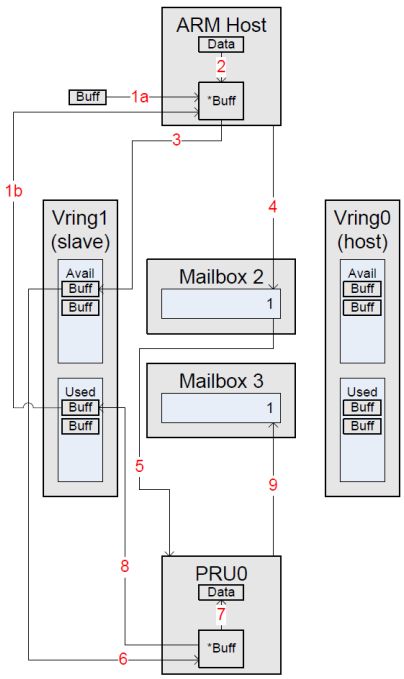

Figure 2.16: Steps needed to send a message from the ARM core to the PRU

2. Retrieve Available Buffer: the PRU obtains the available buffer from the slave Vring.

3. Data extraction: the data to be received is copied from the buffer in the previous step.

4. Add to Used List: the now empty buffer is added to the used list in the slave Vring.

5. Notify the ARM Host: to inform the ARM host about the completion, the PRU kicks the slave Vring by writing its index into a message in Mailbox 3.

PRU to ARM

Steps for the PRU:

1. Retrieve Available Buffer: the PRU starts by obtaining an available buffer from the host Vring.

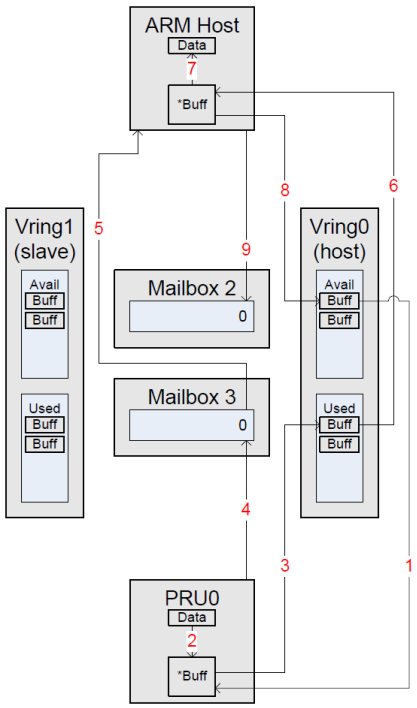

Figure 2.17: Steps needed to send a message from the PRU to the ARM core

2. Data copy: the data to be transferred is copied into the buffer of the previous step.

3. Add to Used List: The filled buffer is then added to the used list in the host Vring.

4. Notify the ARM: to signal the ARM host, the PRU kicks the host Vring by writing its index into a message in Mailbox 3.

Steps for the ARM host:

1. Kicked Vring Detection: an interrupt is triggered, signaling that Mailbox 3 was kicked with the index of the host Vring. This notifies the ARM host that data is available for reception. of the relevant Vring, indicating that data is ready for reception.

2. Retrieve Used Buffer: the ARM host retrieves the used buffer from the host Vring.

3. Data extraction: the data to be received is copied from the buffer acquired in the previous step.
4. Add to Available List: the now empty buffer is added to the available list in the slave Vring.
5. Notify the PRU: to inform the PRU about the completion, the ARM host kicks the host Vring by writing its index into a message in Mailbox 2.

2.6 System boot

The boot process is a complex process in the AM64x, since it has different heterogeneous processors. An additional core, a Cortex-M3, is reserved for the booting process and other management functions. This core is also called Device Management and Security Controller (DMSC).

2.6.1 Boot without Linux

The bootflow, in case the A53 is not running Linux, consists of two main steps that occur when the device is powered on:

1. ROM boot
2. SBL boot

In the ROM boot step, a built-in component of microcontroller hardware, called ROM bootloader, takes control. Its primary responsibility is to initiate the next-stage bootloader, the SBL. The ROM bootloader is generally kept in secure, read-olny memory to ensure its integrity and security, working as root of trust for the rest of the operations.

After the ROM bootloader, the SBL takes control which is typically more versatile and configurable than the ROM bootloader. The SBL is responsible for orchestrating the boot process of the actual application. It can perform complex bootloading tasks, including initializing different processor cores and managing the loading and execution of the application software. It may also

handle the parsing and loading of the application image into the appropriate memory locations and take care of other initialization tasks required for the application to run successfully.

Initial Boot Flow

The initial boot flow is the first part of the boot process and it specifies how everything starts and how all the cores that belongs to the system are initialized and started.

The core components to start the initial boot process are:

1. DMSC
2. R5F core
3. Boot pins

The DMSC is a core dedicated for the central management of the device. [4] Upon powering the system, the DMSC is the first core that starts, executing the first set of instructions on the AM64X device, through the DMSC ROM code. Subsequently, it will run the SYSFW code, which provides runtime functions that help the system to be configured and run correctly.

The Cortex R5F acts as the boot controller of the system. It is the core which manages the rest of the boot flow on the device, after it has been released from reset by the DMSC. When it comes out of the reset, it executes the R5 ROM code, and eventually it will run the SBL.

The boot pins are a set of pins in the board which specifies where the SBL image is located and how the peripheral storing it is configured.

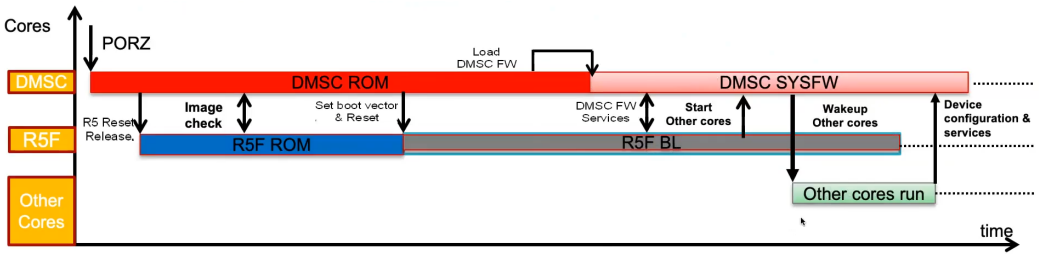

Figure 2.18: Initial boot flow of the SoC

The first thing after the Power On Reset (PORZ) is running the DMSC ROM, which in turn starts the R5F ROM code. The two of them will collaborate to realize the boot of the system.

After that, the SBL is needed for the boot process. The SBL will be loaded in the R5 from the peripheral specified with the boot pins. Before loading it, the DMSC validates the image's integrity. The R5 will find the image for the SYSFW, store it in its internal memory and the DMSC will check that the firmware is valid before storing in its internal memory and executing the code.

The next stages of the boot process will be done by the R5, in conjunction with the services offered by the SYSFW. As the other cores start up, they will use SYSFW services to check and configure the rest of the device.

Boot Process flow

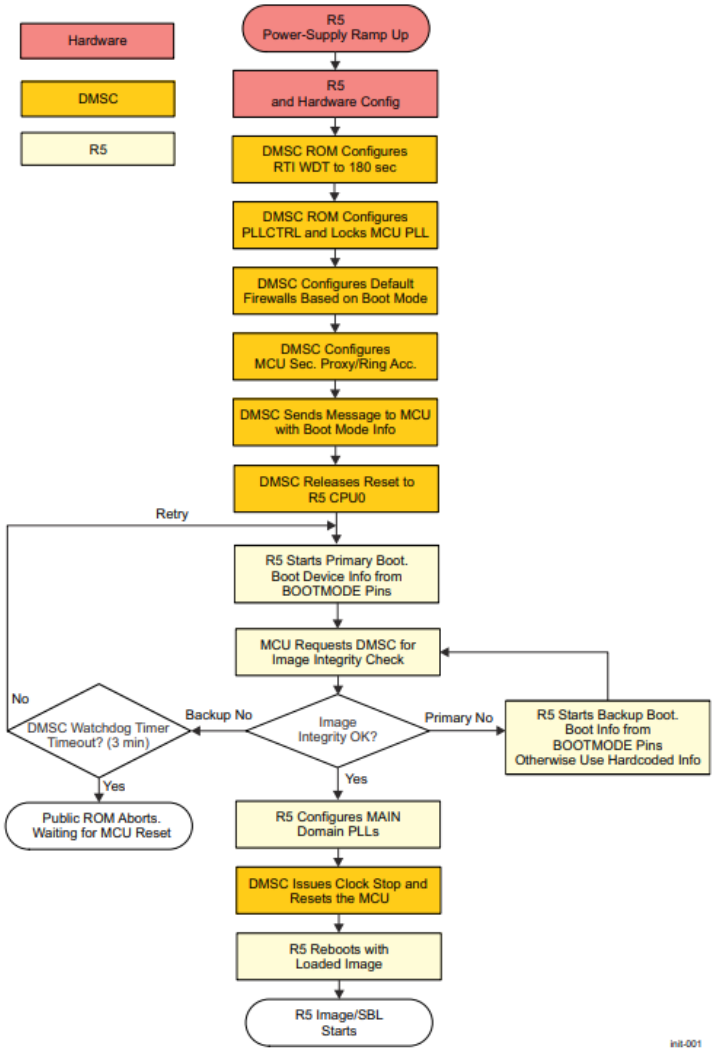

Figure 2.19: Flow of the boot process

In the boot process, the DMSC works as the boot controller for the public ROM. It takes care of essential configurations and releases the reset for R5's CPU. The R5 core checks the boot mode pins and sets up the necessary peripheral interface to access a boot image. After conducting a cursory image check, the verified boot image is handed over to the DMSC.

The DMSC ROM then takes over, performing code verification and routing the boot image to the on-chip RAM. Following this, the R5 core transitions

into an idle state. DMSC ROM code asserts a reset ot the MCU, redirects the boot vector to the freshly loaded image, and releases the reset, thereby initiaing a restart of R5. This restart occurs with the Public ROM code fully disconnected.

Finally, the execution of the Public ROM code is triggered after a cold or warm reset, marking the completion of the boot process.

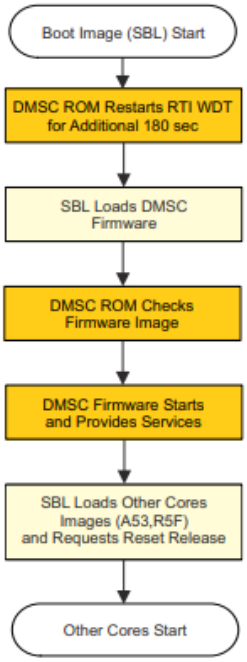

Figure 2.20: Steps done by the SBL

Upon the reset of R5 and the initiation of SBL execution, the DMSC ROM performs a restart of the RTI (Real-Time Interrupt) watchdog timer, extending the timeout for an additional 180 seconds. During this time, it becomes imperative for SBL to load the DMSC firmware provided by Texas Instruments; failure to do so may result in a preemptive MCU reset, serving as a protective measure against potential software misbehavior.

One of the primary responsibilities of SBL is to load the DMSC firmware. Only after successfully executing this task can SBL proceed to load the images for other processors and request a reset release from the DMSC firmware for those cores.

2.6.2 DMSC

The DMSC performs the following functions:

- Device Management
- Configuration of boot vectors and control of the reset release of the R5 core
- IPC configuration via main DMSS rings and Secure Proxy
- PLL configuration (R5 and SA2UL)
- X509 certificate parsing
- SA2UL configuration to SHA512 for image integrity checks
- DMSC fimware loading

In the case of AM64X the firmware loaded by the DMSC is the System Controller Firmware (SYSFW). It acts as a centralized server for the SoC. The services it provides are:

- Resource Management: SYSFW manages system resources, ensuring that various hardware components and resources are allocated, accessed, and controlled efficiently. This includes tasks such as configuring and managing peripherals, memory regions, and interconnects. It is essential to avid conflicts between different components of the SoC.
- Power Management: SYSFW is responsible for controlling the power states of different parts of the SoC. It manages power modes and transitions, ensuring that the SoC operates in power-efficient states when possible.
- Security: SYSFW enforces security policies, handling secure boot processes, and managing cryptographic functions to protect the system from unuathorized access or tampering. It may be involved in setting up and managing hardware security features.

Given the importance of the task performed by the SYSFW, it needs to be loaded and executed before the Secondary Bootloader (SBL) or any other part of the system can perform their tasks effectively.

2.6.3 Linux boot

The initial boot flow is similar to the one presented before, even if Linux has to be run. The DMSC always runs the SYSFW and starts the booting process. The R5F will start loading all the images for the necessary components of the system: ATF, OPTEE, and A53 SPL, all coming from the boot media, specified through the boot pins.

1. ATF: ARM Trusted Firmware, is a software component that secures booting, manages hardware, and establishes trusted environments on Arm-based systems.
2. OPTEE: establishes a secure environment on Arm-based systems for running and managing trusted applications, ensuring isolation and security from the normal system operations.
3. A53 SPL: small piece of code or bootloader necessary for initializing the system hardware, which will run on the A53 core.

