PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

JEAN CARLO HAMERSKI

SUPPORT TO RUN-TIME ADAPTATION BY A PUBLISH-SUBSCRIBE BASED MIDDLEWARE FOR MPSOC ARCHITECTURES

Porto Alegre
2019

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
GRADUATE PROGRAM IN COMPUTER SCIENCE

SUPPORT TO RUN-TIME ADAPTATION BY A PUBLISH-SUBSCRIBE BASED MIDDLEWARE FOR MPSOC ARCHITECTURES

JEAN CARLO HAMERSKI

Thesis presented as partial requirement for obtaining the degree of Ph. D. in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Alexandre de Morais Amory

Porto Alegre
2019

Ficha Catalográfica

H214s Hamerski, Jean Carlo

Support to run-time adaptation by a publish-subscribe based middleware for MPSoC architectures / Jean Carlo Hamerski . – 2019.

157.

Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Alexandre de Morais Amory.

1. MPSoC. 2. many-core systems. 3. programming model. 4. communication model. 5. self-adaptive systems. I. Amory, Alexandre de Morais. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Jean Carlo Hamerski

Support to run-time adaptation by a publish-subscribe based middleware for MPSoC architectures

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 21th, 2019.

COMMITTEE MEMBERS:

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS)

Profa. Dra. Débora da Silva Motta Matos (UERGS)

Prof. Dr. Mateus Beck Rutzig (UFSM)

Prof. Dr. Alexandre de Morais Amory - Advisor (PPGCC/PUCRS)

AGRADECIMENTOS

O caminho percorrido durante o doutorado é longo e cheio de incertezas. É com o apoio dos amigos, familiares e professores que seguimos adiante. Por isso, quero agradecer a todos que de alguma forma tiveram parte na finalização do meu doutorado.

Primeiramente quero agradecer aos meus pais, que desde cedo me incentivaram nos estudos. Se chego à conclusão do doutorado, é porque lá no início da minha formação escolar tive a percepção enraizada de que é através do estudo que abrimos os horizontes. Um agradecimento especial a minha mãe Arlete, que, quando professora, me possibilitou desde muito cedo viver o ambiente escolar e seu potencial de transformação de vidas.

Ao meu orientador, Alexandre Amory, obrigado pelo guiamento nos momentos de dúvida e suporte intelectual nos caminhos tomados durante a pesquisa do doutorado.

Aos demais professores do programa de pós-graduação, pelas aulas, dicas e apoio em eventuais reuniões, em especial aos professores Fernando Moraes e César Marcon.

Aos amigos que fiz durante o doutorado, obrigado pelos momentos de descontração e pelo apoio nas atividades de pesquisa. Obrigado Caimi, Gustavo, André, Jurinha, Fochi, Ruaro, Anderson, Felipe, Ost, Geancarlo e todos os demais amigos do sagrado futebol semanal.

Ao Instituto Federal do Rio Grande do Sul e aos programas de fomento à pesquisa e capacitação de professores pelo necessário suporte e auxílio.

Por fim quero agradecer a minha esposa que foi uma incrível parceira nesses 4 anos. Paula, teu espaço de escuta, carinho e sempre sábias palavras foram indispensáveis. E também a minha filha, Isabela, que com sua alegria foi uma válvula de espace aos momentos de tensão. Amo vocês!

SUPORTE À ADAPTAÇÃO EM TEMPO DE EXECUÇÃO ATRAVÉS DE UM MIDDLEWARE BASEADO EM PUBLISH-SUBSCRIBE PARA ARQUITETURAS MPSOC

RESUMO

As aplicações embarcadas têm migrado de sistemas baseados em um único processador para uma comunicação de dados intensiva que exige sistemas multiprocessados. O desempenho exigido por estas aplicações motivam o uso de arquiteturas multiprocessadas em um único chip (MPSoCs). Mudanças em tempo de execução na qualidade do serviço prestada pela plataforma MPSoC para as aplicações motivam a implementação de plataformas MPSoCs auto-adaptativas. As plataformas MPSoCs auto-adaptativas empregam sistemas compostos por arquiteturas ricas em sensores-atuadores que observam as mudanças no ambiente de execução e adaptam o sistema balanceando dinamicamente múltiplos objetivos em vários níveis de arquitetura. Esses sistemas auto-adaptativos requerem modelos de comunicação/programação bem adaptados à característica distribuída do ambiente para coordenar a comunicação entre os elementos que o compõem. Esta Tese investiga os atuais modelos de programação/comunicação em MPSoC e outros domínios correlatos com relação ao acoplamento entre os elementos comunicantes e a infraestrutura de hardware e software adjacente. A hipótese levantada é que é necessário usar um modelo que, além de abstrair a complexidade da comunicação, também forneça um acoplamento mais flexível entre os elementos comunicantes do sistema auto-adaptativo. Adicionalmente, nós argumentamos que as abordagens atualmente utilizadas para incorporar sistemas auto-adaptativos em plataformas MPSoCs seguem uma metodologia de desenvolvimento não sistematizada, o que impacta na qualidade de software relacionada ao reuso de código e manutenabilidade. Sendo assim, esta Tese propõe aplicar o modelo publish-subscribe em uma abordagem de desenvolvimento baseada em middleware visando melhorar a qualidade do software de sistemas auto-adaptativos enquanto minimiza impactos indesejados da abordagem proposta sobre o sistema. A Tese é suportada através de um estudo de caso em que implementamos um sistema auto-adaptativo utilizando a abordagem proposta e comparamos os resultados com um sistema auto-adaptativo base, de acordo com métricas de desempenho, energia e qualidade do software. Os resultados mostram que o modelo empregado em uma abordagem de desenvolvimento baseada em middleware melhorou a qualidade do software do sistema auto-adaptativo de 33% até 47.8%, dependendo da métrica avaliada, com um reduzido overhead em relação à métricas de desempenho (4.5%) e energia (5.9%). Mostramos também que os requisitos para o middleware são adequados para plataformas MPSoCs caracterizadas por restrições de uso de memória.

Palavras-Chave: MPSoC, múltiplos núcleos, modelo de programação, modelo de comunicação, publish-subscribe, middleware, sistemas auto-adaptativos, gerenciamento de recursos.

SUPPORT TO RUN-TIME ADAPTATION BY A PUBLISH-SUBSCRIBE BASED MIDDLEWARE FOR MPSOC ARCHITECTURES

ABSTRACT

Embedded applications have been migrating from single processor-based systems to intensive data communication requiring multi-processing systems. The performance demanded by applications motivate the use of Multi-Processor System-on-Chip (MPSoC) architectures. Run-time changes in the quality of service provided by the MPSoC platform to the applications motivate the implementation of self-adaptive MPSoC platforms. Self-adaptive MPSoC platforms employ systems composed of sensor-actuator rich architectures that observe the changes in the execution environment and adapt the system dynamically balancing the multiple objectives across multiple architecture levels. These self-adaptive systems require communication/programming models well suited to the distributed characteristic of the environment in order to coordinate communication between the elements that compose it. This Thesis investigates current programming/communication models on MPSoC and other correlated domains regarding the coupling between the communicating elements and the adjacent hardware and software infrastructure. The hypothesis raised is that it is necessary to use a model that, besides abstracting the communication complexity, provides a more flexible coupling between the communicating elements of the self-adaptive system. Additionally, we argue that the current approaches used to incorporate self-adaptive systems in MPSoC platforms follow a non-systematic development methodology, which impacts the quality of software related to code reuse and maintainability. Therefore, this Thesis proposes to apply the publish-subscribe model in a middleware-based development approach to perform the communication employed between the elements of a self-adaptive MPSoC platform and to improve software quality of self-adaptive systems while minimizing undesired impacts of the proposed approach on the system. The Thesis is supported through a case study where we implement a self-adaptive system following the proposed approach and compare the results with a baseline self-adaptive system according to performance, energy and software quality metrics. The results show that the proposed model employed on a middleware based development approach has improved the software quality of the self-adaptive system by 33% to 47.8%, depending on the metrics evaluated, with a reduced overhead regarding metrics of performance (4.5%) and energy spent (5.9%). We also show that the requirements for middleware are suitable for MPSoC platforms with memory usage constraints.

Keywords: MPSoC, many-core systems, programming model, communication model, publish-subscribe, middleware, self-adaptive systems, resource management.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AES – Advanced Encryption Standard
API – Application Programming Interface
ASIC – Application Specific Integrated Circuits
CC – Cyclomatic Complexity
CM – Cluster Manager
CPSOC – Cyber Physical System-on-Chip
CPI – Cycles Per Instruction
CPU – Central Processing Unit
CSM – Centralized Shared Memory
DDS – Data Distribution Service
DMA – Direct Memory Access
DMNI – Direct Memory Network Interface
DRE – Distributed Real-Time and Embedded
DSM – Distributed Shared Memory
DSP – Digital Signal Processor
DTW – Dynamic Time Warping
DVFS – Dynamic Voltage and Frequency Scaling
ELOC – Effective Lines of Code
ETL – Embedded Template Library
FC – Function Complexity
FIFO – First In, First Out
FPGA – Field-Programmable Gate Array
GM – Global Manager
HAL – Hardware Abstraction Layer
HSAL – Hardware/Software Abstraction Layer
HW – Hardware
IC – Interface Complexity
IOT – Internet of Things
LM – Local Manager
LOC – Lines of Code
MCAPI – Multicore Communication API
MORM – Multi-Objective Resource Management
MPEG – Moving Picture Experts Group
MPI – Message Passing Interface
MPSOC – Multi-Processor System-on-Chip
MQSOC – Message-Queuing System-on-Chip
MQTT – Message Queuing Telemetry Transport
NOC – Network-on-Chip
ODA – Observe-Decide-Act

OO-MQSOC – Object-Oriented Message-Queuing System-on-Chip
OOP – Object-Oriented Programming
OS – Operating System
PC – Personal Computer
PDSM – Partially Distributed Shared Memory
PE – Processing Element
PSLAYER – Publish-Subscribe Layer
PUB-SUB – Publish-Subscribe
QM – Quality Metric
QOS – Quality of Service
RAM – Random Access Memory
ROS – Robot Operating System
RPC – Remote Procedure Call
RTL – Register Transfer Level
RTOS – Real-Time Operating System
SOC – System-on-Chip
SP – Slave Process
SS – Subsystem
SW – Software
VF – Voltage-Frequency
VHDL – VHSIC Hardware Description Language
VHSIC – Very High Speed Integrated Circuit
WSN – Wireless Sensor Networks
XML – Extensible Markup Language

CONTENTS

1. INTRODUCTION

Current embedded applications have migrated from single processor-based systems to intensive data communication requiring multi-processing systems, composed by multiple processor cores. Modern System-on-Chip (SoC) design shows a clear trend toward integration of multiple processor cores [PRJW10]. A SoC is an integrated circuit with the components necessary to execute embedded applications forming an entirely electronic system. The system may contain memory, Processing Elements (PEs), specialized logic, buses, and other digital functions [JW04].

As a SoC derivative, an Multi-Processor System-on-Chip (MPSoC) contains multiple processing elements and memory components interconnected by complex communication infrastructures, such as multiple buses or Network-on-Chip (NoC). Integrated with hardware elements, a software programmable subsystem aims to cope with the underlying hardware and software design complexity. The software subsystem is usually tailored in a layered design approach. Typically, the interface between the different layers is performed by an Application Programming Interface (API) that abstracts the complexity of underlying levels. Middleware is a software component that links the application layer with the communication infrastructure [PRJW10]. It is the "middle" of an end-to-end transaction between applications or services running in a distributed environment, bringing benefits such as abstraction of coordination details and reducing design costs through software reuse [SR14].

Customizing MPSoC platforms according to application requirements and system constraints can be done statically or dynamically. In a static way, the MPSoC platform is configured at design time exclusively for a particular application, not allowing the execution of other constrained applications. When dynamic, the programming environment must provide the designer with a degree of system's observability and configurability, in order to develop the system with self-adaptability property. Self-adaptability is related to the system ability of adapt itself at run-time based on its state. Observability is related to the system ability of providing run-time observation of possible information through sensors and other elements of the architecture. Configurability is related to the system ability of driving configuration commands to configurable system resources. A monitoring and actuating infrastructure is provided to enable observability and configurability properties for the designer [FDLP11] [DJS16]. Future SoC will be monitor-rich in order to perform the system monitoring, employing a network of interconnected sensors that will span circuit, micro-architecture, and software layers [DJS16]. The monitoring infrastructure is necessary to observe the state of the system in terms of transactions, energy, temperature, network load, faults, etc. [FDLP11]. The actuating infrastructure is used for system management purposes and acting in order to achieve adaptability [FDLP11].

The need for handling demanding applications while reducing the software design complexity of embedded devices fuels the MPSoC revolution [NVC10]. Some challenges to reduce this complexity are: (i) programming models that allow the implementation of software in an efficient manner [PRJW10]; (ii) effective schemes of data distribution, synchronization and control among the system elements [PRJW10]; (iii) self-adaptive systems to deal with run-time changing on environment conditions [NVC10]; (iv) monitoring and management of run-time mechanisms that give support to self-adaptive systems coordination [FDLP11] [DJS16].

Until recently, the programming environments were not concerned with MPSoC dynamic adaptation, since the MPSoC hardware/software architecture was statically configured in order to achieve the application requirements. Run-time changes in the Quality of Service (QoS) provided by the platform to the applications and the need for resilient systems motivate the implementation of self-adaptive systems. Run-time changes in the environment can be caused by a variety of factors, such as the load of applications at run-time [NVC10], and failures occasioned by advances

in fabrication technologies [CNG10]. Self-adaptive systems perform techniques that observe the changes in the environment and adapt the system dynamically balance the multiple objectives across multiple architecture levels [DJS16]. The adaptive techniques typically follow a closed-loop scheme and they are composed of monitors, decision makers, and actuators [Hof13].

This Thesis argues the existence of two fundamental problems related to the design of self-adaptive systems in MPSoC. Firstly, this Thesis argues the need for less coupling between the elements of the self-adaptive system and both the kernel software and the communication infrastructure. A tightly-coupled code makes it difficult to be extensible and portable to other platforms. Additionally, this Thesis argues that the approaches currently used to incorporate self-adaptive systems in MPSoC platforms are designed following a non-systematic development methodology, which decreases the software quality related to code reuse and maintainability aspects. We support our arguments describing a middleware to achieve desired decoupling and proposing a middleware extension to help the development of self-adaptive systems that are implemented over the middleware to incorporate the observability and configurability properties to the system in a systematic way.

To address the fundamental problems, this Thesis investigates current programming/communication models in MPSoC and other correlated domains like Robotic, Wireless Sensor Networks, Internet of Things and Distributed Real-time Embedded Systems. To check existing distributed programming/communication models that could contribute to the design of the middleware, this Thesis classifies these models for the coupling between the communicating elements and the adjacent hardware and software infrastructure. The state of the art shows that most programming/communication models for MPSoC still use heavily coupled approaches for communicating purpose, such as MPI (Message Passing Interface) or ad-hoc[1] protocols. MPI presents coupling in synchronization levels (blocking/not-blocking send/receive), time (communication elements must be active at the same time in the system), and space (a communication element must know who is its communication pair) [EFGK03]. With ad-hoc protocols, the protocol phases are hard-coded to the kernel software, does not offering a systematic way for including new protocol phases or even abstracting platform-specific communication aspects.