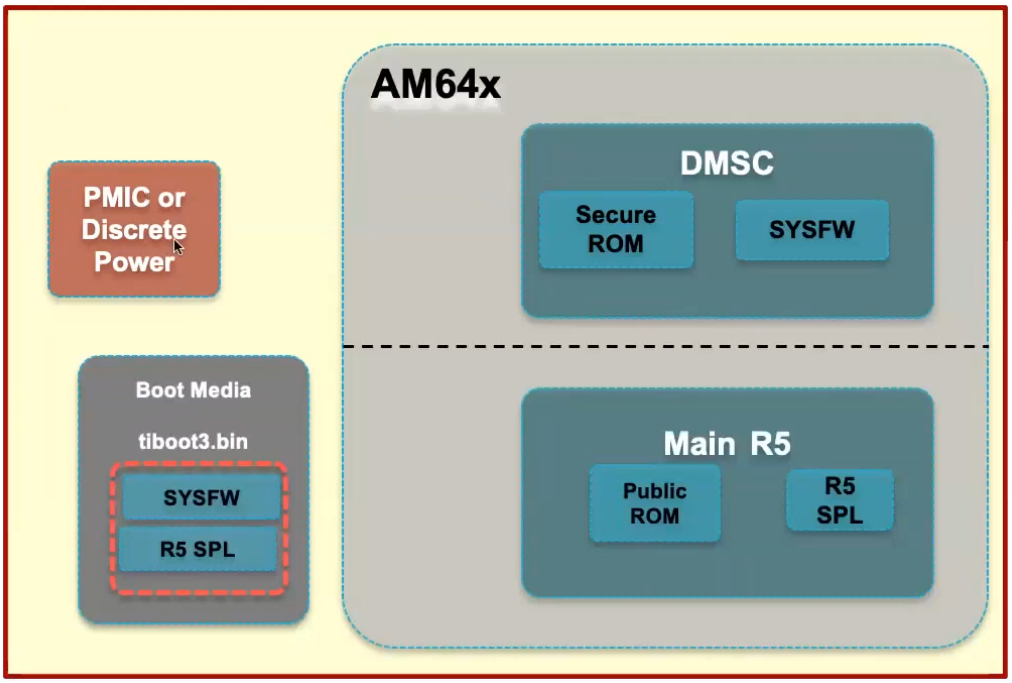

Figure 2.21: Components needed for the boot of Linux

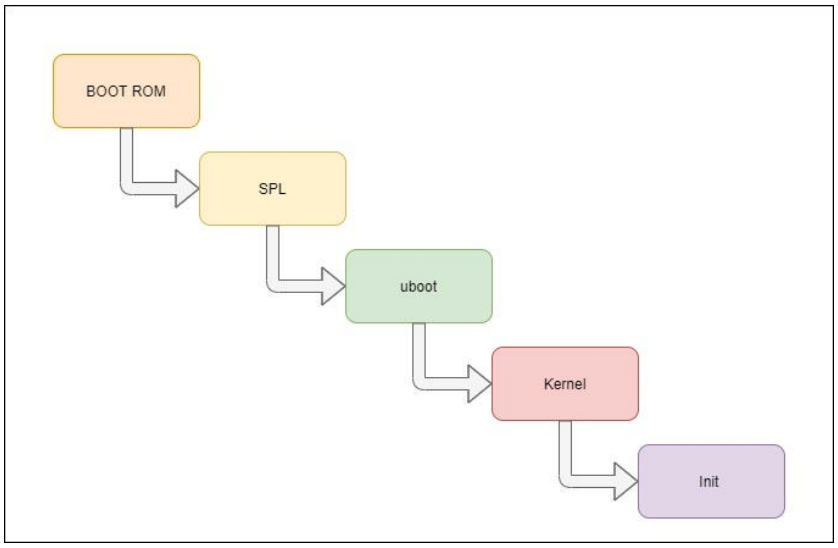

Figure 2.22: Linux Boot Flow

The ROM copies content of the bootloader to static RAM. SRAM memory is limited, due to physical reasons and we only have few KBs for bootloader. Usually, regular bootloader (e.g. U-Boot) binary is bigger than that. So we need to create some additional bootloader, which will initialize regular RAM and copy bootloader to RAM, and then jump to execute that regular bootloader. This additional bootloader is usually referred as uboot-SPL (which stands for Secondary Program Loader).

The bootloader (uboot) decompresses the Linux kernel into RAM from the specified peripheral. It then executes a jump to the kernel's first instruction, starting the execution of the system.

Chapter 3

Publish/Subscribe communication

Publish/subscribe (pub/sub) technology encompasses a wide number of solutions that aim at solving a vital problem pertaining to timely information dissemination and event delivery from publishers to subscribers. [24]

The pub/sub model emerged to address the need for more flexible communication models and systems, reflecting the dynamic and decoupled nature of applications. This approach aims to reduce the burden on application designers by offering infrastructure components that connect the various entities within large-scale settings. [19]

In a pub/sub system there are two entities which can participate to the communication: the publishers and the subscribers. Subscribers have the possibility to express their interest in a type of event or a pattern of events, and they are notified when a publisher generate an event which they are interested in.

The fundamental system model for publish/subscribe interaction (Figure 3.1) relies on an event notification service providing storage and management for subscriptions and efficient delivery of events. Such an event service represents a neutral mediator between publishers, acting as producers of events, and subscribers, acting as consumers of events.

Subscribers register their interest in events by calling an operation on the event service, without knowing the effictive source of those events. The sub-

scription information remains stored in the event service and is not forwarded to publishers.

When a producer generates an event, the event service propagates the event to all relevant subscribers; it can also be seen as a proxy for the consumers.

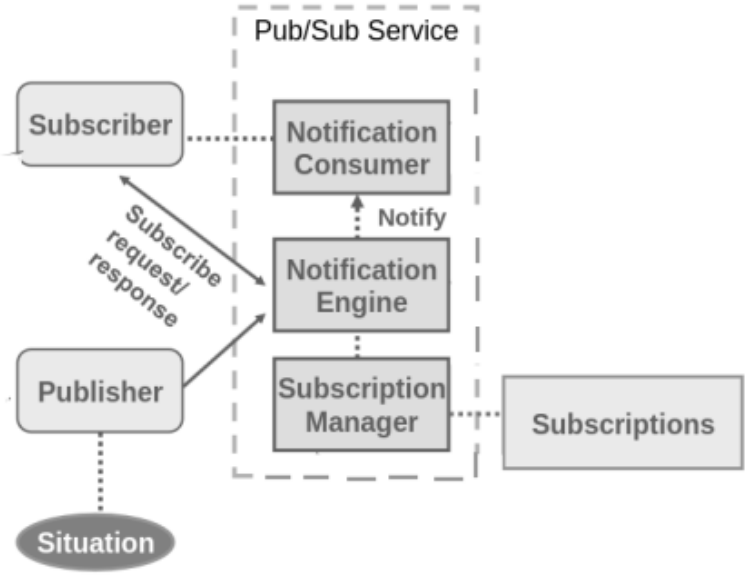

Figure 3.1: General architecture of a pub/sub middleware

The biggest advantage in using pub/sub communication is the decoupling in space, time and synchronization between publishers and subscribers. Decoupling the production and consumption of information increases scalability by removing all explicit dependencies between the interacting participants. In fact, removing these dependencies strongly reduces coordination and thus synchronization between the different entities, and makes the resulting communication infrastructure well adapted to distributed environments that are asynchronous by nature. [19]

With space decoupling publishers and subscribers operate without direct knowledge or dependencies on each other. Publishers disseminate envets through an intermediary event service, unaware of the subscribers receving these events or their count. Similarly, subscribers obtain events indirecly this sevice, without knowledge about the publishers or their numbers. This design promotes system flexibility and scalability. It allows to add and remove components without corrupting the entire system, offering anonymity betewen

publishers and subscribers while enabling a more modular and adaptable architecture.

Time decoupling in a pub/sub system is a pivotal feature that disengages the temporal dependency between publishers and subscribers. This capability allows publishers to generate events independetly of subscriber availability. Similarly, subscribers can receive notifications of events, even those that occured during their disconnected phases.

Synchronization decoupling, within pub/sub systems, means that publishers are not blocked while producing events and allows subscribers to asynchronously receive notifications, often through callbacks, while engaged in other tasks. This decoupling separates the event generation and consumption processes from the primary control flow of publishers and subscribers.

3.1 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform that can publish, subscribe to, store, and process streams of records in real time. It is designed to handle data streams from multiple sources and deliver them to multiple consumers. [20] Kakfa is designed to be scalable, fast, reliable and durable. It can handle a large volume of data which enables you to send messages at end-points. Its workflow involves several steps to transmit and process real-time data: Real-time data are generated by various networked sources such as sensors, devices, or software applications, and transmitted over a computer network. The real-time data is then published to a message broker, which acts as a central hub that receives and distributes data to multiple subscribers in real-time. [22]

The main characteristics of Kafka are:

1. Persistent messaging: the messages are not lost, at least until they are read.
2. High throughput: Kafka support millions of messages per second.
3. Distributed: Kafka support distributing consumption over a cluster of consumer machines while maintaining ordering semantics.

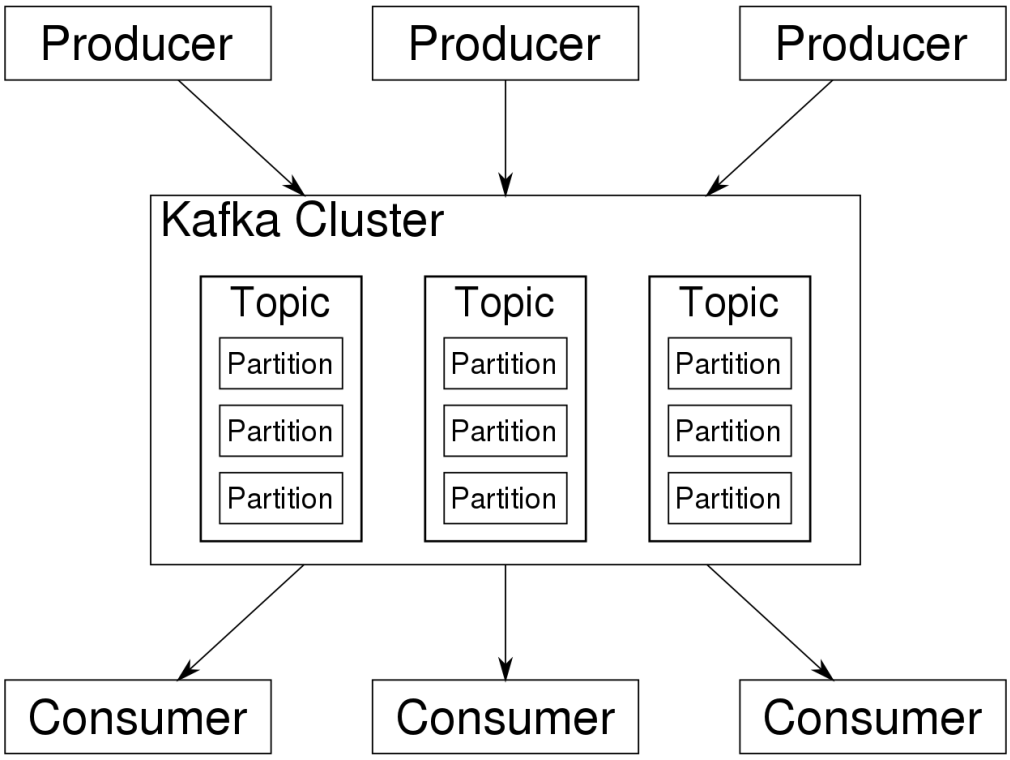

Figure 3.2: Apache Kafka Architecture

4. Multiple client support: Kafka supports the integration of clients from different platforms.

5. Real Time: Messages produced by the threads should be immediately visible to consumers.

Kafka is a distributed system consisting of servers and clients that communicate via a high performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called brokers. For mission critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

Kafka clients allow to write distributed applications and microservices that

read, write and process streams of events in parallel, at scale, and in a fault tolerant manner even in the case of network problems or machine failures.

3.1.1 Events

Events are one of the most important elements in Kafka. An event records the fact that "something happened". When data is written or read, this is done in form of events in Kafka. [10]

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing

An event has a key, value, timestamp and optional metadata headers. An example of event could be:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp "Jun. 25, 2020 at 2:06 p.m."

3.1.2 Topics

In every messaging system based on pub/sub communication, the topic plays one of the most important role, providing the possibility to decouple consumers and producers.

Events, or messages, are organized and durably stored in topics. Topics in Kafka are always multi-producer and multi-subscriber. Events in a topic can be read as often as needed, unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly feasible.

3.1.3 Partitions

Topics are partitioned, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of the data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions.

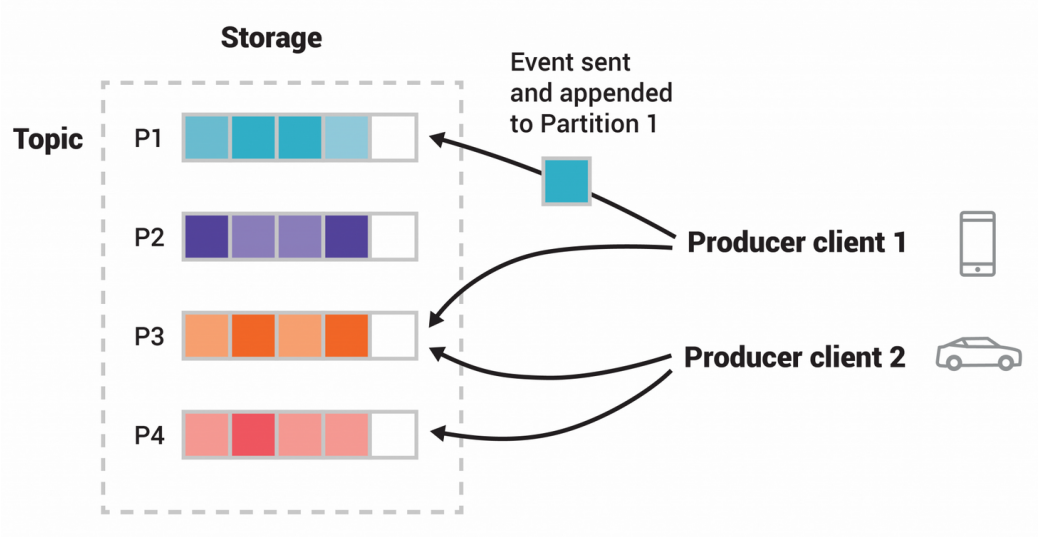

Figure 3.3: Kafka Partitions

Everything in Kafka is modeled around partitions. They rule Kafka's storage, scalability, replication and message movement.

Kafka's topics are divided into several partitions. While the topic is a logical concept in Kafka, a partition is the smallest storage unit that holds a subset of records owned by a topic. Each partition is a single log file where records are written to it in an append-only fashion.

Offsets

The records in the partitions are each assigned a sequential identifier called the offset, which is unique for each record within the partition. The offset is an incremental and immutable number, maintained by Kafka. When a record

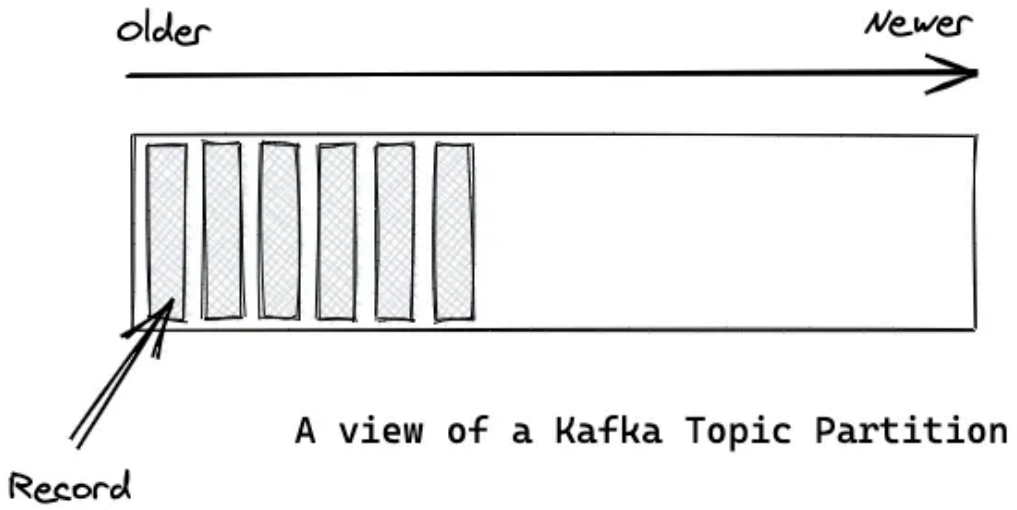

Figure 3.4: Structure of a partition

is written to a partition, it is appended to the end of the log, assigning the next sequential offset.

Offsets are used by consumers when reading records from a partition.

Although messages within a partition are ordered, messages across a topic are not guaranteed to be ordered. For every consumer, Kafka only needs to maintain the information about the latest offset.

Distribution of Partitions

A Kafka cluster is made of one or more servers. In the Kafka universe, they are called brokers. Each broker holds a subset of records that belongs to the entire cluster. Kafka distributes the partitions of a particular topic across multiple brokers.

To make your data fault-tolerant and highly available, every topic can be replicated, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, maintenance on the brokers has to be done, and so on. Replication is done at the partition level.

Advantages:

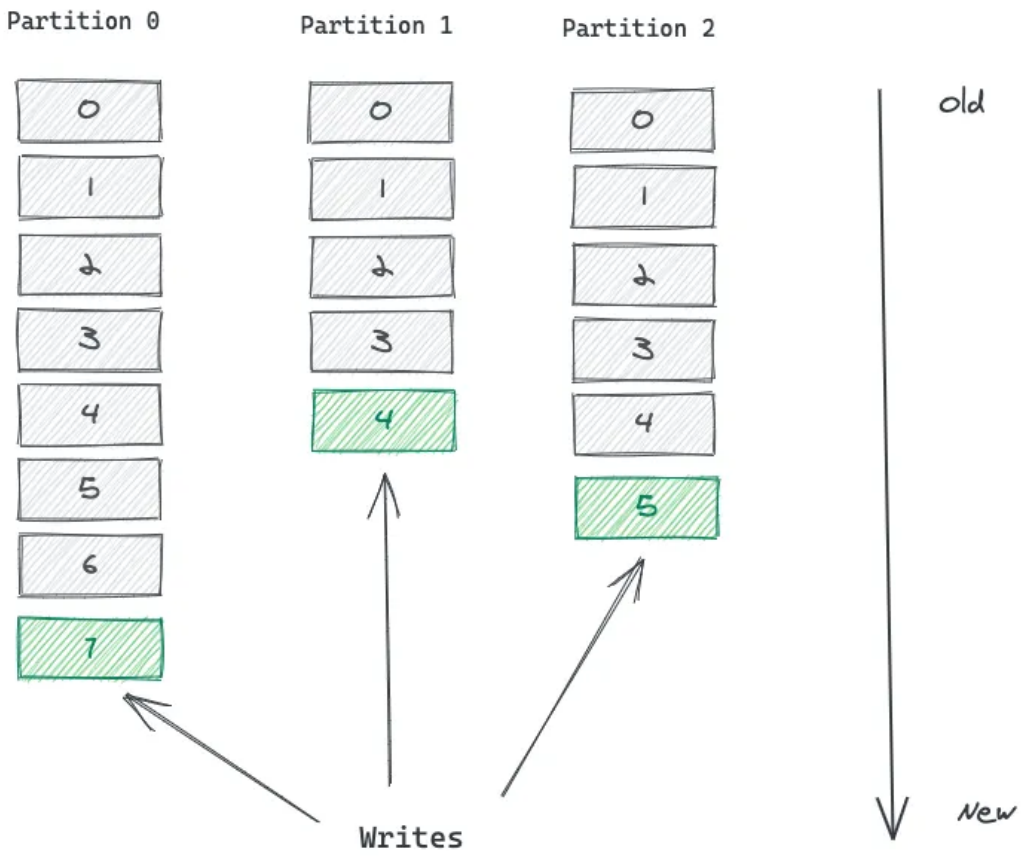

Figure 3.5: Partitions with the offset for every record

- By spreading partitions across multiple brokers, a single topic can be scaled horizontally to provide performance far beyond a single broker's ability.
- A single topic can be consumed by multiple consumers in parallel.
- Serving all partitions from a single broker limits the number of consumers it can support. Partitions on multiple brokers enable more consumers.
- Mutliple instances of the same consumer can connect to partitions on different brokers, allowing very high message processing throuhgput.

The copies of the partitions are called replicas. If a broker fails, Kafka can still serve consumers with the replicas of partitions that failed broker owned.

Kafka replicate partitions so if a broker goes down, a backup partition takes over and processing can resume. The replication factor is configurable (e.g., configurable factor of three creates three copies of a partition, one leader and two followers).

All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log; all have the same offsets and messages in the same order. Although at any given time, the leader may have a few unreplciated messages at the end of its log.

Followers consume messages from the leader like a Kafka consumer would and apply them to their own log. Followers pulling from the leader enables the follower to batch log entries applied to their log.

3.1.4 Producers

Producers in Apache Kafka serve as data publishers, sending messages or events to topics within the Kafka cluster. The producer is *thread safe* and sharing a single producer instance across threads will generally achieve better performance than having multiple instances.

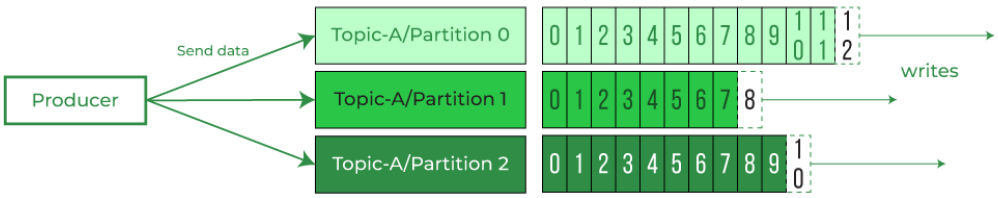

Figure 3.6: Example of Producer publishing data

The producer consists of a pool of buffer space that holds records that have not yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster.

The **send()** method is asynchronous. When called it adds the record to a buffer of pending record and returns immediately. This allows the producer to batch together individual records for efficiency.

The producer maintains buffers of unsent records for each partition. The batch size is configurable, to satisfy the need of every application. Making the batch size larger can result in an improvement in efficiency, but it requires more memory.

Fault Tolerance

The producers in Kafka will automatically know to which broker and partition to write based on the message and in case there is a Kafka broker failure in the Kafka cluster, the producers will automatically recover from it, which makes Kafka resilient and highly adaptable.

Load Balancing

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this, all Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriately direct its requests.

The producer controls which partition it publishes messages to. This can be done at random, implementing a random load balancing, or it can be done by a semantic partitioning function. This second approach allows the developer to define a partition key and use it together with an hashing function to decide to which partition publish the message (e.g., if the key chosen was a user id, then all data for a given user would be send to the same partition).

Messages written to the partition leader are not immediately readable by consumers regardless of the producer's acknowledgement settings. When all in-sync replicas have acknowledged the write, then the message is considered committed, which makes it available for reading. This ensure taht messages cannot be lost by a broker failure after they have already been read. This implies that messages which were acknowledged by the leader only can be lost if the partition leader fails before the replicas have copied the message. Nevertheless, this is often a reasonable compromise in practice to ensure durability in most cases while not impacting throughput too significantly.

Asynchronous Send

Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound. This allows the accumulation of more bytes to send, and few larger I/O opeartions on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

3.1.5 Consumers

Consumers are vital components that subscribe to specific topics within Kafka cluster to retrieve and process messages or events. They play a key role in data consumption, allowing applications to access and utilize the information published by producers, facilitating real-time data processing and analysis. They work by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer has significant control over this position and can rewind it to re-consume data if needed.

Push vs Pull

In this respect, Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. A push based system has difficulties dealing with different consumers as the broker controls the rate at which data is transferred.

A pull based system has the nicer property that if the consumer falls behind in processing messages, it can catch up at any time. Another advantage of a pull-based system s that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency, this will result in sending a single message

at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the consumer always pulls all available messages after its current position in the log. Optimal batching is achieved without introducing unnecessary latency.

Consumer Position (Offset)

Keeping track of *what* has been consumed is one of the key performance points of a messaging system.

A topic is divided into a set of totally ordered partitions, each of which is consumed by exactly one consumer within each subcribing consumer group at any given time. This means that the position of a consumer in each partition is just a signle integer: the offset of the next message to consume. This makes the state about what has been consumed very samll, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgments very cheap.

There is a side benefit of this decision. A consumer can deliberately rewind back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers.

Detecting Consumer Failure

After subscribing to a set of topics, the consumer will automatically join the group when *poll(Duration)* is invoked. The poll API is designed to ensure consumer liveness. As long as you continue to call poll, the consumer will stay in the group and continue to receive messages from the partitions it was assigned. Underneat the surface, the consumer sends periodic hearbeats to the server. If the consumer creashes or is unable to send heartbeats for a configured duration, then the consumer will be considered dead and its partitions will be reassigned.

3.1.6 Brokers

Kafka cluster consists of one or more servers, called brokers, which execute Kafka. Using a single broker for an application is possible, but it does not

allow to exploit all the services that Kafka offers, like data replication.