The publish-subscribe model has gained interest in performing data exchange in distributed environments. As we discuss further, this model enables complete decoupling in the communication between the elements of the system and the underlying hardware/software architecture. We argue in this Thesis that it is necessary to use a model that, besides abstracting the communication complexity, also provides a more flexible coupling between the communicating elements. Thus, this Thesis proposes to apply the publish-subscribe model in MPSoC environments to perform the communication employed among the elements of a self-adaptive system, facilitating the development of adaptive services which consider multiple sensors and actuators.

In summary, this Thesis proposes **a new development approach consisting of publish-subscribed-based middleware and API for the design of both applications and self-adaptive systems in MPSoC platforms**. As a research methodology, we employ case studies to evaluate the protocol and middleware on two different MPSoC platforms: FreeRTOS-based MPSoC [AMR+16] and HeMPS [MM18]. We provide a self-adaptive system model and middleware extension supporting the development of self-adaptive systems.

[1] Ad-hoc - adjective made up of the Latin word "ad" meaning "towards" and of the demonstrative "hoc" meaning "this": "toward this purpose" or "suitable for a specific use" [Lab10].

1.1. Thesis Statement

Faced with current programming/communication models and the need for self-adaptation of the MPSoC systems that meet the application requirements while complying with system constraints, the use of a publish-subscribe model along with a middleware-based approach can improve the software quality of self-adaptive systems while minimizing the middleware impact on the system performance, memory usage and energy spend.

1.2. Thesis Goal

The strategic goal of this Thesis is the proposition of a middleware to support the run-time adaptation of resources by providing a publish-subscribe protocol, middleware and API for MPSoC architectures.

1.2.1. Thesis Specific Goals

The specific goals of the Thesis are the following:

1. Design a new communication protocol based on publish-subscribe model for NoC-based platforms;
2. Incorporate the new communication protocol and underlying structures to a middleware-based development approach;
3. Validate the middleware and underlying structures at instruction and clock cycle accurate abstraction levels;
4. Provide an API for the development of applications and self-adaptive systems;
5. Implement a self-adaptive system as a case study in order to evaluate the use of the proposed middleware.

1.3. Thesis Contributions and Originality

Table 1.1 presents an overview of the main contributions of this Thesis along with the main aspects that involve the research methodology. The main contributions of this Thesis are the following:

1. Publish-Subscribe programming model for MPSoC platforms following a middleware based approach (Section 4.2)[2];
2. Design-pattern based middleware following the object-oriented programming approach (Section 4.3)[3];
3. Benchmark of serialization libraries in an MPSoC platform (Section 4.4)[4];
4. Self-adaptive system model and middleware extension supporting the development of self-adaptive systems (Chapter 5)

[2] Published in part at ISCAS'17 [HAR+17]

[3] Published in part at SBCCI'18 [HAR+18]

[4] Published in part at ICECS'18 [HDFGM18]

1.4. Document Structure

The remaining of the document is organized as follows. Chapter 2 provides relevant background information about concepts that will be extended in the following chapters. Chapter 3 presents some relevant works about middleware architectures in MPSoC and other embedded software domains such as Robotics, Internet of Things, Distributed Real-time Embedded Systems and Wireless Sensor Networks. Chapter 4 presents the provided middleware related to communication aspects, such as protocol phases (Section 4.2), middleware design based on OOP (Object-Oriented Programming) (Section 4.3) and incorporated serialization feature (Section 4.4). Chapter 5 presents the middleware extension that provides the support for the development of self-adaptive systems. Chapter 6 presents a case study where we develop a self-adaptive system using the middleware and compare the results with the baseline self-adaptive system according to performance, energy and software quality metrics. Finally, Chapter 7 presents the conclusion of this Thesis and the future works that can be extended from the current state of the research.

Table 1.1: Overview of the main contributions of this Thesis.

Main Contribution	Sec.	Research Questions	Hypothesis	Evaluation Methodology					
				L[1]	P[2]	PL[3]	Method	Workload	Metrics
Publish-Subscribe programming model for MPSoC platforms following a middleware based approach [HAR+17]	4.2	1. Are there works that use the middleware design approach in MPSoC Environments? 2. Publish-Subscribe model has already been used as programming model in MPSoC platforms?	1. Considering an application execution time metric, a publish-subscribe programming model presents similar performance results compared to an MPI solution on a given platform. 2. The memory resources required by the publish-subscribe based middleware do not exceed the memory available on an MPSoC platform with a maximum of 512KB.	ILP[4]	FreeRTOS-based MPSoC	C	Case study comparing the proposed publish-subscribe-based middleware with an MPI programming API at same platform	DTW Application	1. Memory Footprint 2. Application Execution Time
Design-patterns based middleware following the OOP approach [HAR+18]	4.3	1. Is OOP feasible to design systems in the MPSoC domain? 2. What are the techniques for circumventing the memory resources required by the compiler?	1. The memory footprint can be significantly reduced by the use of design-level optimization techniques and compiler options on an MPSoC platform. 2. OOP along with design-patterns improves the performance of applications in an MPSoC platform.	ILP[4]	FreeRTOS-based MPSoC	C++	Case study comparing the proposed new middleware with the previous middleware implementation	DTW, Producer-Consumer and MPEG Applications.	1. Memory Footprint 2. Application Execution Time
Benchmark of serialization libraries in an MPSoC platform [HDFGM18]	4.4	1. Are there works that evaluate serialization libraries in context of MPSoC environment? 2. Which serialization libraries could be ported to an MPSoC platform?	1. The use of schemas can impact code size and performance of serialization libraries on an MPSoC platform. 2. The ease of use of serialization libraries can increase the code size on an MPSoC platform.	ILP[4]	FreeRTOS-based MPSoC	C++	Benchmark comparison between ported serialization libraries into middleware architecture	Different Data Structs	1. Data Size 2. Code Size 3. Serialization time 4. Deserialization time
Self-adaptive system model and middleware extension supporting the development of self-adaptive systems	5	1. Are there works that aim to modularize self-adaptive systems in MPSoC? 2. What is the impact on software quality and performance when using the middleware-based design approach to the development of self-adaptive systems?	1. The use of middleware-based design approach improves the quality of the self-adaptive system software. 2. The overhead implied by the middleware-based design approach for the development of a self-adaptive system has low impact on the applications performance and the energy spent by the system.	CCP[5]	HEMPS	C++	Case Study comparing the proposed development approach with an baseline self-adaptive system	DTW, AES, MPEG, and Synthetic Applications	1. Memory Footprint 2. Application Execution Time 3. Application Execution energy 4. System Execution Time 5. System Energy 6. Cyclomatic Complexity 7. Interface Complexity 8. Function Complexity 9. Lines of Code 10. Parameters

[1] Level of simulation
[2] Platform
[3] Programming Language
[4] Instruction Level Precision
[5] Clock Cycle Precision

2. BACKGROUND

In this chapter, we present the necessary background required to understand the concepts related to the middleware design on the MPSoC domain. This chapter is organized starting with basic concepts about the MPSoC[1] domain (Section 2.1), following with details about Middleware Design (Section 2.2), related Communication Models (Section 2.3) and basic concepts about Self-Adaptive Systems (Section 2.4).

2.1. MPSoC

MPSoC architectures can be classified as homogeneous and heterogeneous. An MPSoC is homogeneous when it presents processors with the same set of instructions. It is heterogeneous when there are at least two processors with a distinct set of instructions or still when there are different processing elements composing the architecture, such as FPGAs (Field-Programmable Gate Array) or ASICs (Application Specific Integrated Circuit) [PRJW10]. A general scheme representing an MPSoC architecture is showed in Figure 2.1.

A software subsystem (SW-SS) is a programmable subsystem that includes computing/storage resources (e.g. processor and memory) and a Communication Interface used to interconnect the different resources in the MPSoC. A software subsystem enables applications to be run. It can include a Hardware Abstraction Layer (HAL), an Operating System (OS), and a middleware/API. HAL is a layer that abstracts the complexity of the hardware for the upper layers of software including OS, middleware, and application. The middleware provides an API with communication primitives to the application layer. It could manage several communication models/schemes (e.g., MPI, RPC, publish-subscribe, blocking or non-blocking) and provide an interface to the configuration of the lowest level parameters (e.g., OS and hardware). A middleware can have a HAL to facilitate its portability on different platforms. The design of a middleware/API layer is a trade-off between abstraction it provides and the overhead it causes [MSHH11]. It is necessary to consider the level of abstraction, the programming paradigm, and the interface type. The level of abstraction refers to how the middleware user views the system. The programming paradigm deals with the model for programming the applications or services [RMJPC16]. Details about middleware design are described in Section 2.2.

A hardware subsystem (HW-SS) is a custom hardware, without software elements, that can be composed of specific hardware components such as sensors, ASICs or FPGAs.

The Communication Infrastructure interconnects SW-SS and HW-SS. The Communication Infrastructure can be a performed by dedicated wires (point-to-point), a bus or a network-on-chip (NoC). In this thesis, we target NoC-based MPSoCs.

2.1.1. Memory Organization

Regarding memory organization, this Thesis considers the following models for the MPSoC domain: Centralized Shared Memory, Distributed Shared Memory, and Distributed Private Memory.

The Centralized Shared Memory (CSM) organization has a single memory component accessible by all processors [BJR11]. The communication between the processors occurs implicitly

[1] The MPSoC concepts here presented are strongly based on the Book "Embedded Software Design and Programming of Multiprocessor System-on-Chip" [PRJW10].

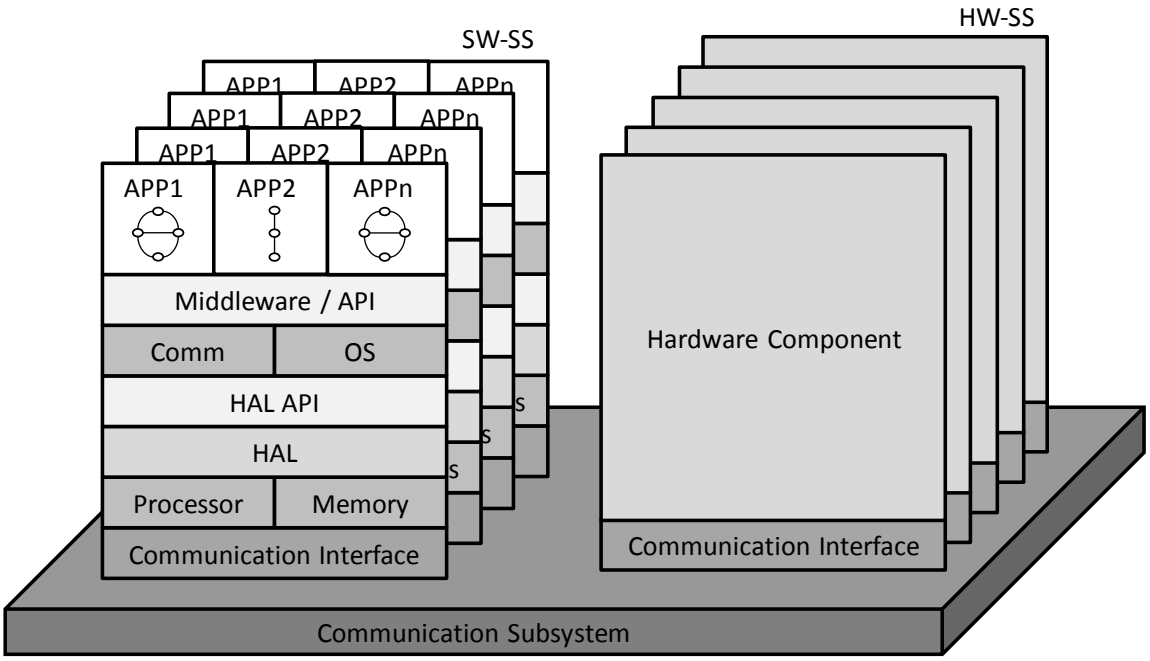


Figure 2.1: General scheme representing an MPSoC architecture containing hardware/software components on the left (SW-SS) and purely hardware components on the right (HW-SS), interconnected by a Communication Infrastructure (adapted from [PRJW10]).

through a global shared memory space. Any processor can read or write a word in the memory by just performing READ/LOAD and WRITE/STORE instructions. Additionally, each processor may have its own local memory, not shared [DJS16]. The CSM is not very useful when the number of processors becomes large, as centralized memory scheme produces a large bottleneck [DRA10].

The Distributed Shared Memory (DSM) organization share the same memory address space, but it is composed of memory elements physically distributed on the hardware architecture [BJR11]. The aim is reducing the bottleneck regarding a centralized scheme. The access to the memory remains being done by READ/LOAD and WRITE/STORE instructions. A memory management system is responsible for abstracting the distributed memory organization to the application level. This memory management system must guarantee cache coherence between the distributed shared cache system [DRA10].

In the Distributed Private Memory (DPM) organization, the processors have their physical private memory (instruction and data). There is no memory address space shared between the processors. A processor can address directly only its local memory. To access the memory of another processor, the program code must explicitly contain a message passing declaration [BJR11]. The DPM organization scales well, but the software development is more complex due to the need of synchronizing the interprocessor communication in the application level [DRA10].

2.1.2. Common Parallel Programming Models for MPSoCs

A parallel programming model specifies how parts of the application running in parallel exchanges data to one another and what synchronization operations are available to coordinate the activities. Applications are written in a programming model. Examples of parallel programming models are as follows [PRJW10]: shared address space and message passing. In the shared address space, the communication is performed by reading and writing shared memory locations. This programming model provides special atomic operations for the synchronization and data protection. In the message passing programming model, the communication is performed between a specific sender and a specific receiver. There are no shared locations accessible for all processors. Variants of *send* and *receive* are the most common communication primitives used in this programming model.

A number of parallel programming models have been defined recently, such as: OpenMP [Cha01] and PThreads [But97] for shared memory architectures, and MPI [MPI15] for message passing architectures. OpenMP [Cha01] is a pragma-based programming model that allows the compiler and run-time system to exploit the hardware complexities thus abstracting these details from the programmer. Hence the performance of applications is highly dependent on efficient compiler and run-time implementations. OpenMP assumes a shared memory model, with all the threads having access to the same, globally shared memory [PRJW10]. PThreads [But97] or Portable Operating System Interface (POSIX) Threads is a programming model for creating and manipulating each of the workers called threads [KMZS08]. The programmer must to explicitly create and destroy the threads by making use of API primitives. PThreads API provides mutex (mutual exclusion) and semaphore mechanisms to protect the portion of code that accesses shared data. MPI [MPI15] is a message-passing library interface that includes a set of primitives for point-to-point communication with message passing, collective communication, process creation and management, one-sided communications, and external interfaces [PRJW10]. More details regarding MPI communication aspects will be treated in Section 2.3.2.