3.1.7 Writing Records to partitions

There are three different ways that a producer can write a record to a partition:

- Partition key
- Kafka decides the partition
- Writing to a customer partition

Partition key

A producer can use a partition key to direct messages to a specific partition. A partition key can be any value that can be derived from the application context (e.g., device ID, user ID).

By default, the partition key is passed through a hashing function, which creates the partition assignment. That assures that all records produced with the same key will be located in the same partition. Specifying a partition key enables keeping related events together in the same partition and in the exact order in which they were sent.

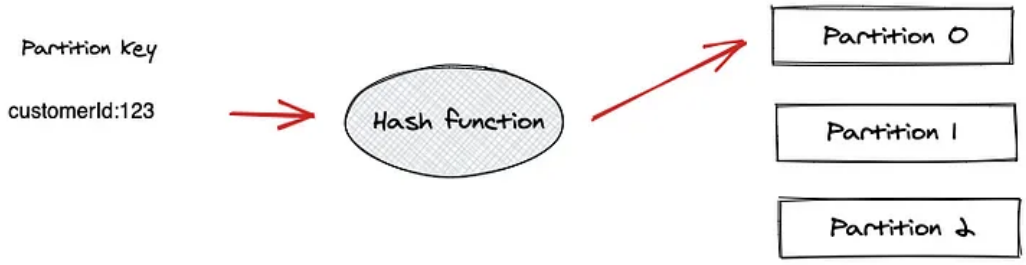

Figure 3.7: Distribution of messages based on partition key

The data distribution of the partition key can lead to broker skew if keys are not well distributed. It is important to use a partition key to put related events together in the same partition in the exact order in which they were sent.

Kafka decide the partition

If a producer does not specify a partition key when producing a record, Kafka will use a round-robin partition assignment. Those records will be written evenly, across all partitions of a particular topic. However, if no partition key is used, the ordering of records can not be guaranteed within a given partition.

Writing a customer partition

In some situations, a producer can use its own partitioner implementation that uses othre business rules to do the partition assignment. This guarantees the maximum customization, satisfying all the needs an application can have.

3.1.8 Reading records from a partition

Unlike the other pub/sub implementations, Kafka does not push messages to consumers. Instead, consumers have to pull messages off Kafka topic partitions. A consumer connects to a partition in a broker, reads the messages in the order they were written.

The offset of a message works as a consumer side cursor at this point. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. After reading a message, the consumer advances its cursor to the next offset in the partition and continues. Advancing and remembering the last rewad offset within a partition is the responsability of the consumer.

By remembering the offset of the last consumed message for each parititon, a consumer cna join a partition at the point in time they choose and resume from there. That is particularly useful for a consumer to resume reading after recoverting from a crash.

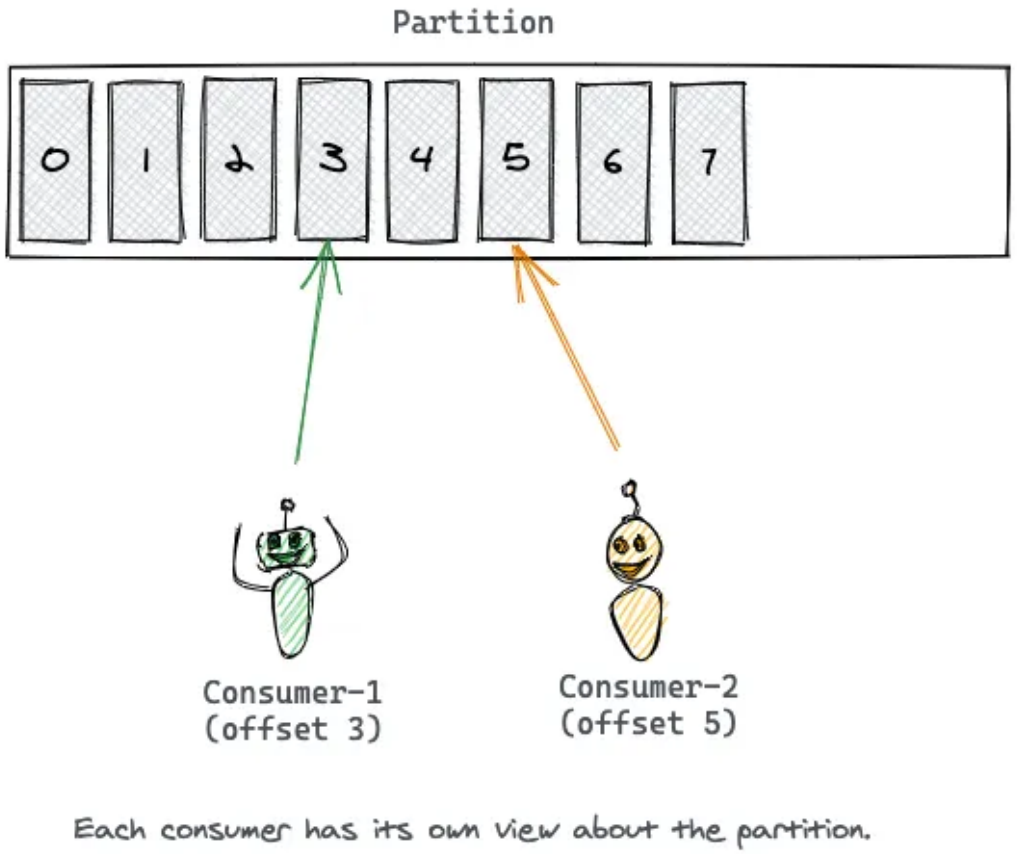

Figure 3.8: Example of reading a partition from two consumers

3.1.9 Consumer Groups

Kafka has the concept of consumer groups where several consumers are grouped to consume a given topic. Consumers in the same consumer group are assigned the same group-ID value.

The consumer group concept ensures that a message is only ever read by a single consumer in the group.

Multiple consumer groups are useful when different applications need to read the same content. Each consumer would have different offset pointer to keep information about the latest read record.

When a consumer group consumes the partition of a topic, Kafka makes sure that each partition is consumed by exactly one consumer in the group.

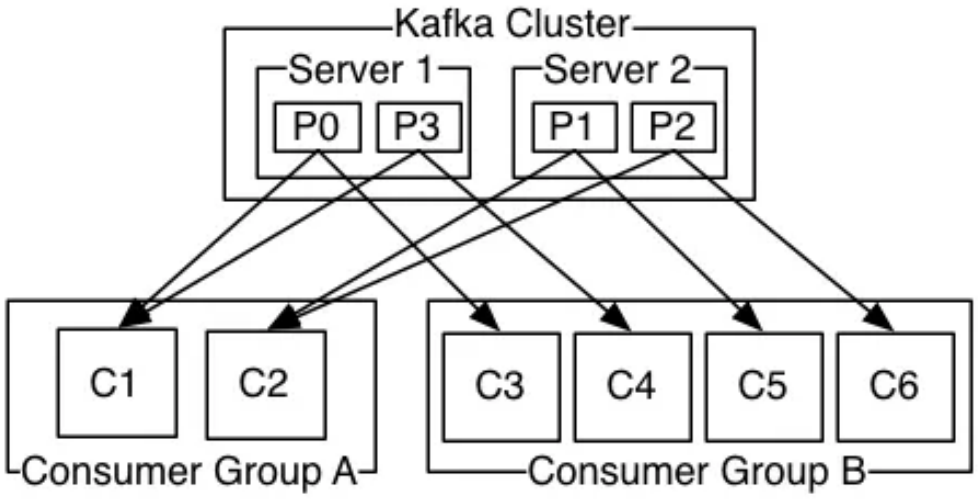

Figure 3.9: Consumer groups reading from the Kafka cluster

Consumer groups enalb econsumer to parallelize and process messages at very high throughputs. However, the maximum parallelism of a group will be equal to the number of partitions of that topic.

The number of consumers do not govern the degree of parallelism of a topic. It is the number of partitions that governs the degree of parallelism.

3.1.10 Message Delivery Semnatics

In Kafka, message semantics betewen producers and consumers offer varying level of assurance:

- *At most once*: Message may be lost, but never redelivered.
- *At least once*: Messages are never lost byt may be redelivered.
- *Exactly once*: Messages are delivered once and only once.

When a message is produced in Kafka, ensuring these semantics involves committing to the log. Once a messages is committed, it will not be lost as long as one broker replicates the partition where it is written. Different use

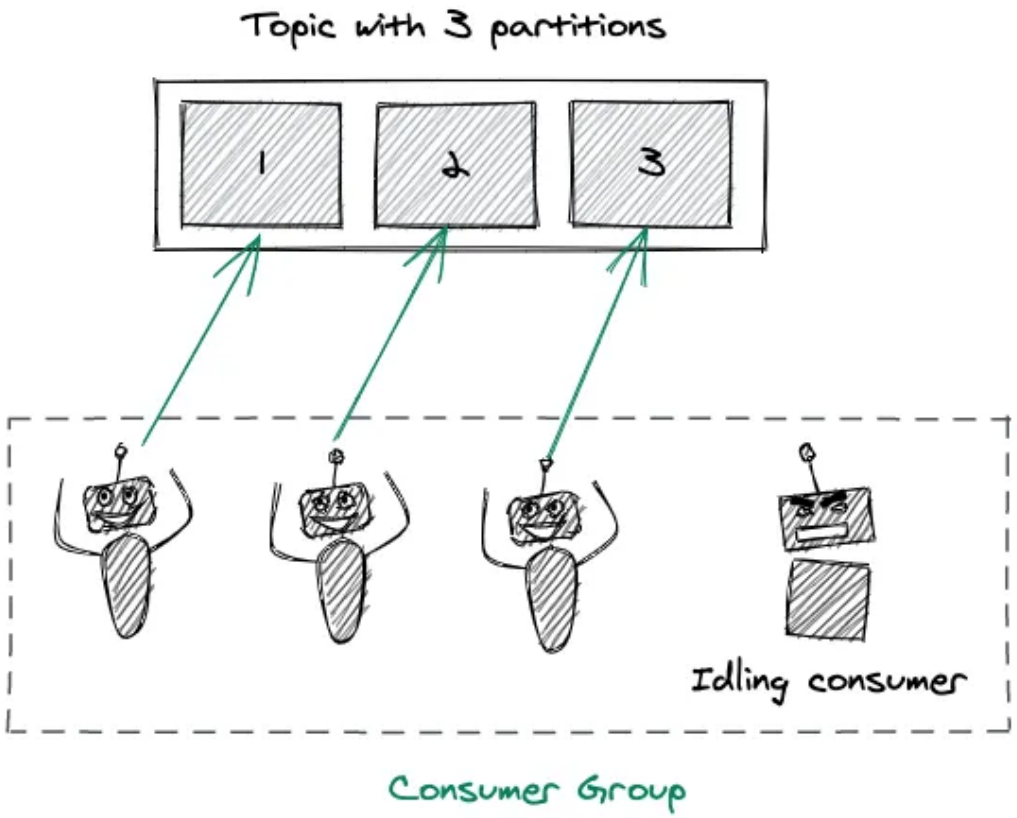

Figure 3.10: Kafka degree of parallelism

cases have different needs of semantic guarantees. Kafka allows specifying the desired level of durability, but stronger semantics can increase communication time, which might not suit latency-sensitive applications.

From the consumer's perspective, achieving these semantics involves managing the position in the log. All replicas maintain identical logs with matching offsets. The consumer determines its position in this log, which it can store in memory or directly on the broker to prevent data loss in case of a consumer crash. Updating the position offers two main approaches:

1. Read, save position, then process: the consumer reads messages, saves its position, and processes the messages. However, the consumer crashes after saving its position but before completing message processing, those messages might never br processed.

2. Read, process, then save position: the consumer reads messages, pro-

cesses them, and then saves its position. Yet, if the consumer crashes after processing but before updating its position, it risks reprocessing messages.

Chapter 4

Design and Architecture of EmbPub

In the context of distributed systems, middleware for pub/sub communication plays a fundamental role, as discussed in the previous chapter. This thesis starts with the idea that embedded systems, particularly those with heterogeneous processors, can be perceived as a form of distributed system.

While substantial progress has been made in the domain of pub/sub platforms for traditional distributed systems, the adaptation of such platforms to satisfy the needs of embedded systems is relatively unexplored territory. This presents an opportune starting point leveraging existing knowledge while navigating the unique challenges inherent in embedded systems.

As seen in the initial chapters, embedded systems have different requisites from traditional systems. Solutions in this domain must take into consideration low latency requirements, safety considerations, and other specific needs for the specific application.

The primary goal of this project is to realize a pub/sub platform that exploits the advantages offered by pub/sub communication while satisfying the needs of embedded systems.

In this chapter are illustrated in detail the design of the pub/sub platform created illustrating the architectural choices.

4.1 Architecture

The proposed pub/sub middleware starts with a layer architecture, carefully structured to address the unique challenges posed by embedded systems. This section illustrates the key components residing in each layer, providing information about their roles, interactions, and contributions to the overall efficacy of the communication framework.

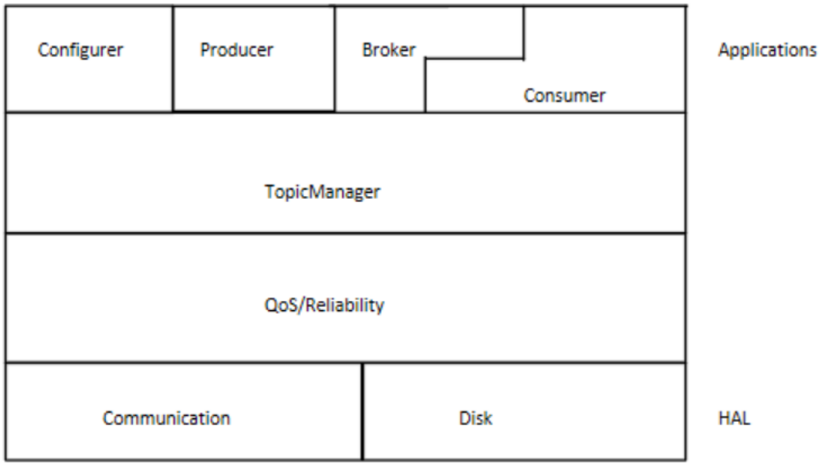

Figure 4.1: Layered architecture of the solution

At the top of the architecture resides the Application layer, the interface through which end-users and system components interact with the pub/sub middleware. The Application layer provides a user-friendly interface for configuring, consuming and producing messages within the ecosystem. This layer is further subdivided into essential elements:

- Configurer: Responsible for configuring and maintain the information about the other components of the system.
- Consumer: Facilitates the consumption of messages and the interactions needed for retrieving messages.
- Broker: Serves as intermediary between producers and consumers, efficiently routing messages based on established topic subscription.
- Producer: initiates the generation of messages into the communication network.

Under the Application layer, the Topic Manager layer serves as the manager of all topic-related operations. The layer manages the creation, deletion and tracks topics within the system. The seamless coordination of topics is one of the most important element, making the Topic Manager one of the most important element to provide flexible and dynamic communication infrastructure.

Coming down from the most abstract layers, the Quality of Service (QoS) layer introduces elements designed to enhance the middleware's reliability and safety. Within this layer, mechanisms for topic replication and the possibility to define custom logic for the distribution of messages between a group of consumers can be found.

Situated at the bottom of the stack there are the communication and disk components, which is defined as the Operating System (OS) layer. The Disk layer introduces a component for persistent data storage; classes dedicated to saving records on disk have great importance to ensure data durability. This layer is particularly important when records have to be preserved beyond runtime.

The other component at the base of the architecture is the Communication layer. It plays a fundamental role for facilitating interaction among distributed components within embedded systems. Leveraging the APIs provided by Texas Instruments, this layer ensures robust and efficient communication between different processors.

4.2 Class interactions

The image (figure 4.2) details the different components that enable the exchange of messages between producers and consumers in a distributed environment.

The architecture is composed of various entities, including a cluster, consumer groups, brokers, replicas, partitions, records, producers and consumers.

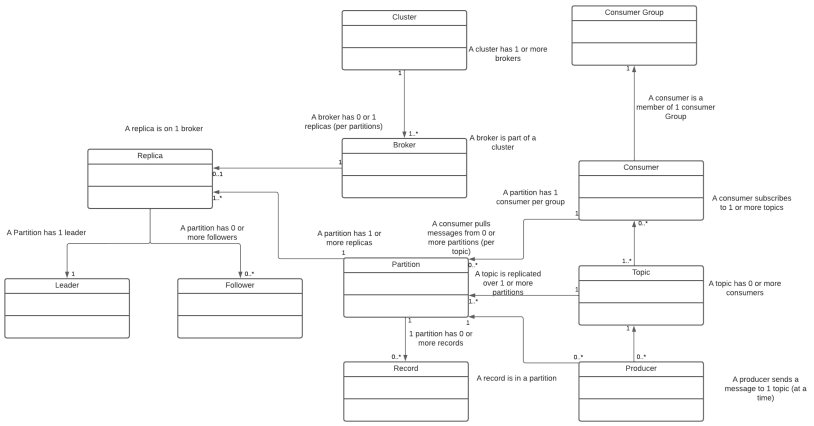

Figure 4.2: Interaction between classes

The most interesting components are the following ones:

- Cluster: the cluster is the foundation of the middleware, acting as a collection of brokers that facilitate message exchange and compose the general system.
- Consumer groups: Consumer groups organize consumers into logical units, allowing them to divide and conquer the workload of consuming messages from a topic. The approach enhances parallelism and scalability.
- Replicas: replicas are copies of partitions stored on different brokers, providing redundancy and fault tolerance. If the primary broker of a partition fails, a replica can take over its role, ensuring uninterrupted message delivery.

These ideas are taken from Apache kafka and applied to the embedded system domain.

4.3 Class Diagrams

In the current section the class diagrams for the different layers will be analyzed.

4.3.1 Topic Manager

The Topic Manager layer is responsible for managing the topic metadata and handling interactions between producers, consumers and brokers. It is a crucial component of the middleware, ensuring efficient message exchange and ensuring the integrity of the messaging system.

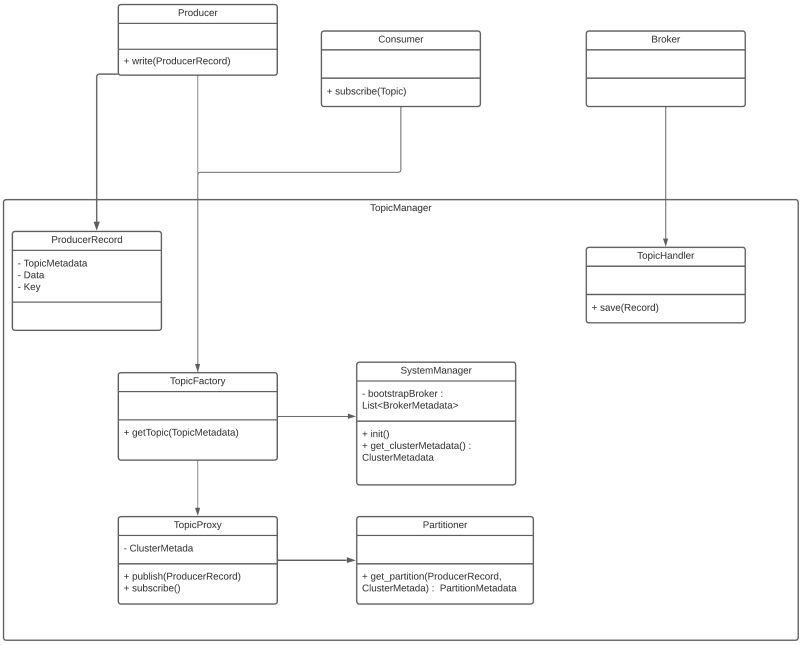

Figure 4.3: Class diagram of the topic manager layer

The topic manager layer interacts with the broker component to maintain the state of topics, including the creation, deletion and modification. It also manages subscriptions, ensuring that consumers are notified of messages published to topics they are subscribed to. Additionally, the topic manager layer handles message routing, directing messages to the appropriate partitions within each topic.

The topic manager layer is implemented as a collection of components that work together to provide the necessary functionality. These components include:

1. TopicFactory: this component is responsible for creating new topics and managing their metadata. It ensures that topics are created with

the appropriate configurations and that they are registered with the broker.

2. TopicHandler: This component handles interactions between producers and consumers. It receives messages from producers, adds them to the appropriate partitions, and updates topic metadata. It also receives subscription requests from consumers and manages subscriptions accordingly.

3. SystemManager: this component is of fundamental importance for all the system operations. It provides a centralized interface for managing the communication with the configurer. It also provides all the information about the current status of the cluster.

4.3.2 QoS

The QoS layer is responsible for managing the quality of services (QoS) of the system.

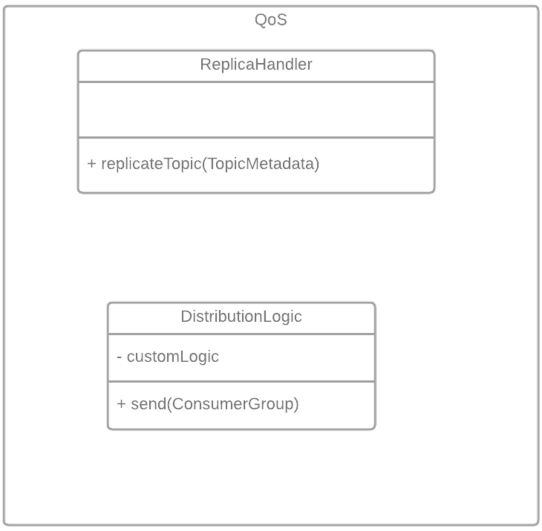

Figure 4.4: Class diagram of the QoS layer

The components are:

- ReplicaHandler: this component is used for replicating topics to ensure high availability and durability of data. It takes a TopicMetadata object as a parameter, which contains information about the topic to be replicated. The ReplicaHandler then replicates the topic to a specified number of brokers.
- DistributionLogic: this component is responsible for distributing messages to consumer groups in a way that meets the QoS requirements of the application. It takes a ConsumerGroup object as a paramter, which contains information about the consumers belonging to the group that has to be serviced. The DistributionLogic then distributes the messages to the consumer group in a way that ensures that all messages are eventually delivered to at least one consumer.

The QoS layer has the necessity to interact with other elements of the system in order to work. In particular, the complexity of the above layer, the topic manager, gives the possibility to simplify this layer, which will exploits the classes given by the communication layer to perform the exchange of messages in a simple way.

4.3.3 Communication

The communication layer is responsible for sending and receiving messages between the processors in the system and outside of the embedded system. The layer is one of the most important ones in the domain of a platform for pub/sub communication. This layer takes into consideration the necessity for embedded system of using restrained resources.

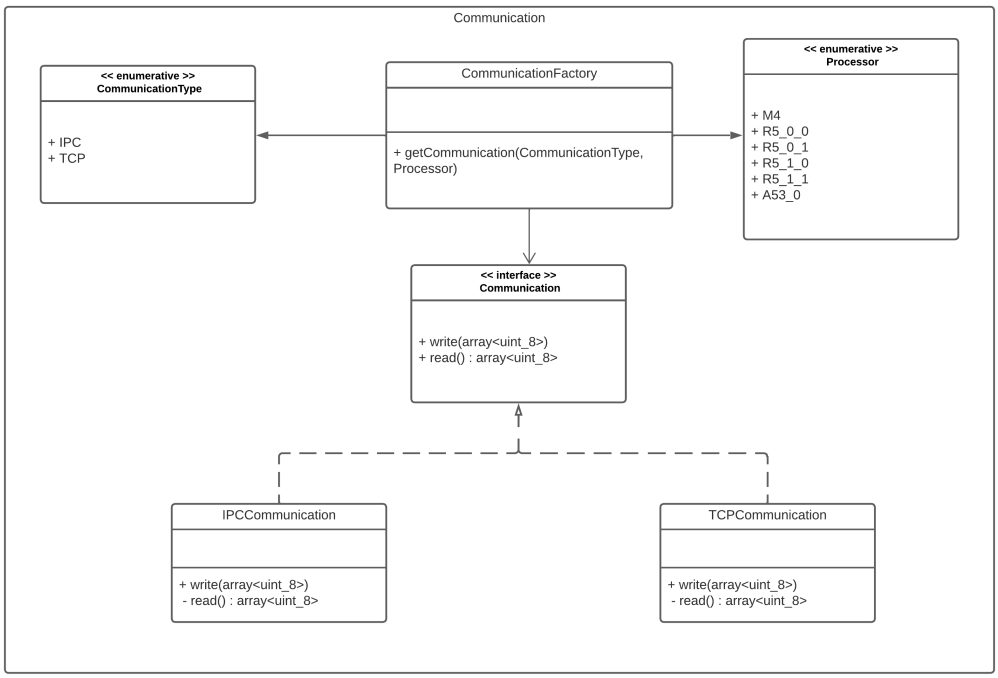

Figure 4.5: Class diagram of the communication layer

The components seen in the class diagram (figure 4.5) are:

- CommunicationFactory: this class is responsible for creating instances of IPCCommunication or TCPCommunication objects. It takes a CommunicationType parameter.
- CommunicationType: this enumeration defines the two types of communication that can be created: IPC or TCP.
- IPCCommunication: This class is used for communication between processors on the same device. It provides methods for sending and receiving messages.
- TCPCommunication: this class is used for communication between processors on different devices. it provides methods for sending and receiving messages over a TCP socket.

4.3.4 Domain level

The image below (figures 4.6 and 4.7) shows all the classes that compose the domain of the system. The domain classes can be used by all the layer of the application (primarily from the top level layers) and they contain information of the specific class and the interaction with the other elements.