The message passing programming models can be divided in three types regarding synchronization [PRJW10]: synchronous, asynchronous blocking, and asynchronous non-blocking. At the synchronous, when the source executes a *send* operation (or equivalent) and the destination has not yet executed a *receive* operation (or equivalent), the source is blocked until the destination executes the *receive* operation. At the asynchronous blocking, the source is not blocked when the destination has not yet executed a *receive* operation. Therefore, it can continue its processing. However, when the destination executes a *receive* operation (or equivalent), it is blocked until the reception is completed. At the asynchronous non-blocking, both the source and destination are not blocked when executing the *send* and *receive* operations.

There are a number of communication models that could be performed by a message passing programming model. They are different in aspects of synchronization and coupling. We explore in this Thesis the following communication models, detailed in the respective sections: MPI (Section 2.3.2), Remote Procedure Call (RPC) (Section 2.3.1), and publish-subscribe (Section 2.3.3).

2.2. Middleware Design

The management of the MPSoC complexity can be maintained transparent to the programmer through the use of a middleware. The concept of "middleware" is used in distributed

systems in general. Middleware establishes a new software layer that standardizes the infrastructure's heterogeneity through a well-defined distributed communication/programming model [ICG07]. A middleware could define:

- An API for specifying data types and primitives of networked hardware/software resources;
- An high-level addressing scheme for location resources;
- An communication model for achieving coordination;
- A naming/discovery protocol, registry structure and matching relation for publishing and discovering the resources available in the network.

The design of a middleware for any environment is an important step since it will define which features will be provided. The work of [PG14] cites some features that can be incorporated to a middleware: (i) communication - abstraction of the low-level details related to communications; (ii) components - enables the systems development by assembling reusable software modules; (iii) adaptive - enables the reconfiguration of the both hardware and software architecture to modify the provided services; (iv) context-aware - interacts with the environment where applications are run, and take itself action to make changes at run-time.

2.3. Communication Models

The communication models classified in this Thesis differ in the level of decoupling between the communicating pair (source and destination). The decoupling level can be decomposed in three dimensions [EFGK03]:

- *Space*: the destination does not know who is the source, and vice versa;
- *Time*: the communicating pair does not need to be active at the same time in the system; the source can generate data or invocations while the destination is disconnected and vice-versa;
- *Synchronization*: both source and destination are not blocked when the communication primitives are called.

The decoupling in one or more of these three levels makes the resulting communication infrastructure more scalable [EFGK03]. Following, we describe the classified communication models.

2.3.1. Remote Procedure Call - RPC

RPC is based on the client/server scheme interaction, where a client (source) requests a service through a method invocation to a server (destination) and waits for a response. In its basic version, the RPC model presents strong coupling at both *time* (both source and destination need to be active at the same time), *synchronization* (the source is blocked until it receives the response) and *space* (since a method invocation holds a remote reference to each invocation when more than one is performed) levels [EFGK03].

Figure 2.2 shows the communication flow performed by RPC model in its basic version. When the source invokes (1 in Figure 2.2) a remote method at the application level, the kernel (or middleware when existing) generates a packet through the NoC (2) to the destination. This packet is received (3) and forwarded to application level that executes the respective local method (4). After ended, a response (5) is generated and sent (6) to the source through the NoC. The kernel level at the source receives the packet and forwards it (7) to the application level that was blocked waiting for the response.

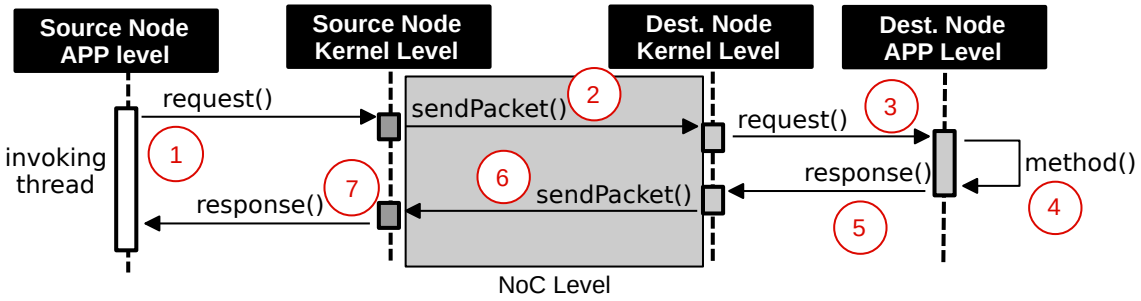


Figure 2.2: General scheme representing a basic version of the RPC model communication flow.

Some RPC implementations of this model use replicated servers to the same method in order to achieve reliability. Asynchronous RPC implementations, like CORBA [OMG11], removes synchronization coupling on the source side, avoiding the blocking. CORBA has a variant where a remote invocation is a handle through which the response will be processed when returned. With this approach, the invoking thread can continue processing other code and return to the invocation thanks to the handle. The RPC model fits with systems that have centralized data model or that are service oriented [SF09].

2.3.2. Message Passing Interface - MPI

MPI represents a low-level way of communication where the participants (source and destination) communicate by simply sending and receiving messages through two basic primitives: send(dst_id) and receive(src_id). Both the source must call the send(dst_id) primitive and the destination must call the receive(src_id) primitive to perform the communication. They also must inform the identification of each communicating pair (src_id and dst_id).

There are several implementations of MPI model. Generally, it is used an MPI variant (basic version) where the communication is asynchronous for the source and synchronous for the destination. In this case, a buffer is used (at the source or destination) to store the packet until the destination consumes it. Figure 2.3 shows the communication flow performed by this MPI variant. In this example, with buffer at the source, the source generates the send(dst_id) primitive before the destination generates the receive(src_id) primitive (could be the inverse). When the source generates the send(dst_id) primitive (1 in Figure 2.3), the message is buffered (2) at the source kernel (or where the buffer was implemented) waiting to be consumed by destination. After a time, the destination generates the receive(src_id) primitive (3). This generates a request packet (4) to the source through the NoC. As the message had already been buffered, it is packed and sent to the destination through the NoC (5). When received, the message is delivered (6) to the destination application. Note that the destination thread is blocked until the message is delivered. If the message has not been buffered yet, the thread would be blocked for longer.

Other variations of MPI synchronization are implemented, differing in buffer location at the source or destination, or blocking/non-blocking schemes. In all variations of MPI implementation, the source and destination are coupled in time (both need to be active at the same time) and space (the source knows who is the destination).

The MPI implementations can provide other features such as collective communication and both process creation and management. An example of MPI implementation is the standard for multi-processor architectures in general [MPI15]. This standard and their predecessors are considered hard to implement in embedded devices because of their code size. However, there are several implementations for embedded devices that will be treated in Section 3.1. A particular implemen-

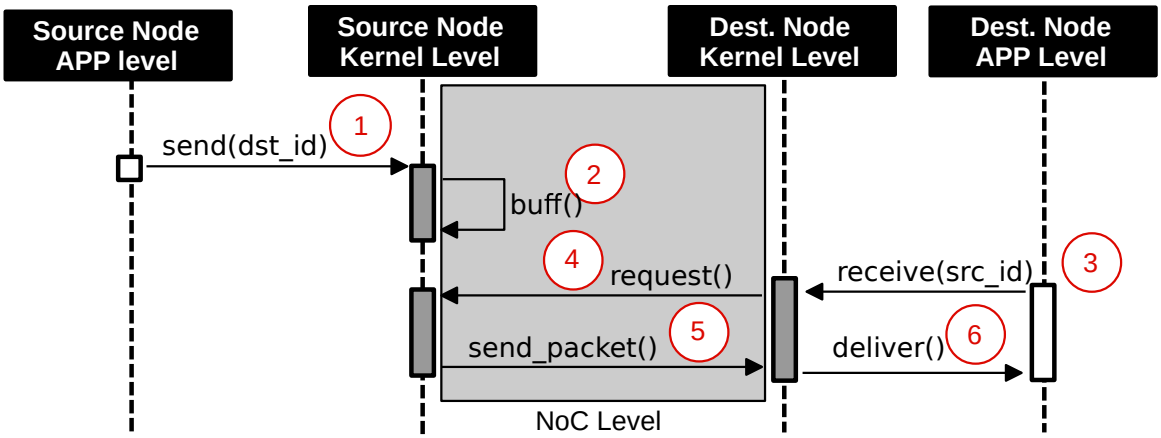


Figure 2.3: General scheme representing a basic version of the MPI model communication flow.

tation of MPI for closely distributed embedded systems is the MCAPI (Multicore Communication API) standard [MCA12]. It implements a reduced set of the MPI standard API [MPI15]. The MPI maps well systems that can be modeled as a set of dataflows transferred in a concurrent way.

2.3.3. Publish/Subscribe

The Publish/Subscribe is a model broadly used in highly distributed environment that requires a demand for more flexible communication models and systems [EFGK03]. This model implements discovery and message passing features, beyond presenting decoupling property in space, time and synchronization between the communication performed by applications and the underlying hardware/software architecture. The discovery feature is necessary as the nodes do not know who is their communicating pair. A middleware performs the role of data location management in the communication architecture.

In this model, the source (*publisher*) and destination (*subscriber*) participants communicate with each other by exchanging messages. A *subscriber* manifests interest in a particular data or event, identified by a *topic* (*subscribe* step). The *subscriber* is notified when this *topic* is generated (*publish* step). The *publisher* node must register itself in the system (*advertise* step) as the *topic* generator, so that future *subscribers* interested in this *topic* can receive notifications asynchronously. A *broker* node mediates the *advertise*, *publish*, and *subscribe* steps.

The decoupling between the communication performed by applications and the underlying HW/SW architecture occurs in: (i) **space**: a *subscriber* does not know who is the *topic publisher*, and a *publisher* does not know who consumes the *topic* generated by it; (ii) **time**: the *nodes* do not need to be active at the same time in the system; a *publisher* can generate *topics* while a *subscriber* is disconnected and vice-versa;(iii) **synchronization**: both *publishers* and *subscribers* are not blocked while they are generating or receiving *topics*; the *subscribers* are asynchronously notified when the *topic* of interest is received, and are processed via *callback function*.

A typical PUB-SUB system has multiple publishers, subscribers, brokers, and topics. Figure 2.4 shows a general scheme with two publishers, three subscribers, three topics, and one broker. A single topic can be published by one or more publishers and subscribed by one or more subscribers.

When the system has more than one broker, a synchronization protocol is required between them for maintaining the list of publishers and subscribers updated in each broker. This approach is useful for reducing the bottleneck in the broker access, and it also provides data redundancy in case of a fault in a broker.

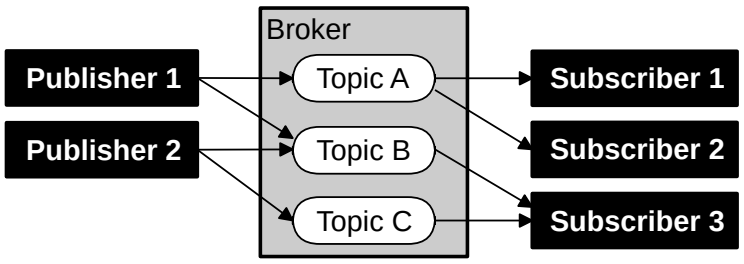


Figure 2.4: General scheme of a publish-subscribe system.

Table 2.1: Decoupling level in each communication model [EFGK03].

Communication Model	**Decoupling level**		
	Space	**Time**	**Synchronization**
RPC	No	No	Destination-side
Asynchronous RPC	No	No	Yes
MPI	No	No	Source-side
PUB-SUB	Yes	Yes	Yes

Summarizing the three treated models of communication, the Table 2.1 shows the decoupling level of each communication model.

2.4. Self-Adaptive Systems

The demand by a SoC that responds to the need for adaptability of the system is growing. It is becoming impracticable that the application programmers to have the systems knowledge necessary to manage all the possibilities of configuration that a system provides in design-time [Hof13]. In addition, design-time adaptive techniques are not efficient in a dynamic workload environment [JSHP14]. Run-time adaptive techniques have been proposed to guarantee higher performance [MFRC15] [CCM14], lower power dissipation [JSHP14], and reliability [TR13] for MPSoC environments prone to dynamic workload or failures.

A system that responds by itself to changes in the environment is called in the literature by terms such as adaptive systems, autonomic systems, self-* systems, goal-oriented systems, etc. [Hof13] [DJS16].

Figure 2.5 shows a comparison of traditional and self-adaptive systems. Traditional systems make actions without flexibility to change its behavior according to system response. In contrast, self-adaptive systems are capable of observing their environment, and changing their policies at run-time (altering a decision made earlier) in a closed loop [Hof13].

We do not find in the literature terms that represent the property related to the function of "Observe" and "Act" in self-adaptive systems. Therefore, we define here two terms that are used in this Thesis:

- **Observability**[2] - A system with the property of observability means that it provides real-time observation of possible information provided by sensors and other elements of the architecture.
- **Configurability** - A system with the property of configurability means that it provides a way to drive control actions to the configurable elements of the architecture.

[2]We emphasize that the "observability" term is different from that found in control theory [SD14].

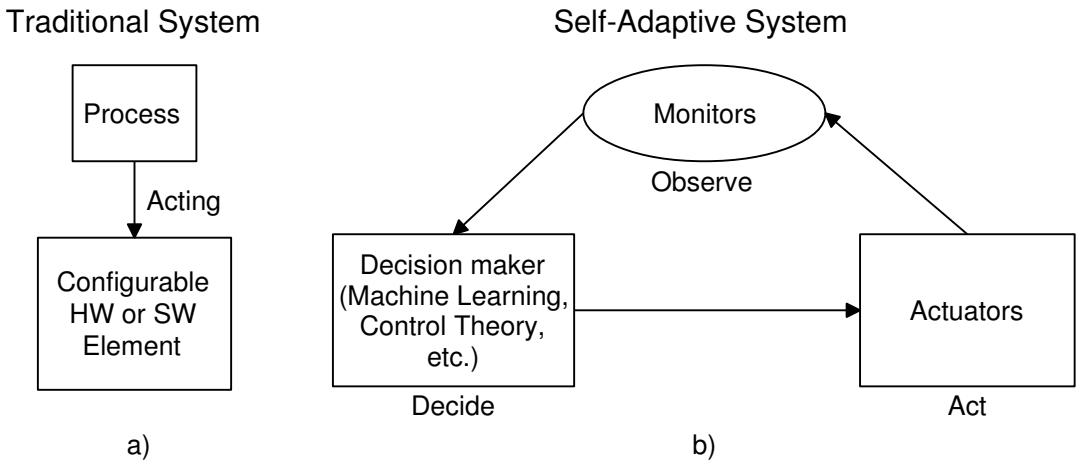


Figure 2.5: Comparison of (a) traditional and (b) self-adaptive systems (adapted from [Hof13]).

3. STATE OF THE ART

The research topic of middleware is very vast, with several hundreds of relevant papers in many areas of the Computer Science. In order to guide the review of the state of the art of this topic, we have defined a research design that also guided the entire process of research, development and evaluation of the activities covered by this Thesis.

The research design, as illustrated in Figure 3.1, is composed of three macro phases: exploratory, development and evaluation.

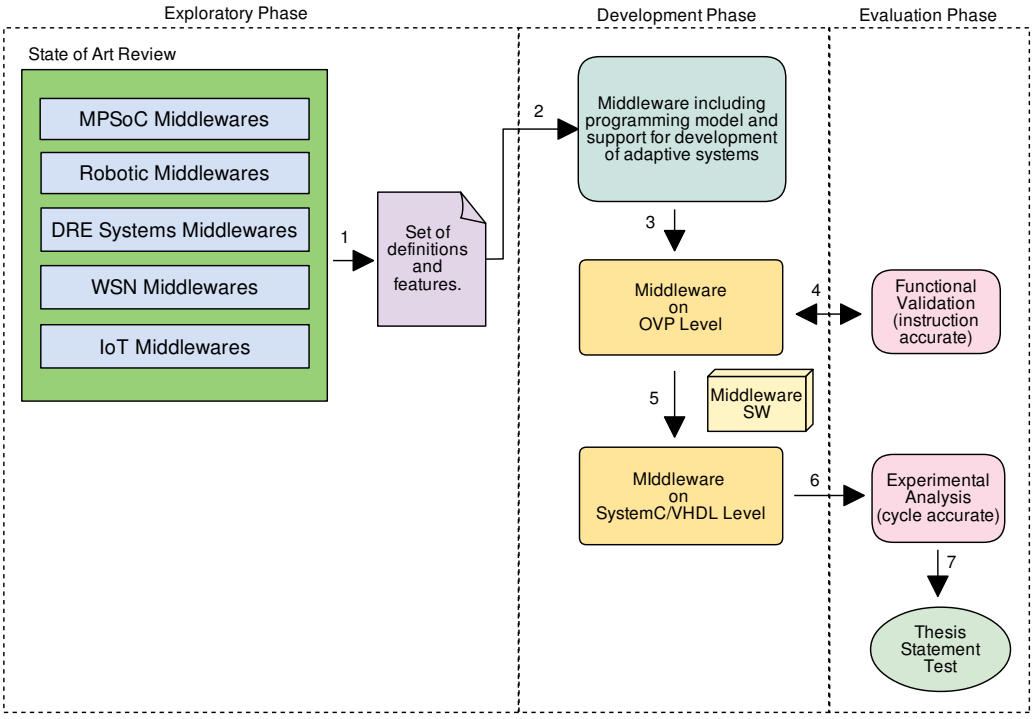


Figure 3.1: Research design of the Thesis.

In the exploratory phase, we do the review of the state of the art, where the goal is to perform a wide theoretical and technical interpretation about middleware in MPSoC domain, detailed in Section 3.1. We also expand the search to other similar application domains such as Robotics (Section 3.2.1), Internet of things (Section 3.2.2), Distributed Real-Time and Embedded Systems (Section 3.2.3) and Wireless Sensor Networks (Section 3.2.4).

In the development phase, we follow an evolutionary design of the proposed middleware. The development and evaluation phases are detailed in Chapters 4, 5 and 6. The related chapters may also present an incremental study of the state of the art.

3.1. Middlewares for MPSoCs

This section details the state of the art on the topic of middleware for MPSoCs. Chapter 2 detailed the main programming/communication models used in the various domains. Each of the papers found in the state of the art study is described and classified according to their programming models. In each work, we highlight the presence of an API and its respective primitives as a property that allows the extension of the proposal for the development of applications and/or services over

the proposed API. In addition, we highlight the presence of communication and adaptive features. Table 3.12 shows an overview of the collected works.

The "middleware" term is not commonly used in the MPSoC domain. We understand middleware for this domain as an existing layer between the application level and the operating system (or directly hardware when without operating system).

The RPC-based communication scheme (more detail about RPC, see Section 2.3.1) is used by some works in the MPSoC domain. Cassano et al. [CCJ+14] present an inter-processor communication interface for data-flow-centric heterogeneous embedded multiprocessor systems. This model enables applications invoking functions and passing parameters from another application executing on a different processor. The proposed communication interface implements a point-to-point and half-duplex communication. The communication is structured into flows, where client nodes send request messages to server nodes to invoke functions and send parameters, and server nodes send response messages to client nodes to send back the results. Table 3.1 shows the communication primitives available on this work. Another work that uses RPC-like communications is presented in [HLLL08], specially designed for streaming programming. However, the platform case study is very simple (an ARM PE and one DSP), and the scalability is not evaluated.

Table 3.1: RPC-based communication primitives proposed by [CCJ+14].

Primitive	**Description**
create client queue (opID, serverID)	Client allocates the queue to receive replays.
remote invocation (opID, serverID, seqNumber)	Client creates, initializes and sends the remote function invocation.
close flow (opID, serverID, seqNumber)	Client or server deallocates the queue of the flow
create server queue (opID, queueSize)	Server allocates the queue to receive requests.
wait remote invocations (opID)	Server waits for remote invocations of a given functionality and pop a message from the queue associated with the function
remote invocation replay (opID, clientID, data)	Server replays to a remote invocation.

Several works present approaches in order to include support for MPI in specific MPSoC architectures. In [GHHDB10], the most frequently used MPI primitives are available through an API to allow the execution of any MPI compatible software in the RAMPSoC framework [GB10]. The RAMPSoC framework is an MPSoC with a distributed memory approach consisting of a combination of heterogeneous processors. In [GWHB11], the authors adapt an embedded Linux kernel to extend support to its MPI API, allowing multitasking and multithreading processing. Table 3.2 presents some of the available API primitives.

Table 3.2: MPI-based communication primitives proposed by [GWHB11, GHHDB10].

Primitive	**Description**
MPI_init()	Initialize an application
MPI_Finalize()	Finalize an application
MPI_Send()	Send a message (ex.: ELF code) to a remote address
MPI_Recv()	Request and receive a message (ex.: result of a computation) of a remote address
MPI_Bcast()	Send a broadcast message for all remote addresses

Garibotti et al. [GOB+13] implement an open-source and customizable NoC-based MPSoC platform at RTL (Register Transfer Level). The main characteristic of this platform is its ability to enable the creation of clusters according to a CSM architecture. Each cluster is composed of a shared memory and tiles. Applications mapped onto a cluster share memory through the host tile while those mapped onto different clusters transfer data via message-passing. It consists of a tiled multicore platform comprising: embedded MicroBlaze cores; a NoC router based on Hermes [MCM+04]; and an internal scratchpad memory to store application code and microkernel. Pthreads are used with a shared-memory parallel programming model. Recently, the work has incorporated DSM capabilities [GBO+16], with modifications in the microkernel and software stack allocation. The microkernel was modified to support a different memory address space configuration. The software stack allocation splits the object code over the system. The legacy CSM multithreaded code can be ported over to the DSM architecture. A subset of the widely used Pthread API has been ported into the microkernel, as shown in Table 3.3.

Table 3.3: Pthread-based primitives proposed in [GOB+13] and [GBO+16].

Primitive	**Description**
pthread_create()	Creates a new thread in the calling process
pthread_exit()	Terminates the calling thread
pthread_join()	Waits for the specified thread to terminate
pthread_mutex_init()	Initializes the specified mutex
pthread_mutex_destroy()	Destroys the specified mutex
pthread_mutex_lock()	Locks the mutex object
pthread_mutex_unlock()	Unlocks the mutex object
pthread_barrier_init()	Initializes a barrier object
pthread_barrier_wait()	Synchronizes participating threads at the barrier pointed to by the barrier argument

Carara et al. [CCM14] present an approach that enables differentiated communication services in the application level onto NoC-Based MPSoCs. A communication API exposes the communication services offered by the NoC to the application developer. This work assumes that the multiple applications on the same platform have a distinct set of communication requirements. Therefore, each of which may benefit from distinct network communication services provided by programming API. The proposed NoC offers the following services: prioritization (at message level); connection establishment (at NoC level); differentiated routing (deterministic or adaptive routing); and collective communication (multicast). Two MPSoC distributions are modified to incorporate the proposed API: HeMPS [CdOCM09] and HS-Scale [ASB+09]. Table 3.4 shows the primitives available in the developed API.

Table 3.4: MPI-based communication primitives proposed by [CCM14].

Primitive	**Description**
Send(Message; target; priority);	Message transmission with priority (HIGH or LOW) for HeMPS
MPISend(Message; target; priority);	Message transmission with priority (HIGH or LOW) for HS-Scale
Connect(target);	Connection establishment
Close();	Connection release
Multicast (Message; target list);	Multicast message

Joven et al. [JBA+13] present a QoS-driven reconfigurable parallel computing framework. The aim is to hide hardware many-core complexity and support parallel programming on scalable

NoC-based clustered MPSoCs. A customized on-chip MPI (ocMPI) library was presented supporting a distributed shared memory with no cache coherency protocol. Two QoS services are available to meet application requirements: prioritization (at message level - up to eight levels) and connection establishment (at NoC level). Table 3.5 shows the primitives available on ocMPI API.

Table 3.5: MPI-based communication primitives proposed by [JBA+13].

Type	**Primitive**
Management	*ocMPI_Init(), ocMPI_Finalize(), ocMPI_Initialized(), ocMPI_Finalized(), ocMPI_Comm_size(), ocMPI_Comm_rank(), ocMPI_Get_processor_name(), ocMPI_Get_version()*
Profiling	*ocMPI_Wtick(), ocMPI_Wtime()*
Point-to-point Communication	*ocMPI_Send(), ocMPI_Recv(), ocMPI_SendRecv()*
Advanced and Collective Communication	*ocMPI_Broadcast(), ocMPI_Barrier(), ocMPI_Gather(), ocMPI_Scatter(), ocMPI_Reduce(), ocMPI_Scan(), ocMPI_Exscan(), ocMPI_Allgather(), ocMPI_Allreduce(), ocMPI_Alltoall()*
QoS - set up an end-to-end circuit unidirectional or full duplex	*ni_open_channel(uing32_t address, bool full_duplex)*
QoS - tear down a circuit	*ni_close_channel(uing32_t address, bool full_duplex)*
QoS - set high-priority in all w/r packets between an processor and a memory	*setPriority(int PROC_ID, int MEM_ID, int level)*
QoS - reset priority	*resetPriority(int PROC_ID, int MEM_ID)*
QoS - reset priority of a specific processor	*resetPriorities(int PROC_ID)*
QoS - reset all priorities on the system	*resetAllPriorities()*

Aguiar et al. [AJMH13] present the HellfireFW framework that allows the configuration of the customizable RTOS and the communication protocol. With the framework, the designer can manually explore the platform design space by defining initially the application model with data of period, worst-case execution time, deadline, and workload of the application. Furthermore, the designer chooses an initial HW/SW platform configuration. After, this configuration is performed resulting several graphical results for the designer to analyze and to reconfigure the platform in case of the results were not satisfactory. This approach relies on expertise and intuition of the designer, what could be considered a disadvantage. Further, the application behavior must be known in design time, what is a restriction for dynamic applications. Table 3.6 shows the communication primitives available on HellfireOS. These primitives are implemented following the MPI communication model and each task has a circular reception queue, with configurable size to hold incoming packets. During the receiving process, whenever the receiving queue is empty, the task is either blocked (in case of a blocking primitive) or kept in the primitive call until a timeout occurs.

Table 3.6: MPI-based communication primitives proposed by [AJMH13].

Primitive	**Description**
HF_Send(processor_id, task_id, buffer, size_msg)	Message transmission.
HF_Receive(processor_id, task_id, buffer, size_msg)	Message reception.

Ruaro et al. [RCM14] present a NoC-based MPSoC framework with an API to support run-time adaptive QoS management technique. The communication between homogeneous processing

elements is based on blocking send/receive MPI primitives. A set of others primitives is available to set the tasks QoS constraints, obtained through application profile step in design-time. With these constraints, the OS actuates in the processor scheduling priority or/and in the task migration to achieve the constraints. The available API primitives are not listed.

A similar work that emphasis on run-time system adaptability and fault-tolerance is proposed in [DCT+13]. It is presented the MADNESS project, where an MPSoC framework is available with a middleware infrastructure in order to achieve run-time migration of processes among tiles, and exploit reconfiguration strategies in the case of faults involving processing elements. No more details about the primitives API are accessible. Table 3.7 shows the communication primitives available on the API.

Table 3.7: MPI-based communication primitives proposed by [DCT+13].

Primitive	Description
send(receiver_id, message, tag)	Sends a message with "tag" to "receiver_id" (destination processor, or process in case of multithread).
receive(sender_id, tag)	Receives a message with "tag" from "sender_id" (sender processor, or process in case of multithread).

Mahr et al. [MLIB08] present an another message passing library for MPSoC, supporting the use of several networks. The library is integrated into a high-level synthesis flow, where an FPGA-based MPSoC is created from a specific parallel program. Table 3.8 shows the communication primitives available on the library.

Table 3.8: MPI-based communication primitives proposed by [MLIB08].

Primitive	Description
MPI_Send	Sends a message.
MPI_RSend	Sends in ready mode.
MPI_BSend	Sends in buffered mode.
MPI_SSend	Sends in synchronous mode.
MPI_Recv	Receives a message.
MPI_SendRecv	Combines a Send and a Recv call in one call.
MPI_Bcast	Broadcasts a message to a set of nodes.
MPI_Gather	Collect multiple message segments into one.
MPI_Scatter	Delivers message segments to multiple receiver.

Minhass et al. [MOS09] present a reduced version of MPI for MPSoC. The case-study platform consists of four Altera FPGA boards, with each FPGA device hosting a Quad-Core NoC. The used processor is the Nios II. A 10kB on-chip memory has been provided to each processor in order to store the software executable file. The NoC platform supports four standard communication MPI-based primitives (blocking receive), listed in Table 3.9.

Motakis et al. [MKC11] present an abstract API on the Spidergon STNoC platform to explore the management of the NoC services aiming to adapt the hardware resources through software techniques. A library named libstnoc is available to enable dynamic reconfiguration and access to information services such as energy management, routing, and security. Although the API improves the dynamic adaptation of NoC resources according to the state of the system, it cannot be extended to hardware or software elements beyond the NoC. The available API primitives are not listed.

Table 3.9: MPI-based communication primitives proposed by [MOS09].

Primitive	**Description**
NOC_MPI_SEND(buffer_s, msg_len, dest)	Sends a message to "dest".
NOC_MPI_RECV(buffer_r, msg_len_r, source, multiple_source)	Receives a message from "source" or "multiple_source".
NOC_MPI_PID_SEND(s_pid, d_pid, dest, buffer_s, msg_len_s)	Sends a message to the process "d_pid" into "dest".
NOC_MPI_PID_RECV(self_pid, desired_pid, desired_source, buffer_r, msg_len_r)	Receives a message from the specific process "desided_pid" into "desired_source".