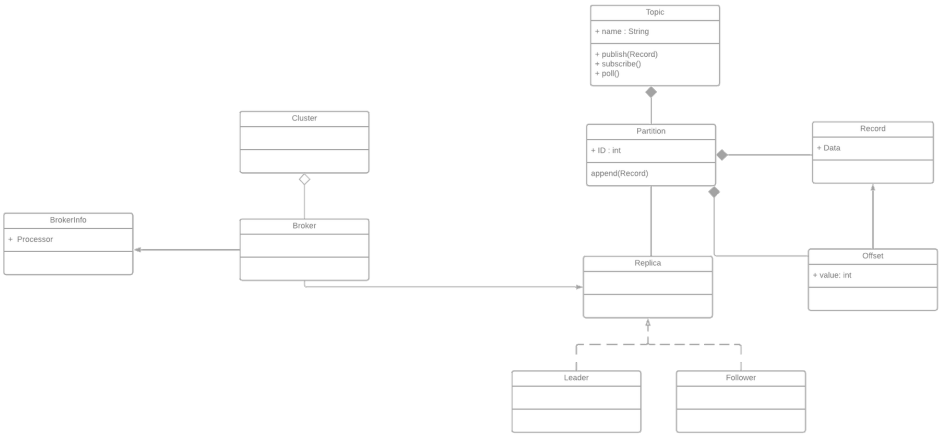

Figure 4.6: Class diagram of domain of the middleware 1

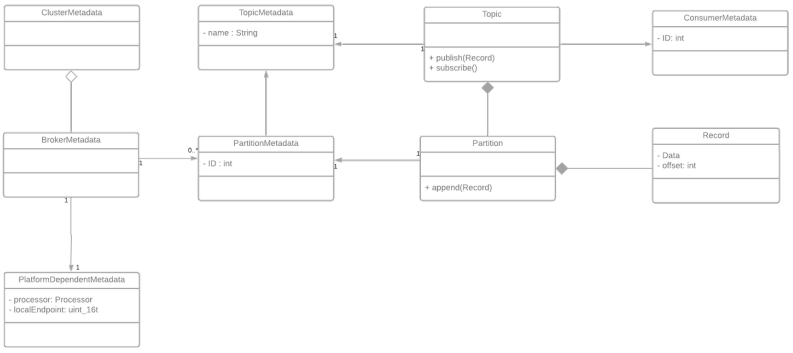

Figure 4.7: Class diagram of domain of the middleware 2

4.4 Sequence Diagram

The following section will use sequence diagrams to visualize the steps needed and the classes interaction to perform the most important operations of the system.

4.4.1 Consumer asking for a record

The sequence diagram depicts the interaction between a message consumer and the rest of the system. The consumer initiates the interaction by requesting a TopicProxy from the TopicFactory. The pattern proxy helps to hide all the operations that will be done underneath to perform the communication between the different processors, making it seem like a local communication. The consumer then establishes a communication channel with the broker containing the interested topic using the CommunicationFactory. The SystemManager is used to retrieve the information about the cluster status.

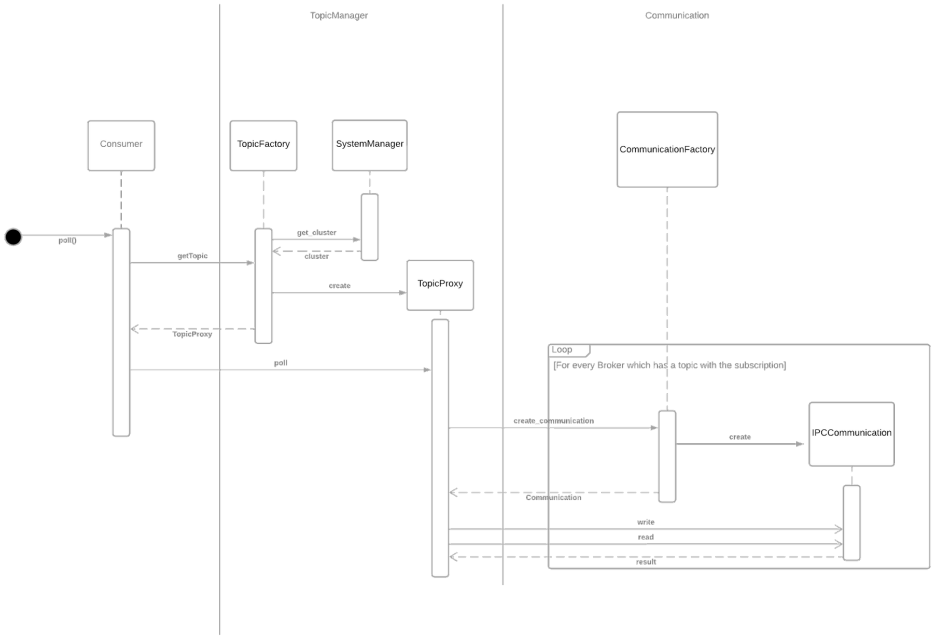

Figure 4.8: Sequence diagram of a consumer asking for a record

4.4.2 Publisher sending a record

The publishing of a message is handled in a similar way to the consumer read. All the details is hidden under the TopicProxy and the SystemManager.

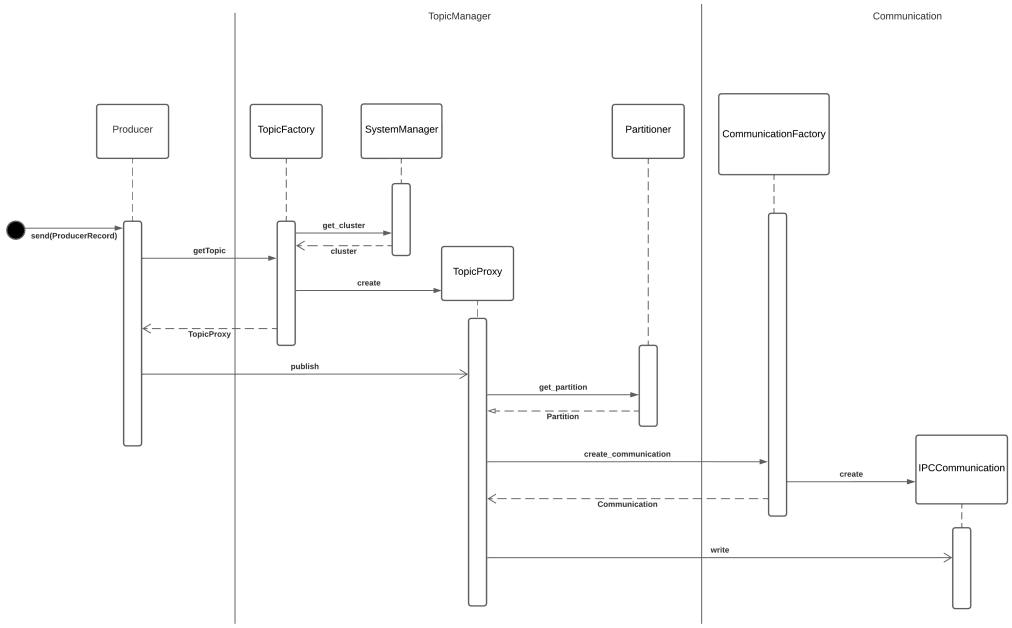

Figure 4.9: Sequence diagram of a producer publishing a record

4.4.3 Synchronization of the cluster status

The synchronization of the cluster status is a fundamental operation in a distributed system. The operation is divided in two steps:

1. The first one is performed by the system manager to ask to the centralized "server" (configurer) about the information.
2. The second one is performed by the configurer which responds to the request sent by the system manager.

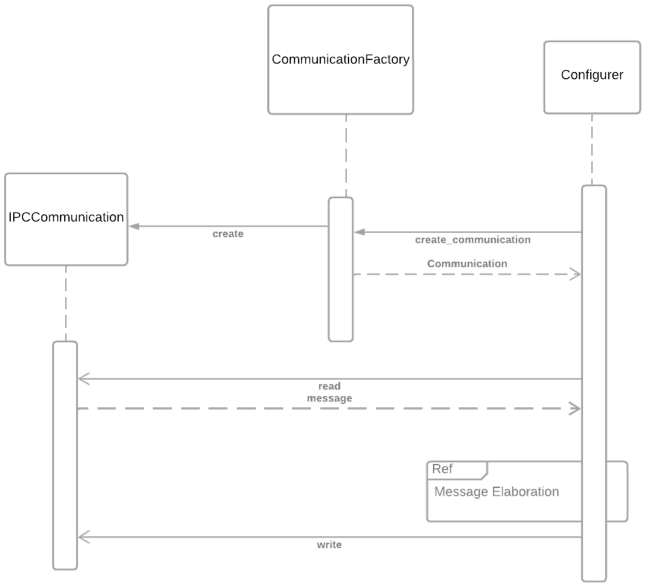

Figure 4.10: Sequence diagram of the configurer operations done for the cluster status synchronization

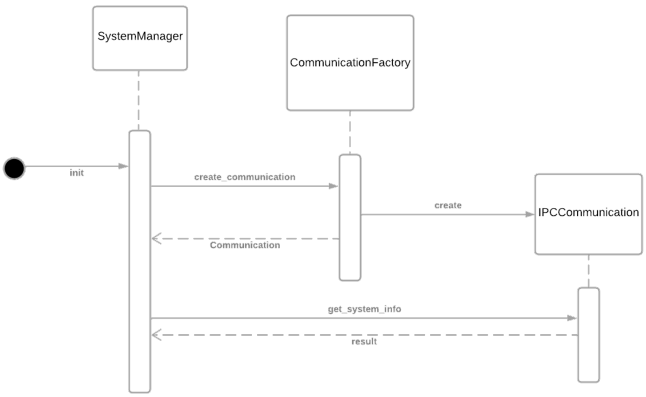

Figure 4.11: Sequence diagram of the system manager operations done for the cluster status synchronization

4.4.4 Broker receiving a record

One of the most important operation is the possibility to store the records, sent by the producers, in a broker.

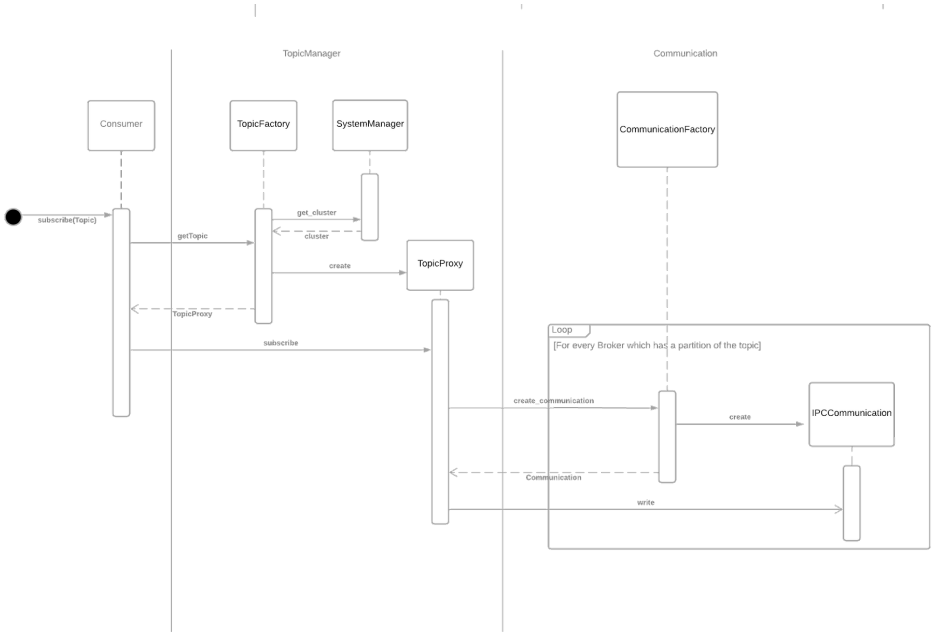

Figure 4.12: Sequence diagram of a broker receiving a record to store

4.4.5 Consumer subscribing to a topic

The subscription process involves a consumer expressing its interest in a specific topic, initiating a series of interaction with the underlying system. The Consumer class uses the usual TopicProxy to communicate with the broker holding the needed topic.