Khemaissia et al. [KMKL16] present a middleware that handles run-time reconfiguration regarding mapping and migration of tasks on an MPSoC platform named RMPSoC. The middleware is based on master and slaves agents. The master agent controls the evolution of the whole system before applying software or hardware reconfiguration. A slave agent checks the power consumption and compliance with constraints of each processor. The middleware listens for input reconfigurations, arranges the parameters of tasks and monitors the traffic on the NoC. The work does not provide an API for middleware programming to allow, for example, the addition of other reconfiguration mechanisms.

Ross et al. [RRPS16] demonstrate a programming model for the Epiphany many-core platform based on MPI standard. The Epiphany platform is a 2D tiled mesh Network-on-Chip (NoC) of low-power RISC cores with minimal functionality. The provided MPI-based API enables MPI codes to execute on the RISC array processor with little modification. A minimal subset of the MPI standard, shown in Table 3.10, was implemented. The goal of the work is to highlight the importance of fast inter-core communication for the architecture. However, no additional features regarding the adaptability of the platform are provided.

Table 3.10: MPI-based communication primitives proposed by [RRPS16].

Primitives	
MPI_Init	*MPI_Cart_coords*
MPI_Finalize	*MPI_Cart_shift*
MPI_Comm_size	*MPI_Sendrecv_replace*
MPI_Comm_rank	*MPI_Send*
MPI_Cart_create	*MPI_Send*
MPI_Comm_free	*MPI_Recv*

Kim et al. [KKKH16] present a software platform, named SoPHy+, for hybrid resource management that can be ported to various many-core accelerators. The platform provides run-time environment for adaptive resource management by performing task remapping according to dynamic behaviors of concurrent applications. The platform was implemented on Xeon Phi coprocessor and Epiphany platforms. SoPHy+ provides a design flow across design-time and run-time stages. At design-time, a programming front-end module automatically generates platform-dependent function codes from dataflow specification of applications and makes mapping and scheduling of tasks following Pareto-optimal solutions. At run-time, SoPHy+ performs task remapping to adapt the platform following a hybrid scheme that performs task migration and check-pointing according to pre-computed results made at the design-time stage. No further details are provided about the software platform extension capability.

Sarma et al. [SDG+15] present the Cyber Physical SoC (CPSoC) that is an MPSoC platform that deploys an high-level paradigm to self-aware embedded systems combining a sensor-actuator-rich architecture and the closed loop computing model to achieve adaptability. CPSoC employs a middleware layer to control the manifestations of computations (e.g., aging, overheating, parameter variability etc.) on the physical characteristics of the chip itself and the outside interacting environment. Despite providing a comprehensive infrastructure of virtual/physical sensors and actuators, the approach is intrinsically part (hard-coded) of the MPSoC fabric designed by the same research group [SDV+13]. The authors do not provide details about the availability of an API for middleware programming to customize new adaptive techniques or other enhancements.

Automated generation of parallel application is an alternative to providing abstraction at the application level. The CAP (Communication Aware Programming) [HZZ+14] is a framework composed of hardware architecture and a language programming that aims efficient communication for NoC architectures. It uses the X10 language [SBP+09] to build resource aware applications. From application communication requirements defined in the software code, the NoC hardware is configured aiming application speedup, low NoC utilization and power consumption. Virtual channels guarantee throughput and latency for individual applications. The available API primitives added to original X10 language are not listed in this work. Other similar approaches are used by [CCS+08] and [CPC10]. However, all these proposals need to change the compiler in order to generate the parallel applications.

Some works use the Multicore Communication API (MCAPI) standard [MCA12] aiming portability in different platforms that support this standard. A variation of the MCAPI for a heterogeneous MPSoC platform is presented by [MSHH11], named FUNCAPI. The purpose is to provide a standard programming API to different processor and OS types as well as hardware IP-blocks. The case study is a platform composed of two processing elements (NIOS) and an external memory interconnected by a HIBI (Heterogeneous IP Block Interconnection) network. The architecture is in an FPGA connected via PCIe to a PC (Personal Computer). The application code can be transferred to FPGA or distributed among PC and FPGA processors. The transport layer communication is performed by MCAPI standard. It supports the channels and message passing (connection-less) MCAPI communication types, without broadcast and multicast features. Table 3.11 shows the communication primitives available on this platform, based on MCAPI functions. A similar work is presented in [RLC15], reporting lower memory footprint, but only makes use of channels MCAPI communication.

Table 3.11: MPI-based communication primitives proposed by [MSHH11].

Primitive	**Description**
msg_send	Sends a (connectionless) message.
msg_recv	Receives a (connectionless) message. Blocks until the whole message has arrived.
pktchan_send	Sends a (connected) packet on a channel.
pktchan_recv	Receives a data packet on a (connected) channel.
wait	Waits for a non-blocking operation to complete
connect_pktchan	Connects send and receive side endpoints with a channel (non-blocking function)

3.1.1. Middleware Comparison for MPSoCs

Table 3.12 shows the comparison between the works cited in this state of the art review. We classify each work according to its model of communication, programming, memory scheme (visible to application), used operating system, and available QoS/communication features. We use some abbreviations in the table, listed following:

- MP: Message Prioritization;
- CS: Circuit Switching Establishment;
- AR: Adaptive Routing;
- TM: Task Migration
- SP: Processor Scheduling Priority;
- MC: Multicast;
- BC: Broadcast;
- BA: Bandwidth Allocation;
- CSM: Centralized Shared Memory;
- DSM: Distributed Shared Memory;
- DPM: Distributed Private Memory;
- X*: Partially;
- X": Inherited from the underlying platform;
- API: Availability of API.

Table 3.12: Middleware comparison for MPSoCs.

Middleware	Model	Programming	Memory	OS	QoS/Communication Features								
					MP	CS	AR	TM	SP	MC	BC	BA	API
[CCJ+14]	RPC-like	Multithread	DPM	Without OS									X
[HZZ+14]	X10	Multithread	DPM	in-house OS		X							
[MSHH11]	MCAPI	Single-thread	DPM	Without OS		X							X
[GOB+13]	Pthread	Multithread	CSM	in-house RTOS									X
[GBO+16]	Pthread	Multithread	DSM	in-house RTOS									X
[MLIB08]	MPI-like	Single-thread	DPM	Without OS							X		X
[MOS09]	MPI-like	Multithread	DPM	Without OS							X		X
[GWHB11]	MPI-like	Multithread	DPM	embedded Linux							X		X
[DCT+13]	MPI-like	Multithread	DPM	Xilkernel (Xilinx)				X					X
[JBA+13]	MPI-like	Single-thread	DSM	in-house OS	X	X					X		X
[AJMH13]	MPI-like	Multithread	DPM	in-house RTOS									X
[CCM14]	MPI-like	Single-thread	NORMA	in-house OS	X	X	X			X			X
[RRPS16]	MPI-like	Multithread	DPM	in-house OS									X
[MKC11]	ad-hoc	Multithread	DPM	embedded Linux								X	
[RCM14]	ad-hoc	Multithread	DPM	in-house OS		X	X*	X	X				
[SDG+15]	ad-hoc	Multithread	DPM	in-house OS	X	X	X	X	X			X	
[KMKL16]	ad-hoc	Multithread	DPM	in-house OS				X					
[KKKH16]	ad-hoc	Multithread	DSM	in-house OS				X					
This Thesis	PUB-SUB	Multithread	DPM	Any	X"	X"	X"	X"	X"	X"	X"	X"	X

Analyzing the Table 3.12, most works present variations of the MPI model or use ad-hoc protocols to perform the communication. In most of the works, the operating system is based on an in-house implementation, being some of them real-time operating system. QoS features are generally present in the most recent works. The works that present adaptive techniques (CS, AR, TM, SP and BA) use hardware or software monitors. The data from the monitors are used to adapt the system to its dynamic behavior, maintaining the desired level of QoS. The communication scheme to distribute the monitor data in the system and apply actuations to configurable elements is commonly coupled in the kernel space (ad-hoc model), in a non-standardized way. In this thesis, we present a publish-subscribe middleware that can be ported to any operating system by adapting the hardware/software abstraction layer. In this way, the middleware inherits the QoS features of the underlying MPSoC platform. In addition, an API is provided for both usability and extensibility of the proposed middleware. In the case studies to be presented in the Chapters 4, 5 and 6, we have ported the middleware to a MPSoC platform based on FreeRTOS and another MPSoC platform based on an in-house OS.

3.2. Middleware for other Fields Related to Embedded Software

This section describes features of existing middleware in domains correlated to the MPSoC domain, characterized mainly by memory usage restrictions: Robotics (Section 3.2.1), Internet of things (Section 3.2.2), Distributed Real-Time and Embedded Systems (Section 3.2.3) and Wireless Sensor Networks (Section 3.2.4). The purpose of this section is not to compare the middleware of the correlated domains with the middleware proposed in this Thesis. Instead, we analyze and collect possible features that was applied in the design of the middleware proposed in this Thesis or that can be used in future extensions of the middleware.

3.2.1. Middleware for Robotics

Modern robots are complex distributed systems consisting of a number of integrated hardware and software modules. The modules can be sensors, actuators, and controllers, acting together to achieve specific tasks [MAJJ08]. The robotics domain has several sophisticated middlewares that were already surveyed by [MAJJ08] [CPS14] [ES12] [MSK15].

In the robotic domain, the middleware is viewed as a user programming interface that provides the high-level constructs of the programming language translating them to operating system level, and executes applications [MAJJ08]. It should manage heterogeneity of the hardware, facilitate the communication, improve software quality, code reuse among different robots, reduce time and costs to build new applications, allow robots to be self-configuring, self-adaptive and self-optimizing to environment changes [CPS14].

The literature lists desired attributes of middlewares for robotics. We select some of these attributes that could be applied to MPSoC environments, listed next:

- Durable data storage [CPS14]: mechanisms that allow to persist data from sensors and other devices of the system. This is important for saving taken decisions and data forwarding for nodes that enter lately in the system or in case of communication failures.
- Robustness to failures [CPS14]: the middleware could be aware of failures in order to continue performing their tasks in a degraded mode until the system has recovered from the failure.

- Management and monitoring [CPS14]: mechanisms provided to manage, debug, configure and monitor the middleware components. It is important to offer a complete vision of the sensors, actuators and other components status.
- Multi-robot coordination services [CPS14]: mechanisms to make consensus over network shared values, to elect a leader node or to assign specific tasks.
- Simulation environment [ES12]: Important for fast prototyping and educational purposes.
- Real-Time Capability [ES12]: a real-time system guarantees the reactiveness of a node by providing real-time capabilities in the system communication and computation.
- Dynamic Wiring [ES12]: dynamic configuration of connections between services of components at run-time, making both control flow and the data flow configurable.
- Automatic resource discovery and configuration [MAJJ08]: services and devices can be dynamically available/unavailable in the system. Automatic and dynamic resource discovery and configuration are important in order to support mechanisms of self-adapting, self-configuring, and self-optimizing.

ROS

One of the middlewares most popular in the robotic domain is ROS [QCG+09], chosen by Chitic [CPS14] as the most suitable middleware for multi-robot systems. ROS [QCG+09] is a component-based framework with a message oriented communication model. It provides, among other features: (i) design of distributed applications using the publisher-subscriber and RPC models (services); (ii) automatic definition of message types; (iii) graph resource topic names; (iv) run-time programming environment.

In the design of distributed applications using the publisher-subscriber model, ROS is based on concepts like nodes, messages, topics, and services. Nodes are processes, which could perform various tasks, communicating with each other via messages. Messages are data structures, which consist of primitive data types (ex. like integer, float, boolean, and others), arrays of primitive types, or sets of other messages. Nodes send their messages by assigning them to topics, represented as strings. Nodes receive messages when subscribing to one or more topic.

In the ROS, the publish-subscribe model is efficient but can not handle synchronous processes [MSK15]. Based on RPC communication model, the concept of services was introduced to solve this problem. These services are represented by a string and a pair of messages: request and response. A node requests the execution of a service located on a remote node. After the remote node executes the service, a response is generated and sent to the node that originated the request. The nodes can request services only from one node [MSK15]. A node can make a persistent connection to a service, which enables higher performance at the cost of less robustness to service provider changes [ROS16b].

Both ROS publisher-subscriber and services models have an associated a message type declaration that is defined in an easy way through one ".msg" file for publisher-subscriber or ".srv" file for services. The ROS programming environment automatically generates all the resulting source code necessary to handle this message type. An example of ".msg" file, where a message is structured with three interleaved data type, is showed next.

```
string first_name
string last_name
uint8 age
```

An example of ".srv" file, where the request message is structured by two "integer" and the response message is structured by one "integer", is showed next.

```
int64 a
int64 b
---
int64 sum
```

ROS present a Graph Resource Names mechanism for providing resource encapsulation. Graph Resource Names uses a hierarchical naming structure that is tracked for all resources such as Nodes, Parameters, Topics, and Services. For example, when a node wants to provide a service, the respective service is defined as a system resource within a namespace. The resource name is explicitly defined in the node source code. The resource name is shared in the ROS system and accessible through available tools in ROS programming environment. In general, nodes can create resources within their namespace and they can access resources within or above their namespace [ROS16b]. Alternatively, it can be defined global and private resource names.

ROS also provides a programming environment for application development in run-time. This programming environment contains several command line tools that are used to interact with the system at run-time. Among other features, the command line tools could be used to know what services/topics are currently available in the system and what are the types of messages related to each service/topic.

3.2.2. Middleware for Internet of Things

The Internet of Things (IoT) comprises an environment with a wide variety of physical devices or things such as home appliances, surveillance cameras, monitoring sensors, actuators, displays, vehicles, machines and so on. IoT applications are applied in many different domains, such as home automation, industrial automation, medical aids, mobile health-care, intelligent energy management and smart grids, automotive, traffic management, smart cities, and many others [RMJPC16]. The IoT is very rich in terms of survey papers, such as [RMJPC16] [CM12] [Fer15]. A middleware for IoT provides an abstract layer interposed between the IT infrastructure and the applications. It aims to hide the technological details to enable the application developers to focus on the development of the IoT applications [CM12].

Some characteristics of IoT domain present similarities with the MPSoC domain [RMJPC16]:

- Heterogeneous devices: the "things" in IoT are naturally different in capacity, features, multivendor products and application requirements, getting the middleware the responsibility to hide this complexity from the programmer;
- Resource-constrained: Embedded computing and sensors present limitations about their processing, memory, and communication capacity;
- Dynamic network: IoT devices leave or join the network according to unexpected behavior;
- Context-aware: Context-awareness plays a vital role in the adaptive and autonomous behavior of the things in the IoT, eliminating human-centric mediation.

Middleware requirements for the IoT could be categorized as both functional and non-functional. Functional requirements capture the services or functions which a middleware provides, and non-functional requirements capture QoS support or performance issues [RMJPC16]. Some functional requirements listed in [RMJPC16] could also be applied to the MPSoC domain:

- Resource discovery: Involves the automatic discovery of new services or applications that enter the network; each device must announce its presence and report the features it offers;
- Resource management: The use of resources by applications need to be monitored, allocated and provisioned in a fair way; the architecture should potentially have the ability to reconfigure itself to meet the needs of the applications;

- Event management: Transforms simple observed events into meaningful events.

Some non-functional requirements listed in [RMJPC16] and [Fer15] could also be applied to the MPSoC domain:

- Scalability: The middleware needs to be scalable to accommodate the growing number of devices, applications and services;
- Reliability: A middleware should remain operational even in the presence of failures;
- Availability: Even if there is a failure somewhere in the system, its recovery time and failure frequency must be small enough to achieve the desired availability;
- Security/Privacy: The middleware must consider the context-aware property in situations where a device could disclose confidential information (e.g., the location of an object or person) that should not be available to all devices;
- Ease-of-deployment: The application design should not require expert knowledge or support;
- Adaptative: A middleware needs to be adaptive so that it can evolve to fit itself into environment changes.

Mosquitto

Mosquitto [Ecl16] is a broker that implement the MQTT (Message Queuing Telemetry Transport) protocol [MQT99], based on publish-subscribe communication model. Mosquitto enables communication between subscribers and publishers through a topic subscription. It inherits some features of the MQTT protocol: three level of QoS regarding message delivery; retained messages for late subscriptions; durable connections that store messages to forwarding in case of short subscriber disconnection.

Mosquitto also provides the use of two wildcards in subscriptions, in addition to allowing clients to subscribe to specific topics. A wildcard can be used to automatically subscribe to adjacent/subsequent topics in a hierarchical topic structure. The available wildcards are "+" and "#". The "+" is the wildcard used to match a single level of hierarchy. For example, for a topic "a/b/c/d", some examples of subscriptions are:

+/b/c/d	-> subscription in all topics of the first level of hierarchy
a/+/c/d	-> subscription in all topics of the second level of hierarchy
a/+/+/d	-> subscription in all topics of the second and third level of hierarchy
+/+/+/+	-> subscription in all topics in all levels of hierarchy

The "#" is the wildcard used to match all subsequent levels of hierarchy. For example, for a topic "a/b/c/d", some examples of subscriptions are:

#	-> subscription in all topics in all levels of hierarchy
a/#	-> subscription in all topics of the second, third and fourth level of hierarchy
a/b/#	-> subscription in all topics of the third and fourth level of hierarchy
+/b/#	-> subscription in all topics of the first, third and fourth level of hierarchy

Hermes

Hermes [Pie04] is a middleware created for large-scale distributed applications based on events. It is appropriated in systems where mobility and failures are common. The events can be either type-based or attribute-based, following the publish-subscribe model. It uses fault-tolerance mechanisms that can tolerate different kinds of failures in the middleware. It addresses interoperability and reliability requirements. The middleware consists of several modules that implement features such as fault-tolerance, reliable event delivery, event-type discovery, and security.

3.2.3. Middleware for Distributed Real-Time and Embedded Systems

A real-time system is defined as a special kind of system whose logical correctness is based on both the correctness of the outputs and their timeliness. The applications must satisfy particular timing constraints [PG14]. In the case of real-time distributed systems, the timeliness must be guaranteed to take into account complex dependencies among data or processes allocated in different processors. The delay induced by the network needs to be considered. General-purpose middlewares present several potential sources of indeterminism, such as transmission/reception queues for network messages and delays in transport service or dispatching of requests. Real-time middlewares aim to solve these issues by implementing support of QoS parameters [SR14] and predictable mechanisms, such as the use of special-purpose real-time communication networks or the management of scheduling parameters [PG14]. The following survey papers of middlewares for Distributed Real-Time and Embedded Systems (DRE) were studied [SR14] [PG14].

DDS

DDS (Data Distribution Service) [Par03] was the first open international standard directly addressing publish/subscribe middleware for real-time systems. It provides fine control of QoS parameters, that includes reliability, bandwidth control, delivery deadlines, message prioritization, and resource limits. This control is tuned per-node or per-stream basis. In other words, each publisher/subscriber pair or even a specific topic can establish independent QoS agreements. DDS is well suited for dynamic systems, quickly discovering new nodes, new participants on those nodes, and new data topics between participants [SF09]. DDS has already been deployed in several real-time scenarios such as Defense [SC08], Automation [RR08], or Space [GLH+12]. The set of QoS parameters available in DDS allows several aspects of data, networks and computing resources to be configured at application level. It may be classified in the following categories [PG14]:

- Data Availability: controlling queuing policies and data storage parameters such as durability, lifespan and history;
- Data Delivery: delivery parameters that represent the way that data will be presented to the application, such as presentation, reliability, partition, destination order, and ownership;
- Data Timeliness: latency parameters in the distribution of data, such as deadline, latency budget, and transport priority;
- Maximum Resources: limits the amount of resources that may be used in the system through parameters such as resource limits or time-based filter.

RT-CORBA

RT-CORBA (Real-Time Common Object Request Broker Architecture) [OMG05] is an extension of the CORBA (Common Object Request Broker Architecture) specification for real-time systems, adding new interfaces and mechanisms that aim to increase the predictability of distributed applications. CORBA [OMG11] is a middleware that follows the RPC paradigm. Although CORBA provides comprehensive support for distributed objects, this standard does not include support for real-time applications. RT-CORBA incorporates real-time features enabling applications to configure and control the system resources explicitly [PG14]. RT-CORBA allows applications to configure and control resources of: (i) processor, via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service; (ii) communication, via protocol properties and explicit bindings; (ii) memory, via buffering requests in queues and bounding the size of thread pools [SK00].

3.2.4. Middleware for Wireless Sensor Network

A Wireless Sensor Network (WSN) consists a collection of different sensors nodes which are connected through wireless channels and which can be used to build different distributed system for data collection and processing [SKL11]. WSNs are used in many industrial and consumer applications, such as industrial process monitoring and control, and machine health monitoring [LM14]. Implementing applications for WSN domain is non-trivial due to the high distribution, dynamic properties, and heterogeneity of the devices. Incorporating a middleware layer is a used approach to meet the design and implementation issues of WSN applications, such as integration, scalability, reliability, security, usability, QoS, and operational issues [LBP15]. The following survey papers describe middlewares for WSN domain: [LBP15] [LM14] [SKL11].

PS-QUASAR

PS-QUASAR [CDRT13] is a middleware based on publish/subscribe model that provides QoS support (reliability, deadline, priority) and an API to applications. It presents features that enable multicast techniques, energy efficient, sensor/actuator support, and broker-less routing. The broker-less routing is implemented by maintaining in each node a routing table, using the Bellman–Ford algorithm [Bel58, FF62, Moo59] to build a routing tree. The routing protocol dynamically adapts its behavior to the neighbor status to route the packets using different paths and allows the information to be transmitted with certain QoS policies. It handles priority, deadline and reliability requirements in the communication between nodes. The routing scheme is built at the expense of memory resources of the sensors.

MQTT-S

MQTT-S [HTSC08] is an IBM publish-subscribe protocol extended from MQTT [MQT99] protocol. It is adapted for constrained devices, and low-bandwidth, high-latency or unreliable networks. The system that uses the MQTT-S protocol is composed by brokers that use the original MQTT implementation. The publisher and subscriber nodes communicate with the broker using the MQTT-S protocol. Reliability QoS is implemented on three levels: (i) best effort (sends just once either successfully received or not); (ii) retransmits until the message is acknowledged (may incur redundancy, since messages are delivered, but they may arrive multiple times at the destination because of the retransmissions); (iii) assures no redundancy, since it assures not only the reception of the messages, but also that they are delivered only once on the destination side [SARM16].

UPSWSN-MM

UPSWSN-MM (Ubiquitous Publish/Subscribe platform for WSN with Mobile Mules) [TN12] is an application-specific pub/sub middleware for ubiquitous WSN with Mobile Mules. The Internet users can access WSN data anytime, from anywhere (Ubiquitous). The system is composed of sensors distributed over a monitored area. The sensors publish data (such as temperature, humidity, light intensity, and hiking speed) for mobile phones, which are then sent to interested subscribers (Internet customers) via mobile phone networks (wifi, 3G). The system is not suitable for real-time systems due to the need to acknowledge in the sent messages. It also has no other QoS mechanisms as priority and deadline.

TinyDDS

TinyDDS [BS10] is the DDS version for WSN. It allows WSN applications to have fine control over non-functional properties, allowing the system adaptation according to the application requirements. The middleware provides data aggregation, event filtering, routing, among others features. TinyDDS performs event publication in an adaptive way according to dynamic network conditions, balancing its performance among conflicting objectives (by evolutionary multi-objective optimization mechanism) [SARM16].

PRISMA

PRISMA [SDP+14] is a resource-oriented publish/subscribe middleware for WSN. It provides features such as the resource discovery and QoS mechanisms to meet applications constraints. The system architecture is composed by distributed brokers, where each broker is responsible for receiving and forwarding the data sensors. The energy is saved by using a topology algorithm, where the subscription messages are not broadcasted, but it is forwarded only to the nodes that are active and relevant to the topic.

3.2.5. Middleware Comparison for other Fields Related to Embedded Software

Table 3.13 shows the comparison between the works of correlated domains cited in Section 3.2. The works are classified according to its model of communication, domain, and additional features present in each of them. We use some abbreviations in the table, listed as follows:

- P/S: Publish/Subscribe Model;
- RPC: Remote Procedure Call Model;
- SF (Set Frequency of Update): Publisher can indicate the maximum rate that data can be sent in a time interval for agreement purpose;
- RN (Reliability in Network Level): Allows set differing level of reliable delivery (e.g. best-effort and in-order) in the network level;
- TN (Timeleness in Network Level): Allows management on one or more timing parameters in the network level (e.g. delay and jitter);
- PH (Publication History): Publishers can store published data for late-joining subscribers;
- PM (Priority in Message Level): Determines the processing priority that a message will be processed for routing purposes in the network;
- CT (Compound types): It is possible to define compound data types by use of primary data types;
- PS (Priority in System Level): Determines the processing priority that a requisition will be processed in the system level (Operating System);
- BA (Band Allocation): Allows band allocation in the network level for the application flow;
- PC (Private connection): Allows configure end-to-end private connection in the network level, non-multiplexed;
- SC (Secure Connectivity): Allows one or more features related to secure connectivity in the network level (e.g. authentication, encryption and access control);
- PE (Publication Expiry Interval): Allows set a time in that a publication message is valid;
- DN (Disconnect Notification): A notification message is sent when a client is disconnected by some cause.

Table 3.13: Middleware comparison for other application domains related to embedded software.

Middleware	Model	Domain	Additional Features											
			SF	RN	TN	PH	PM	CT	PS	BA	PC	SC	PE	DN
ROS [QCG+09]	P/S, RPC	Robotic						X						
DDS [Par03]	P/S	DRE	X	X	X	X	X					X		
RT-CORBA [OMG05]	RPC	DRE		X	X				X	X	X			
Hermes [Pie04]	P/S	IoT		X	X		X			X		X		
Mosquitto [Ecl16]	P/S	IoT		X		X							X	X
PS-QUASAR [CDRT13]	P/S	WSN		X	X		X						X	
MQTT-S [HTSC08]	P/S	WSN		X	X		X							
UPSWSN-MM [TN12]	P/S	WSN		X										
TinyDDS [BS10]	P/S	WSN		X	X		X							
PRISMA [SDP+14]	P/S	WSN	X		X									

Analyzing the Table 3.13, we verified that the presence of QoS features is practically a constant, with some middlewares covering more features than others. Network-level reliability (RN) is implemented in almost all middlewares, since they are often used in unreliable physical channels or transport protocols. Therefore, the middleware must be able to guarantee the delivery of the end-to-end messages. Network-level timeliness (TN) attributes are also present in most middlewares, which allows temporal predictability in the communication between different elements of the network. Message-level prioritization (PM) is also used, allowing applications that need a higher level of QoS to have priority in messages multiplexing process on best-effort networks. Security-related features are not frequently exploited in these middlewares.

4. MIDDLEWARE COMMUNICATION

This chapter presents the Message-Queuing System-on-Chip (MQSoC) Middleware, which is the main contribution of this thesis, highlighting the management of the publish-subscribe communication protocol. MQSoC Middleware, called in the next sections just as *"middleware"*, follows an evolutionary design incorporating (i) a proposed publish-subscribe protocol, (ii) design based on object-oriented techniques and design patterns, and (iii) serialization/deserialization feature to deliver a new programming API for communication between tasks of an application. The middleware's communication protocol described in this chapter will also be used in the middleware support for development of self-adaptive services detailed in Chapter 5.

Section 4.1 presents the FreeRTOS-based MPSoC platform used to validate (i), (ii) and (iii) which are detailed in Sections 4.2, 4.3 and 4.4, respectively. Section 4.2 presents the proposed publish-subscribe protocol phases implemented into a middleware design based on non-object oriented programming. As a case study, we present a comparison with an MPI-based programming model for MPSoC architectures. The contributions detailed in Section 4.2 have been published in part at ISCAS'17 [HAR+17]. Section 4.3 presents the object-oriented middleware design, published in part at SBCCI'18 [HAR+18]. Section 4.4 presents a performed benchmark on available libraries for data serialization for embedded systems with small memory, aiming flexible data encapsulation. The contributions detailed in Section 4.4 have been published in part at ICECS'18 [HDFGM18].

4.1. FreeRTOS-based MPSoC Platform

This section presents the adopted platform used to validate the middleware features detailed in Sections 4.2, 4.3 and 4.4. This platform was presented in [AMR+16] along with an MPI-based programming API. Figure 4.1 illustrates an instance of the adopted platform composed of a 4x4[1] NoC-based MPSoC platform, with homogeneous processing elements (PEs) organized in clusters of 2x2 size. Each PE includes a Cortex-M4F processor, private random access memory (RAM), network interface, DMA and router. RAM stores the system (kernel) and applications. Each PE runs an extended FreeRTOS kernel independently, which uses cluster-based distributed management with dynamic task mapping feature. The MPSoC hardware infrastructure, including NoC and PE, was described using OVPSIM APIs[2] by Imperas, which provides an instruction accurate simulation framework. A timing model presented in [ROR+14] ensures the instruction accurate by capturing the executed instructions for each PE, estimating an execution time from total executed instructions.

In this platform, each PE has functions of Global Manager (GM), Local Manager (LM), or Slave PE (SP). The LM is responsible for mapping application tasks onto SP PEs belonging to its cluster. The GM, in addition to the LM functions, assumes global functions such as application-to-cluster mapping and controlling the access to application repository. Note that GM is the only PE that has access to the application repository. The SP executes the application tasks. The applications enter in the system in an order and time instant defined at design time. Always that a new application must enter in the system, the application repository generates an interruption that is treated by the kernel present in the GM.

In this programming model, the applications are modeled as a task graph. Figure 4.2-a shows the DTW (Dynamic Time Warping) application following this programming model. A directed

[1] The size of this MPSoC instance is only for representation in the figure. The experiments performed in the next sections could use another configuration.