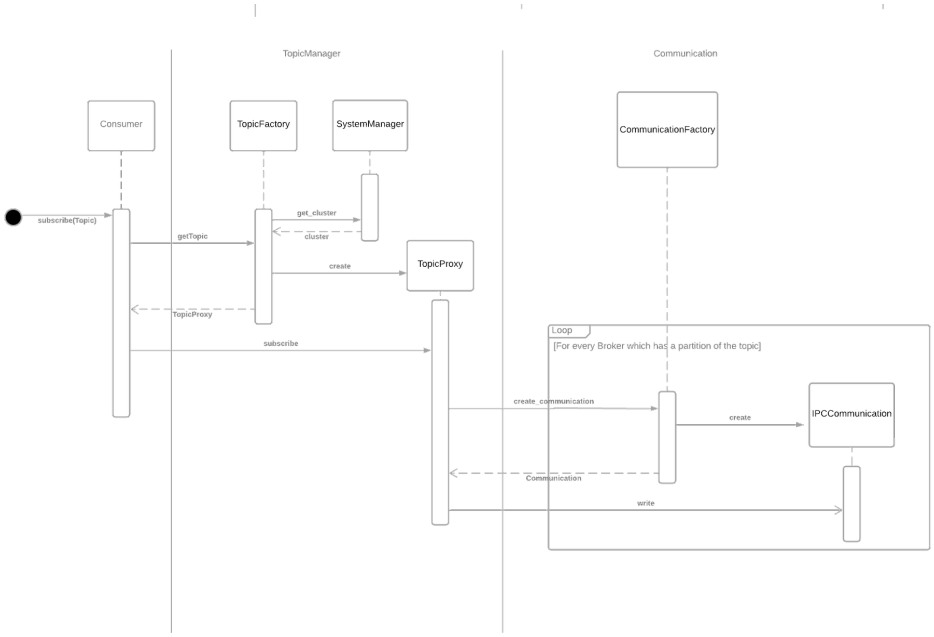

Figure 4.13: Sequence diagram of a consumer subscribing to a topic handled by a broker

Chapter 5

Implementation and Experimental Results

In the concluding chapter of the thesis, the implementation details surrounding important aspects of the system will be analyzed. Also a comprehensive examination of the experimental results obtained from testing the system in different situations will be performed.

5.1 Message format

In the context of embedded systems, the design choices for message serialization and deserialization play a crucial role in the overall efficiency and performance of the communication system. The structure of the Message and the accompanying SerializeMessage and DeserializeMessage methods demonstrate an attempt to balance the need for a comprehensive representation of information with the constraints of the underlying communication infrastructure.

The importance of having a low overhead in the message packets becomes particularly evident when considering the limitations imposed by the RPMessage library, which enforces a maximum message size of 1024 bytes.

```
struct Message {
    std::string operation;
    ClientMetadata clientMetadata;
    Record record;
    TopicMetadata topicMetadata;
};
```

Figure 5.1: Message Structure

The SerializeMessage function strives to achieve this optimization by concatenating the essential fields of the Message structure into a single string. This representation minimizes unnecessary padding or additional metadata that might contribute to increased packet size. Furthermore, the use of simple delimiters, such as commas, helps reduce the overhead associated with encoding and decoding.

```
std::string SerializeMessage(const Message& message) {
    std::string serializedMessage = message.operation + ",";
    serializedMessage += std::to_string(message.clientMetadata.getId()) + ",";
    serializedMessage += message.record.getData() + ",";
    serializedMessage += message.topicMetadata.getName();
    return serializedMessage;
}
```

Figure 5.2: Message Serialization

Additionally, the DeserializeMessage function efficiently extracts information from the serialized string, ensuring that the deserialization process does not introduce unnecessary complexities.

```
Message DeserializeMessage(const std::string& serializedMessage) {
    Message message;
    std::string copy = serializedMessage;

    size_t pos = copy.find(",");
    if (pos != std::string::npos) {
        message.operation = copy.substr(0, pos);
        copy.erase(0, pos + 1);
    }

    pos = copy.find(",");
    if (pos != std::string::npos) {
        message.clientMetadata.setId(std::stoi(copy.substr(0, pos)));
        copy.erase(0, pos + 1);
    }

    pos = copy.find(",");
    if (pos != std::string::npos) {
        message.record.setData( copy.substr(0, pos));
        copy.erase(0, pos + 1);
    }

    message.topicMetadata.setName(copy);

    return message;
}
```

Figure 5.3: Message Deserialization

5.2 Cluster information distribution

The process of disseminating information about the position of brokers in the system, as well as creating a vector containing details about topics, involves the collaboration of two classes: SystemManager and TopicFactory. This workflow is orchestrated to ensure that consumers and producers can access relevant information about the cluster configuration efficiently.

The SystemManager follows these steps:

1. Initialization (SystemManager::init()): The SystemManager class is responsible for initiating the system. In this process, it starts by constructing a JSON request using the nlohmann/json library, specifying the operation as ”getClusterInformation”.

2. Request to Configurer (CommunicationUtils::request): The constructed JSON request is then sent to a configurer, through a communication utility (CommunicationUtils::request).

3. Parsing Response (json::parse): The response is then parsed using the nlohmann/json library, converting it into a JSON object. This object contains information about the cluster configuration, such as broker metadata and topic metadata.

```
void SystemManager::init()
{
    requestJson["operation"] = "getClusterInformation";
    std::string requestString = requestJson.dump();

    char response[1024];
    CommunicationUtils::request(communication, communicationType, bootstrapBroker.getEndpoint(),
                                requestString.c_str(), requestString.size(), response, sizeof(response));
    logger.log("[System Manager] Received response: %s", response);

    json j = json::parse(response);
    clusterMetadata.from_json(j);
}
```

Figure 5.4: SystemManager requesting the state of the cluster to the configurer

The TopicFactoy is directly used by consumers and producers to retrieve the information.

1. Creation of Topics (TopicFactory::createTopics()): The TopicFactory class utilizes the cluster information obtained by the SystemManager to create topics. It iterates through the broker metadata in the cluster, then for each broker, it iterates through the topics associated with that broker.

2. TopicProxy Instantiation: For each topic, a TopicProxy object is instantiated using the communicationType, BrokerMetadata, TopicMetadata, and a logger. The TopicProxy encapsulates functionality related to communication with the specified broker for the given topic.

```
void TopicFactory::createTopics()
{
    for (BrokerMetadata brokerMetadata : clusterMetadata.getBrokersMetadata())
    {
        for (TopicMetadata topicMetadata : brokerMetadata.getTopicsMetadata())
        {
            TopicProxy *topicProxy = new TopicProxy(communicationType, brokerMetadata, topicMetadata, logger);
            logger.log("%s", topicProxy->getTopicMetadata().getName().c_str());
            topics.push_back(topicProxy);
        }
    }
}
```

Figure 5.5: TopicFactory creating a vector of Topics

5.3 Broker mode

In pub/sub systems, having both push and pull modes is essential for serving different application scenarios. Push mode enables immediate data dissemination to all subscribed consumers as soon as a record is received by the broker, allowing for real-time updates and synchronous interactions. On the other hand, pull mode allows consumers to retrieve data at their own pace, providing them with the autonomy to fetch information when they are ready.

```
void TopicHandler::save(Record record, TopicMetadata topicMetadata, ProducerMetadata producerMetadata)
{
    int topicIndex = findTopicIndex(topicMetadata);

    if (topicIndex != -1)
    {
        if(push) {
            for (const auto &pair : topics[topicIndex].getConsumers())
            {
                const ConsumerMetadata cons = pair.first;
                cons.getEndpoint()->printEndpointInformation(logger);
                json recordJSON;
                record.to_json(recordJSON);
                communication->write(recordJSON.dump().c_str(), recordJSON.dump().size() + 1, *cons.getEndpoint());
            }
        }
        else {
            topics[topicIndex].publish(producerMetadata, record);
        }
    }
}
```

Figure 5.6: Broker mode

The TopicHandler::save method plays a fundamental role in realizing this dual-mode communication strategy. Upon receiving a record, the method dynamically adapts its behavior based on the operational mode set for the broker. In the push mode scenario, the method iterates through the consumers subscribed to the relevant topic, sending the record individually to each consumer after serialization.

In pull mode, the method delegates the received record to the publish method of the corresponding topic. This approach respects the asynchronous nature of pull-based communication, allowing consumers to retrieve records at their convenience.

5.4 Communiation

In this section, the fundamental communication infrastructure will be explored. RPMessageCommunication and RPMessageLinuxCommunication are the classes on which most of the interactions are done in the system. RPMessageCommunication specializes in real-time communication between cores, ensuring low-latency data transfer. Meanwhile, RPMessageLinuxCommunication extends this capability into the Linux embedded environment, making sure communication remains robust through RPMessage char devices. Together, these components establish the basis for effective communication across the different components.

5.4.1 RPMessage

The following implementation encapsulates the communication mechanisms using RPMessage for real-time cores within the RPMessageCommunication class. This class extends the more general Communication class and relies on the RPMessage library to facilitate seamless interaction between cores.

```
RPMessageCommunication::RPMessageCommunication(const RPMessageEndpoint &ep, const Logger &l) : Communication(l), endpoint(ep)
{
    endpoint.printEndpointInformation(logger);
    int32_t status;
    RPMessage_CreateParams createParams;

    RPMessage_CreateParams_init(&createParams);
    createParams.localEndPt = endpoint.getServiceEndpoint();
    status = RPMessage_construct(&msgObject, &createParams);
}
```

Figure 5.7: Initialization of RPMessage

This constructor initializes a communication instance, setting the stage for subsequent operations. It configures the RPMessageCommunication object with a specific RPMessageEndpoint, creating a communication channel using the RPMessage library.

```
int RPMessageCommunication::write(const char *message, size_t messageSize, const Endpoint &destination)
{
    const RPMessageEndpoint &rpMessageDestination = static_cast<const RPMessageEndpoint &>(destination);

    uint32_t status;
    uint16_t msgSize;

    msgSize = static_cast<uint16_t>(messageSize);

    status = RPMessage_send(
        const_cast<char *>(message), msgSize,
        rpMessageDestination.getCoreId(),
        rpMessageDestination.getServiceEndpoint(),
        RPMessage_getLocalEndPt(&msgObject),
    );

    return msgSize;
}
```

Figure 5.8: Write method of RPMessage

Responsible for transmitting messages, the write method employs the RPMessage_send function to dispatch messages to a specified destination. It extracts core ID and service endpoint details from the provided RPMessageEndpoint. The return value signifies the size of the transmitted message.

```
int RPMessageCommunication::read(char *buffer, size_t bufferSize, Endpoint &source)
{
    uint32_t status;
    uint16_t remoteCoreId, remoteCoreServiceEndpoint;
    uint16_t size = static_cast<uint16_t>(bufferSize);

    status = RPMessage_recv(&msgObject,
                            buffer, &size,
                            &remoteCoreId,
                            &remoteCoreServiceEndpoint
    );

    buffer[size] = 0;

    static_cast<RPMessageEndpoint &>(source).setCoreId(remoteCoreId);
    static_cast<RPMessageEndpoint &>(source).setServiceEndpoint(remoteCoreServiceEndpoint);

    return size;
}
```

Figure 5.9: Read method of RPMessage

Handling message reception, the read method utilizes the RPMessage_recv function to retrieve messages from the communication channel. It extracts

essential details such as the size of the received message, the remote core ID, and the service endpoint.

5.4.2 RPMessage for Linux embedded

The write method within the RPMessageLinuxCommunication class facilitates the transmission of messages from the current endpoint to a specified destination in a Linux embedded environment using the RPMessage library. Exploiting the RPMessage char device model, the method dynamically generates a unique device name based on the destination's core ID and the current process ID. It then checks if the destination is already present in the endpointMap. If not, it opens a new RPMessage char device, updates the endpointMap, and includes the associated file descriptor in the list of monitored file descriptors. Subsequently, the method utilizes the send_msg function to transmit the message via the established communication channel, returning the result of the transmission. This implementation ensures efficient and reliable communication between RPMessage endpoints in the Linux embedded system.

```
int RPMessageLinuxCommunication::write(const char *message, size_t messageSize, const Endpoint &destination)
{
    rpmsg_char_dev_t *rcdev;
    const RPMessageEndpoint &rpMessageDestination = static_cast<const RPMessageEndpoint &>(destination);
    char eptdev_name[32] = {0};
    int flags = 0, ret = 0;
    int fd;

    sprintf(eptdev_name, "rpmsg-char-%d-%d", rpMessageDestination.getCoreId(), getpid());

    if (endpointMap.find(rpMessageDestination) == endpointMap.end())
    {
        rcdev = rpmsg_char_open(static_cast<rproc_id>(rpMessageDestination.getCoreId()), NULL, RPMSG_ADDR_ANY, rpMessageDestination.getServiceEndpoint(),
                                eptdev_name, flags);
        endpointMap.insert(std::make_pair(rpMessageDestination, rcdev));
        fds.push_back(rcdev->fd);
    }

    rcdev = endpointMap[rpMessageDestination];

    ret = send_msg(rcdev->fd, message, messageSize);

    return ret;
}
```

Figure 5.10: Write method of RPMessage for Linux embedded

The read method in the RPMessageLinuxCommunication class manages the reception of messages in a Linux embedded environment using the RPMessage library. Employing the select function, the method efficiently monitors a set of file descriptors for incoming data. Upon detecting input availability, it identifies the file descriptor with pending data, updates the source Endpoint accordingly using the setSourceEndpoint function, and reads the message

content via the recv_msg function. This implementation ensures a responsive mechanism for message retrieval from RPMessage endpoints within the Linux embedded system, balancing efficiency and real-time considerations.

```
int RPMessageLinuxCommunication::read(char *buffer, size_t bufferSize, Endpoint &source)
{
    int packet_len;
    int maxfd = *std::max_element(fds.begin(), fds.end());

    fd_set read_fds;
    FD_ZERO(&read_fds);

    for (const auto &fd : fds)
    {
        FD_SET(fd, &read_fds);
    }

    struct timeval timeout;
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    int num_ready = select(maxfd + 1, &read_fds, nullptr, nullptr, NULL);
    if (num_ready == -1)
    {
        logger.log("Error in select");
    }
    else if (num_ready == 0)
    {
        logger.log("Timeout occured");
    }
    else
    {
        logger.log("Input is available");
        for (const auto &fd : fds)
        {
            if (FD_ISSET(fd, &read_fds))
            {
                setSourceEndpoint(fd, source);
                int bytesRead = recv_msg(fd, 512, buffer, &packet_len);
                logger.log("I have read %d bytes", packet_len);
                if (packet_len > 0)
                {
                    buffer[packet_len] = '\0';
                    logger.log("Read message: %s, from fd %d", buffer, fd);
                }
                return bytesRead;
            }
        }
    }
    return 0;
}
```

Figure 5.11: Read method of RPMessage for Linux embedded

5.5 Results

The last phase of the project involved the development and testing the performance of the implemented system. The focus of the testing was on measuring

the time required for message exchanges among different system elements.