[2] http://www.ovpworld.org/technology_ovpsim

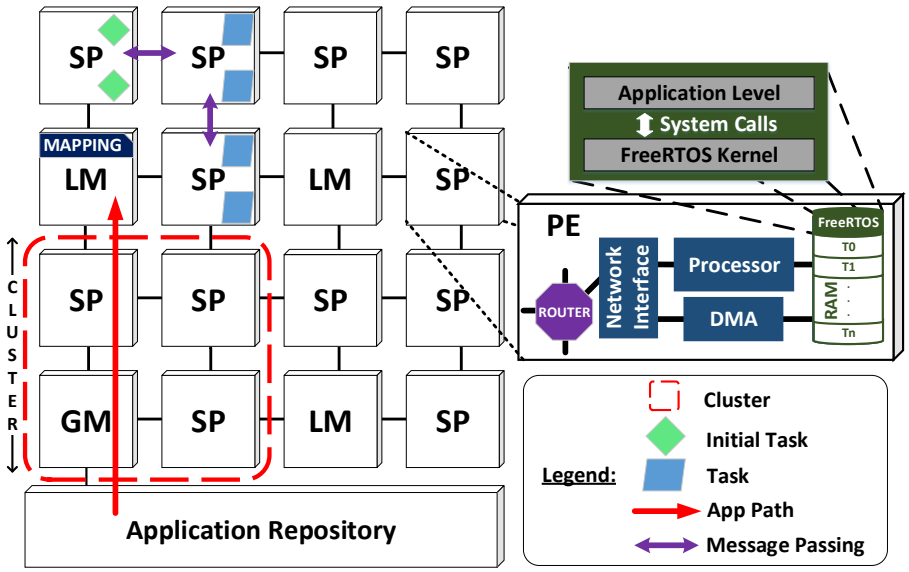


Figure 4.1: FreeRTOS-based MPSoC 4x4 platform instance [AMR+16]

arrow between two tasks (blocks in the figure) means that the first task sends data to the second one.

Abich et al. [AMR+16] have modified a FreeRTOS kernel to support the exchange of messages through the NoC incorporating a Task Manager (TM) with a Task Management Structure (TMS), a Communication Buffer (CB), and a Task Location Buffer (TLB). TMS contains for each task the local ID, the global application ID, the task relationship ID, and the CB. CB stores the outgoing task messages. TLB stores the pair task ID and PE physical address to know where the application tasks are allocated.

The SP kernel supports multitasking using a paging mechanism, which divides the memory into fixed-size *pages*. The number and size of pages are defined at design-time. For single-task processing, the number of pages must equal one. For multitasking, the number of pages should be greater than one (one page for each task). For example, when the number of pages is set to 2, the SP could handle up to two tasks.

The MPI-based API includes two communication primitives: MPI_Send and MPI_Receive, which are used to transfer data dedicated to inter-task communication. A communication buffer at sender PE stores the message whenever an MPI_Send is invoked, and it suspends the sender task when the buffer overflows. When a task invokes the MPI_Receive, the kernel blocks the task until the data are available at the buffer of the sender PE. The MPI_Send and MPI_Receive primitives are atomic, that is, for each call to MPI_Receive on the receiver task, there should be a call to MPI_Send on the respective sender task.

The OVPSim simulator framework supports the customization of the MPSoC platform, being possible to define: platform size, GM position, cluster size, the maximum number of tasks (pages) per PE, CB size and application set to execute. All the experiments that we have performed in this platform use the Nearest Neighbor (NN) mapping heuristic [MCS+15]. The NN heuristic algorithm tries to map the communicating tasks as near as possible. The algorithm tests all n-hop neighbor PEs, n varying between 1 and the NoC limits (or cluster limits, when clusterized) in a spiral way, stopping when the first free PE with free page is found.

In the mapping request for an application, GM searches for a cluster with free resources (pages) to receive the application. If no resources are available, the application is scheduled to run later. Otherwise, GM sends the application header to the LM of the selected cluster. The LM performs the mapping heuristics to select the SPs that will receive the application's initial tasks

(identified in the application modeling). The LM then communicates with the selected SPs, which request the GM to send the object code of the task. The other tasks are mapped on demand when a task wants to communicate with another one not yet mapped. When the tasks are finished, the LM releases the allocated resources. When an application is finished, the LM reports the resources available to the GM, which can map another incoming application or finalize the execution of the system.

4.2. Proposed Publish-Subscribe Protocol for MPSoC environments

This section describes the proposed publish-subscribe protocol for MPSoC environments incorporated into a middleware-based design. The middleware is evaluated in the FreeRTOS-based MPSoC platform detailed in Section 4.1. Section 4.2.1 explains the motivation of using the publish-subscribe programming model into MPSoC environments. Section 4.2.2 shows how an application initially modeled in MPI programming model can be ported to the publish-subscribe programming model. Section 4.2.3 presents the publish-subscribe protocol phases implemented into the middleware-based design along with the provided API to perform the communication between tasks of an application. Finally, Section 4.2.4 describes the evaluation of the proposed publish-subscribe protocol presenting the results compared to the MPI programming model.

4.2.1. Motivation

The MPSoC programming is based on shared or distributed memory models. The shared memory model is easier to program based on threads, but it is potentially less scalable. The distributed memory model is scalable, but the software development is more complex due to strong coupling and synchronization between communicating elements [EFGK03]. MPSoC programming frameworks have evolved in terms of functionality, but they are still based on the same main programming model: MPI [RCM, GWHB11, CCM14, GBO+16].

Concerning widely distributed embedded system, on-chip or not, several authors argue that traditional programming models are not appropriate to deal with ever-increasing unpredictable and dynamic applications' behaviors [EFGK03, DDF+06]. These models are based on static assumptions as conventional communication and synchronization, usually defined in design time. Therefore, the applications adaptation is hard in a dynamic environment with unpredictable changes, such load fluctuations. Also, it is assumed that the nodes are on the same network at the same time, and that each node knows its communicating pair. The *publish-subscribe* (PUB-SUB) programming model has been used in middlewares for highly distributed domains, such as: MQTT[3] for sensors networks and mobile devices domains; DDS[4] for real-time systems domains; and ROS[5] for robotics domains. All these middlewares evolved to provide properties, such as reliability, security, low power consumption, and QoS [BCR14].

We bring the foundations of the publish-subscribe model to the context of MPSoC environments. Therefore, we present protocol phases to implement the publish-subscribe programming model following a middleware-based design. The middleware-based project is due to the concern that the proposed solution can be ported to other MPSoC platforms with lower cost. The *main contribution* here is developing a middleware based on publish-subscribe, which can be used to improve

[3] http://mqtt.org/documentation

[4] http://www.omg.org/spec/DDS/1.4/

[5] http://www.ros.org

the programmability of distributed private memory NoC-based MPSoCs. For evaluation purpose, developed middleware has been incorporated into FreeRTOS-based kernel embedded in an MPSoC platform. The proposed publish-subscribe protocol allows: i) unicast/multicast-like communication, abstracting the NoC protocols and infrastructure; ii) decoupling in the time, space, and synchronization dimensions. The experiments compare the use of MPI API and proposed publish-subscribe API to perform the communication between the tasks of the DTW application.

4.2.2. Designing an Application from MPI to Publish-Subscribe

A general definition of publish-subscribe and MPI communication models can be found in Sections 2.3.3 and 2.3.2, respectively.

Bringing the foundations of the publish-subscribe model to the MPSoC domain, we support that the model can be used to perform the communication between tasks of an application, being able to replace the MPI model commonly used in MPSoC platforms. For example, the communication elements that want to produce data assume the function of *publishers*, and those who want to consume data are the *subscribers*. The *topics* represent the atomic data, which could be information about sensors, node, kernel, buffer, or any other information that needs to be made available to one or more elements interested in receiving them.

As an example, Figure 4.2 shows the DTW (Dynamic Time Warping) application with ten tasks (Bank, P1-P8 workers, and Recognizer) [RCM] represented through a task graph with MPI (a) and PUB-SUB (b) communication primitives. A directed arrow between two tasks (blocks in the figure) means that the first task sends data to the second one.

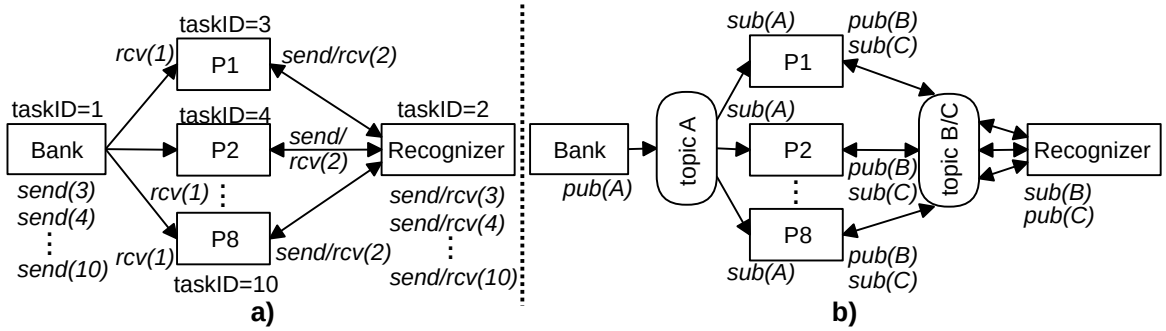


Figure 4.2: DTW task graph with a) MPI and b) PUB-SUB primitives.

In the example, the MPI primitives of the DTW application are replaced by PUB-SUB primitives. The sender side defines a topic ID for the flow, registers itself in the system as the publisher of that topic and publishes the data. The receiver side registers itself as a subscriber of that topic, setting a callback function to treat the incoming data.

4.2.3. Proposed Publish-Subscribe Protocol

We present the publish-subscribe model as an alternative to perform the communication between participants (tasks) of a parallel application. As in the publish-subscribe systems used in other domains, in our proposal the sender task (publisher) does not implicitly send messages to a specific receiver task (subscriber), what happens in the MPI programming model. Instead, the messages are classified into topics of interest, and a subscriber receives messages only of those topics

to which it has subscribed. Publishers send messages to topics without knowledge of which are the subscribers since the publish-subscribe system coordinates the communication between publishers and subscribers at system-level. In our proposal, a middleware-design approach assumes functions of coordination, specifically in a component named *Broker*. The middleware also contains components present in the clients' middleware to communicate with the broker, named *Publishers Management* and *Subscribers Management*.

In this context, we present a set of protocol phases to accomplish the publish-subscribe system coordination, that are: (i) ***BrokerAdvertise*** - the *Publishers Management* announces a new topic to the *Broker*; (ii) ***BrokerUnadvertise*** - the *Publishers Management* makes a topic unavailable to the *Broker*; (iii) ***BrokerSubscribe*** - the *Subscribers Management* announces to the *Broker* that it wants to receive data from a specific topic; (iv) ***BrokerUnsubscribe*** - the *Subscribers Management* announces to the *Broker* that it no long wants to receive data from the topic; (v) ***PublisherSubscribe*** - the *Broker* informs to the *Publishers Management* the subscriber network address to which the message must be sent in a given topic; (vi) ***PublisherUnsubscribe*** - the *Broker* informs to the *Publishers Management* that the subscriber identified by the network address no longer wants to receive message in a topic; (vii) ***Publish*** - the *Publishers Management* sends data to a specific topic that is received for all subscribed clients. The protocol phases (i) to (iv) presents, optionally, acknowledgment phases, as explained next. The protocol phases (i) to (iv) are performed from the client to the broker. The optional acknowledgment phases are performed in the reverse way. The protocol phase (v) and (vi) is performed from the broker to the publisher client. The protocol phase (vii) is performed from the publisher client directly to the subscriber client since the protocol phase (v) informs to the publisher the network address of the subscriber client.

The set of protocol phases are triggered in our middleware-based implementation when an application task performs one of the API primitives provided by the middleware. Table 4.1 shows the list of primitives available on the API. Figures 4.3 to 4.8 show the sequence diagrams representing the protocol phases and processes performed by each primitive in the middleware level and NoC.

When an application task calls the *MQSoCAdvertise(topicID)* primitive (1 in Figure 4.3), the PublishersManager in the middleware level stores the data of identification of both task and topic in its *Publishers* table (2), and sends a message of type *BrokerAdvertise* to the broker informing these data (3). The broker then stores this data into its *Publishers* table and checks if there are subscriber tasks not yet applied (4), necessary to treat cases where the broker receives a *BrokerSubscribe* message before the *BrokerAdvertise* message. In this case, the broker sends a message of type *PublisherSubscribe* back to the publisher client with the network address of the subscriber client for each subscriber not yet applied. The middleware in the publisher client then stores this data in the *Subscribers* table (6). Optionally, when the acknowledgment feature is enabled, the broker generates a message of type *BrokerAdvertiseAck* back to the publisher client (7), which sets the respective index in the *Publishers* table as acknowledged (8) and resumes the task to continue its processing (9). In parallel, the middleware could resend the *BrokerAdvertise* message to the broker if the acknowledgment message is not received after a defined timeout. When the acknowledgment feature is enabled, note that the task is suspended after calling the *MQSoCAdvertise(topicID)* primitive, being resumed when the acknowledgment message is received.

When an application task calls the *MQSoCUnadvertise(topicID)* primitive (1 in Figure 4.4), the middleware erases the data of identification of the publisher task and topic in its *Publishers* table (2) only if the acknowledgment feature is disabled. Otherwise, it sends a message of type *BrokerUnadvertise* to the broker informing these data (3) and only erases the data of identification of the publisher task and topic in its *Publishers* table when the acknowledgment message is received (6). At receiving of the *BrokerUnadvertise* message, the broker erases the respective index into its *Publishers* table (4). Optionally, when the acknowledgment feature is enabled, the broker and

Table 4.1: Primitives of the proposed experimental PUB-SUB middleware.

Primitive	Used by	Description
MQSoCAdvertise(topicID)	Publisher	Advertises the system that the respective client is the publisher of the topic identified by topicID.
MQSoCUnadvertise(topicID)	Publisher	Unadvertises the system that the respective client is the publisher of the topic identified by topicID.
MQSoCPublish(topicID, payload)	Publisher	Sends the payload data to the topic identified by topicID.
MQSoCSubscribe(topicID, callbackf)	Subscriber	Subscribes to the topic identified by topicID passing the pointer to the function that will process the message when it arrives.
MQSoCUnsubscribe(topicID)	Subscriber	Unsubscribes to the topic identified by topicID.
MQSoCYield(timeout, cnt_rcv, suspend)	Subscriber	Generates a loop that verifies in a *timeout* frequency whether a message was received; the callback function defined in *MQSoCSubscribe* primitive is executed whether there is a message to the respective topic; the loop is finished when all *cnt_rcv* messages are received; the task is suspended when *suspend* is set and no message is received (it is resumed when a message arrives).

PublishersManager in the middleware level perform the same acknowledgment phases and processing as accomplished in the *BrokerAdvertise* protocol phase, detailed before.