Various tests were conducted, with the nature of each test contingent on the processors involved in the communication. The specific test performed include:

1. Time needed for the direct communication between real time cores and the Linux core.
2. Time needed for the exchange of messages using the implemented system varying in the processor used and in the number of consumers and producers.

Throughout all tests, the size of the exchanged messages was varied to understand the system performance under different conditions.

5.5.1 Direct Communication

The direct communication has been tested by using the Communication layer realized previously. Specifically, communication between different processors relied on the RemoteProcessorMessage (RPMsg) function provided by Texas Instruments. This API proves to be a robust solution for facilitating efficient communication in a multi-core environemnt. RPMsg excels in establishing direct communication channels between real-time cores and other cores within a heterogeneous system. Leveraging this API ensures that the communication channels are not only reliable but also characterized by low latency.

Within the RPMessage library, each packet is equipped with a 16 byte header, shaping the structure of the communication. Consequently, the actual message payload size is constrained by MAX_MSG_DIMENSION - 16 bytes.

RTOS Cores

Given the importance of real-time operations in embedded systems, the initial set of tests is about the direct communication of this kind of processors. The processors which are taken into examination are the Cortex-R5F and the Cortex-M4F.

The round trip time for R5s processors is computed by timing a message exchange between two different R5s. The round trip time for M4 processor is computed by timing a message exchange between the M4 core and one of the R5s.

Table 5.1: Round trip time results using RP Message between RTOS cores

Message Size (in bytes)	32	64	128	256	512	1024
RTT R5 (in us)	16.7	21.9	32.9	54.9	98.9	186.8
RTT M4 (in us)	35.7	48.2	73.2	125.7	228.1	435.7

With an increase in message dimension, a corresponding rise in Round Trip Time (RTT) is anticipated, as indicated by the observed results. However, an examination of table 5.1 reveals a non-linear relationship between message dimension and RTT. The RTT demonstrates a slower growth rate as the message dimension increases.

Linux and RTOS cores

The results presented in Table 5.2 are derived from message exchanges between real-time cores and the Cortex-A53 processor running Linux. These tests hold particular significance as the more potent Cortex-A53 processor serves as a centralized access point for various services. It is imperative to ensure that the communication latency remains minimal.

In the Liux IPC library communication the maximum dimension for messages is 512 bytes.

Table 5.2: Round trip time results using RP Message between Linux and RTOS cores

Message Size (in bytes)	32	64	128	256	512
Linux and R5 (in us)	50	61	77	108	168
Linux and M4 (in us)	64	80	110	177	306

5.5.2 Broker Communication

In this section there are the measurements obtained by using the pub/sub middleware. There are two modes which are tested:

- Push: after that a producer sends a message to a broker, the broker sends it directly to all the subscribed consumers.
- Pull: the consumers have to ask for the message to be delivered.

RTOS cores

Following the idea behind the tests done for the direct communication, also in the broker communication the first processors which are tested are real-time processors (Cortex-R5F and Cortex-M4F).

Push

When there is only one consumer and one producer in push mode the time needed for the consumer and the producer are similar. This result is expected given that the consumer is only expecting to receive messages.

Table 5.3: Round trip time results using the middleware between RTOS cores in push mode

Message Size (in bytes)	32	64	128	256	512	1024
Producer R5 Time (in us)	73.9	90.4	121.7	182.7	317.4	501.1
Producer M4 Time (in us)	99.7	111.1	137.6	190.7	322.4	506.9
Consumer R5 Time (in us)	76.9	92.9	122.9	182.9	317.6	503.2
Consumer M4 Time (in us)	74.9	92.1	120.6	182.8	315.2	570.3

As it can be seen in the table 5.3, the time for the consumer on the M4 is the same for the consumer on the R5 core, since no messages have to be sent from the consumer when the communication is in push mode.

Table 5.4: Round trip time results using the middleware between RTOS cores with multiple consumers in push mode

Message Size (in bytes)	32	64	128	256	512	1024
Producer Time (in us)	277.2	300.9	358.9	478.2	744.9	1110.1
Consumer 1 Time (in us)	282.2	306.1	364.1	483.2	750.1	1115.1
Consumer 2 Time (in us)	763.4	765.4	804.1	991.4	1279.1	1503.4
Consumer 3 Time (in us)	539.4	510.1	634.6	754.2	1063.6	1283.4

The variations in time measurements among different consumers, which can be seen in 5.4 can be attributed to the system's booting sequence. During the boot process, the initialization of memory areas can differ, consequently leading to variations in the prioritization of message delivery. It becomes apparent that the order in which processes are booted significantly influences the subsequent behavior of the system, resulting in different timing outcomes across consumers.

Pull

Pull communication necessitates a higher volume of message exchanges as consumers explicitly request records to be sent to them. Despite this increased messaging overhead, pull communication offers several notable advantages. Notably, it provides the flexibility for consumers to process messages only when the core is ready, eliminating the need for message buffering when the consumer is unable to process the message at a given time. This on-demand processing capability enhances efficiency and resource utilization. Additionally, pull communication simplifies the implementation of Quality of Service (QoS) features, allowing for more precise control and customization of message delivery based on the dynamic requirements of the system. These advantages collectively contribute to the appeal and practicality of pull communication in scenarios where fine-grained control over data retrieval and processing is important.

Table 5.5: Round trip time results using the middleware between RTOS cores in pull mode

Message Size (in bytes)	32	64	128	256	512	1024
Producer R5 Time (in us)	93.1	104.1	121.2	152.4	212.7	301.8
Producer M4 Time (in us)	171.1	182.4	191.5	262.4	391.9	579.2
Consumer R5 Time (in us)	228.6	238.9	276.1	325.4	439.9	618.9
Consumer M4 Time (in us)	372.3	411.5	490.4	589.9	843.8	1236.7

5.5.3 Linux on Cortex-A53

Finally, the RTT for messages using the middleware in an heterogeneous environment is tested, adding the Cortex-A53 with Linux to the real-time cores.

Push

Table 5.6: Round trip time results using the middleware between Linux and RTOS cores in push mode

Message Size (in bytes)	32	64	128	256	512
Producer Time(in us)	98	127	191	264	393
Consumer Time (in us)	97	127	191	264	393

Pull

Table 5.7: Round trip time results using the middleware between Linux and RTOS cores in pull mode

Message Size (in bytes)	32	64	128	256	512
Producer Time(in us)	148	171	227	231	367
Consumer Time (in us)	264	292	322	406	565

5.6 Result Analysis and future extensions

The observed results in the experimental phase of the thesis reveal distinct patterns in the performance of the implemented pub/sub platform for embedded systems.

In the context of direct communication, the lower round-trip times (RTT) between real-time cores compared to those involving Linux cores underscore the efficiency of direct communication channels, particularly leveraging the RemoteProcessorMessage (RPMsg) framework. However, the middleware introduces additional services, contributing to a higher RTT. The intricacies of the Linux operating system and its IPC library, while providing a centralized access point for services, result in comparatively higher communication latencies.

Notably, the push mode in broker communication demonstrates symmetrical RTT for single consumers and producers, as expected, while variations in multiple consumer scenarios highlight the system's booting sequence impact. In pull mode, the flexibility and on-demand processing capability contribute to advantages despite increased messaging overhead. Overall, the results reflect a trade-off between direct communication efficiency and the richer feature set provided by the middleware, while Linux's higher RTT emphasizes the complexities introduced by the operating system.

For the end of the thesis, it is important to outline potential directions for future investigations

First and foremost, exploring the seamless integration of the pub/sub middleware with emerging edge computing paradigms represents a crucial direction. Investigating how the middleware can enhance communication efficiency and resource utilization in edge computing scenarios holds the potential to address the unique challenges presented by distributed computing at the network's periphery.

Additionally, security and reliability enhancements are essential, with a focus on lightweight cryptographic protocols and fault-tolerant mechanisms to fortify communication channels and ensure system robustness.

Furthermore, the integration of artificial intelligence techniques with the pub/sub middleware offers an exciting frontier. Enabling intelligent data filtering, prediction, and adaptation in response to dynamic changes in the

embedded environment can significantly elevate the platform's adaptability and responsiveness.

Finally, the introduction of new protocols, potentially at an industrial level, can have a big impact in the automation industry. Such protocols can be customized to meet specific industry requirements, ensuring compatibility with diverse embedded applications and expanding the platform's utility in various domains.

Conclusion

In conclusion, the primary objective of creating an effective communication framework between heterogeneous processors in embedded systems has been achieved with satisfying results. The thesis has presented a comprehensive exploration of the design, implementation, and performance analysis of a pub/sub middleware for embedded systems.

The evaluation of the pub/sub middleware has revealed highly promising results, affirming its potential advantages and practical viability. Notably, the middleware exhibited superior performance in resource-constrained environments, demonstrating capabilities such as impressive scalability and adaptability to diverse workloads. Its low-latency characteristics make it particularly well-suited for applications requiring real-time responsiveness.

Moreover, the middleware showcased robustness in dynamic environments, effectively adapting to changes in network topology and device dynamics. The evaluation emphasized the system's efficiency in handling real-time data, optimizing bandwidth usage, and maintaining a minimal memory footprint, essential features for resource-limited embedded environments. The observed advantages underscore its reliability and suitability for a wide range of embedded applications.

In addition to its performance attributes, the middleware's design lends itself to enhancing safety measures through the duplication of brokers responsible for data storage. This redundancy introduces a layer of fault tolerance, minimizing the risk of data loss or system failures. The dynamism of this approach ensures that the system can adapt to changes in the environment, providing a robust and fail-safe mechanism in safety-critical applications.

Looking ahead, potential future directions include integration with edge computing paradigms, enhancements in security and reliability mechanisms, ex-

ploration of artificial intelligence integration, and the introduction of new protocols.

Bibliography

[1] Real time linux. https://www.embedded.com/real-time-linux/, 2001.

[2] Arm m4 specifications. https://developer.arm.com/Processors/Cortex-M4, 2014.

[3] Embedded linux ubuntu. https://ubuntu.com/blog/what-is-embedded-linux, 2019.

[4] Am64x technical reference manual. https://www.ti.com/lit/ug/spruim2h/spruim2h.pdf, 2020.

[5] Embedded linux windriver. https://www.windriver.com/solutions/learning/embedded-linux, 2020.

[6] Cortex-m4 technial reference manual. https://developer.arm.com/documentation/ddi0439/b/Introduction/Product-documentation/Documentation, 2021.

[7] Tightly coupled memories documentation. https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory, 2021.

[8] Am64x datasheet. https://www.ti.com/lit/ds/symlink/am6412.pdf, 2023.

[9] Am64x sitara™ processors. https://www.ti.com/product/AM6442, 2023.

[10] Apache kafka documentation. https://kafka.apache.org/documentation/, 2023.

[11] freertos site. https://www.freertos.org/about-RTOS.html, 2023.

[12] Gnu make site. https://www.gnu.org/software/make/, 2023.

[13] Rockchip rk3xx series. `https://www.rock-chips.com/a/en/News/Press_Releases/index-89-9.html`, 2023.

[14] Rz/g2 mpus. `https://www.solid-run.com/embedded-industrial-iot/renesas-rz-family/`, 2023.

[15] X2xx series. `http://www.ingenic.com.cn/`, 2023.

[16] Yocto project. `https://www.yoctoproject.org/`, 2023.

[17] Zynq ultrascale+ mpsocs. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`, 2023.

[18] Doug Abbott. *Linux for embedded and real-time applications*. Elsevier, 2011.

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, jun 2003.

[20] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[21] Daniele Lacamera. *Embedded systems architecture*. 2023.

[22] Theofanis P Raptis and Andrea Passarella. A survey on networked data streaming with apache kafka. *IEEE access*, 2023.

[23] Otavio Salvador and Daiane Angolini. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014.

[24] Sasu Tarkoma. *Publish/subscribe systems: design and principle*. 2012.