When an application task performs a *MQSoCSubscribe(topicID, callBackFunction)* primitive (1 in Figure 4.5), the SubscribersManager in the middleware level stores the identification of both task and topic in its *Subscribers* table (2), and sends a message of type *BrokerSubscribe* to the broker informing these data (3). The broker then stores this data into its *Subscribers* table and checks if there is a publisher registered to this topic in its *Publishers* table (4). If no publisher is found, the subscription is marked as not applied in the *Subscribers* table, waiting for an *advertise* to that topic (5). Otherwise, the broker sends a message of type *PublisherSubscribe* to the fetched publisher client with the network address of the subscriber client along with both topic and task identification (6). The middleware in the publisher client then stores this data in the *Subscribers* table (7). Optionally, when the acknowledgment feature is enabled, the broker and SubscribersManager in the middleware level performs the same acknowledgment phases and processing as accomplished in the *BrokerAdvertise* protocol phase, detailed before.

When an application task calls the *MQSoCUnsubscribe(topicID)* primitive (1 in Figure 4.6), the middleware erases the data of both task and topic identification in its *Subscribers* table (2) only if the acknowledgment feature is disabled. Otherwise, it sends a message of type *BrokerUnsubscribe* to the broker informing these data (3) and only erases the data of identification of the publisher task and topic in its *Subscribers* table when the acknowledgment message is received (8). At receiving of the emphBrokerUnadvertise message, the broker erases the respective index into its *Subscribers* table (4) and sends a message of type *PublisherUnsubscribe* to the fetched publisher client with the network address of the subscriber client along with both topic and task identification (6). The middleware in the publisher client then erases this data in its *Subscribers* table (7). Optionally, when

the acknowledgment feature is enabled, the respective acknowledgment phases and processing are accomplished.

When an application task performs a *MQSoCPublish(topicID, payload)* primitive (1 in in Figure 4.7), the middleware searches by subscriber clients to the respective topic in its *Subscribers* table (2). If there is a subscriber to that topic, then the middleware sends a message of type *Publish* to the fetched subscriber clients with the identification of both topic and task (3). At receiving in the subscriber client, the middleware adds the message to the buffer (4).

When an application task performs a *MQSoCYield(timeout, cnt_rcv, suspend)* primitive (1 in Figure 4.8), the middleware verifies whether there is a message to the task in its message buffer (2). In this case, the middleware invokes the respective callback function informed in the *MQSoCSubscribe(topicID, callBackFunction)* primitive that process the message payload (3). Otherwise, the task could be suspended until a message is received (if *suspend* is true), or the task could execute any other part of its software after a number of verifications defined by *cnt_rcv* and spaced for a time interval defined by *timeout*.

The experiment described in Section 4.2.4 does not use the acknowledgment phases of the proposed publish-subscribe protocol. The acknowledgment feature is useful in scenarios with faults, which is not the case of the evaluated scenarios. A particular study where we use the acknowledgment feature was published at SBCCI'18 [DHA18] in a proposal of a lightweight extension of the publish-subscribe model with a fault recovery method for the broker structures.

4.2.4. Experimental Setup and Results

This section describes the evaluation of the proposed publish-subscribe programming model comparing the results with an MPI programming model. The platform was implemented and validated using the platform described in Section 4.1. We incorporate the middleware structure composed by C-based source code files to the platform, building an intermediate level between the kernel and application levels. Figure 4.9 shows the modified platform.

The experiments are based on the DTW application (Figure 4.2-b). This application has been chosen because it uses a communication pattern of 1:N and N:1 (N is the number of workers). This application uses eight workers. We analyze three scenarios: *MPI-all*, *MPI-dem*, and *PUB-SUB*. The first two scenarios use the MPI primitives with all the tasks mapped at the beginning of the execution (*MPI-all*), or tasks mapped on demand (*MPI-dem*), where only the initial tasks are mapped at the begin of the execution and the other tasks are mapped as soon as there is a communication among them. The *PUB-SUB* scenario uses the proposed publish-subscribe primitives and middleware, with all the tasks mapped at the begin of the execution. All scenarios use a single-cluster 5x5 MPSoC, with each PE executing a single task to stimulate the NoC communication between the tasks and evaluate the middleware protocol.

Figure 4.21 shows the results of DTW application execution time. PUB-SUB reduces the execution time from 2.6% to 29.9% as the number of patterns (iterations) is increased, respectively, from 16 to 256. Compared to MPI, the PUB-SUB model requires an initial setup time to advertise the topics. Besides, the PUB-SUB application object code is lightly bigger, taking more time to finish the task mapping. Therefore, the MPI has advantages for small communication volumes. However, the MPI model presents the drawback of generating more system calls and network interruptions caused by the messages, as detailed next.

Figure 4.11 is a detailed view of the results obtained for 64 patterns, presented in Figure 4.21. The X axis represents the order of System Calls or NIs (Network Interruptions) generated in the system, and the Y axis represents the instant of time (timestamp) in which each of them

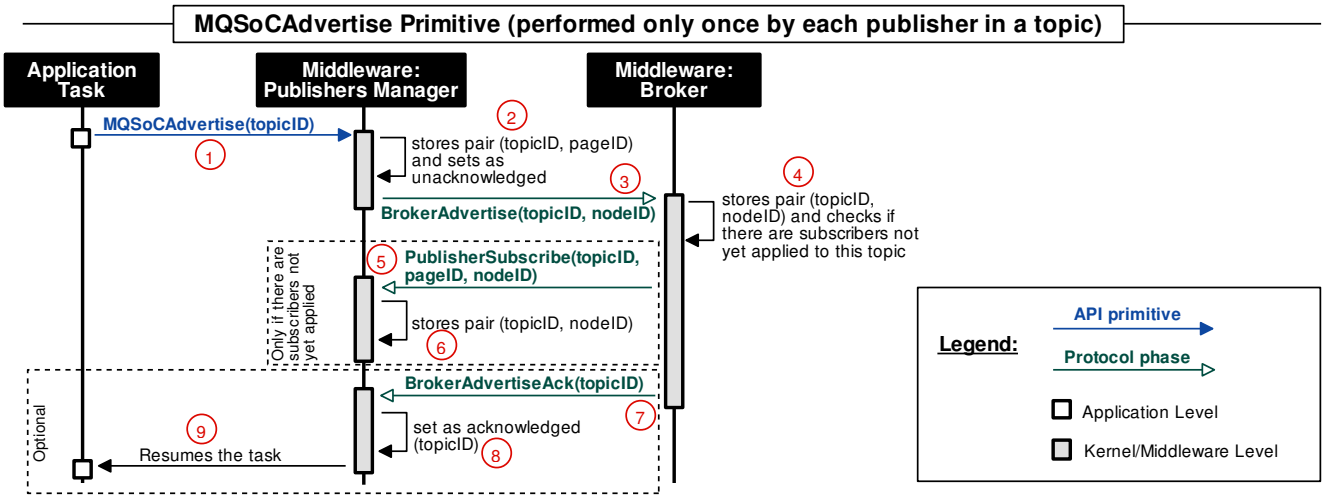


Figure 4.3: Sequence diagram of the MQSoCAdvertise primitive.

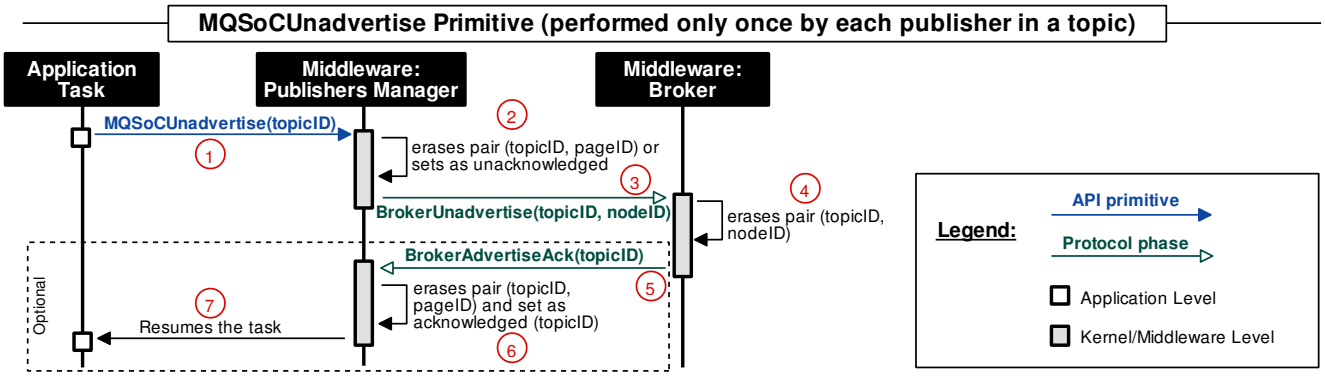


Figure 4.4: Sequence diagram of the MQSoCUnadvertise primitive.

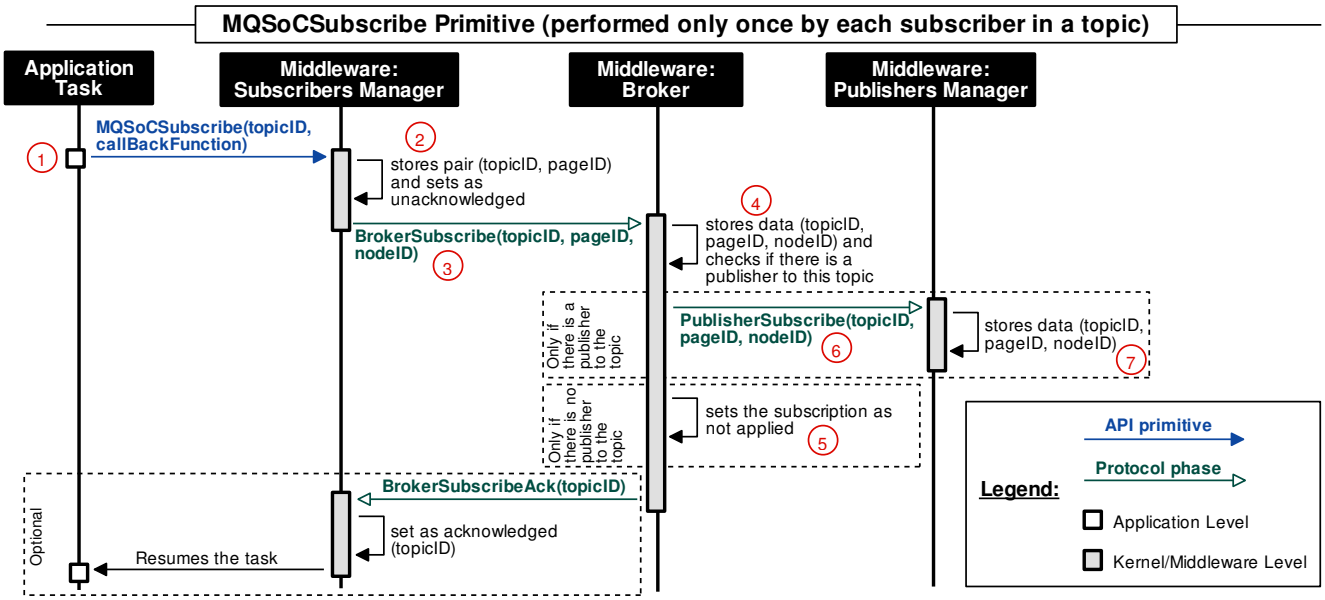


Figure 4.5: Sequence diagram of the MQSoCSubscribe primitive.

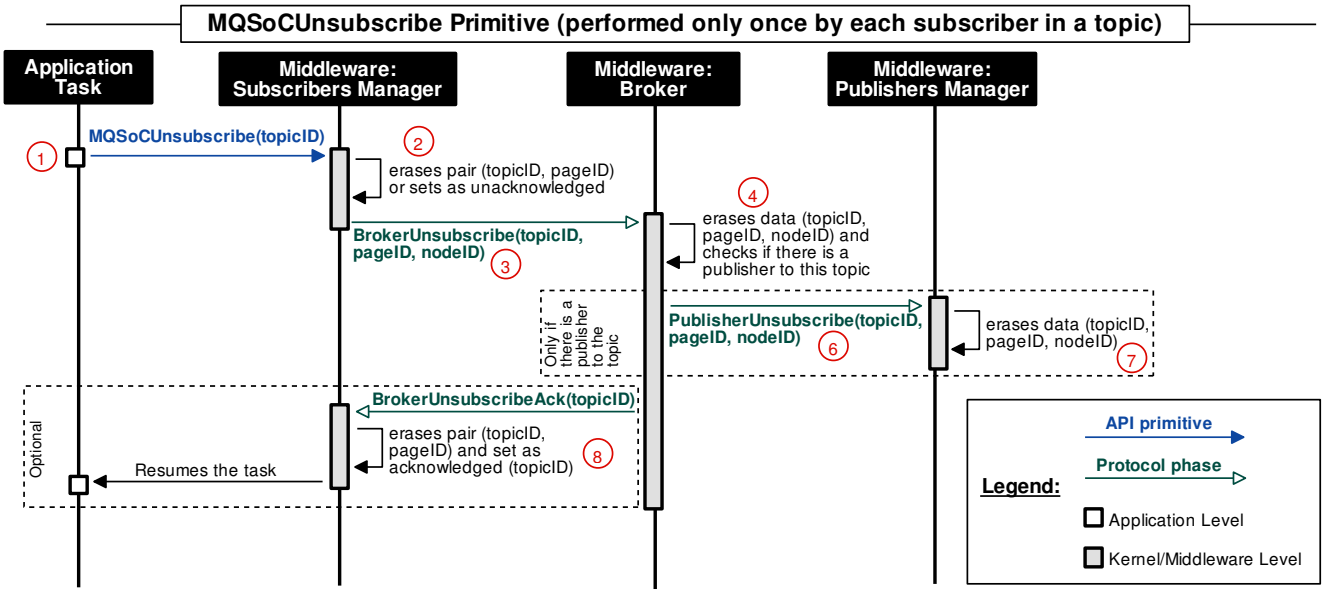


Figure 4.6: Sequence diagram of the MQSoCUnsubscribe primitive.

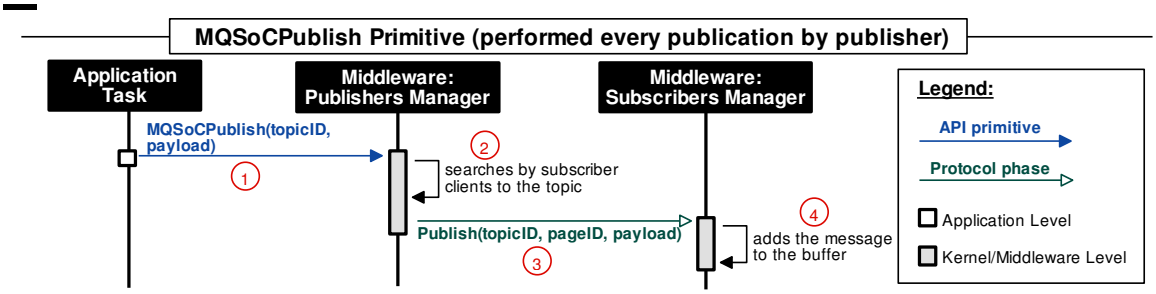


Figure 4.7: Sequence diagram of the MQSoCPublish primitive.

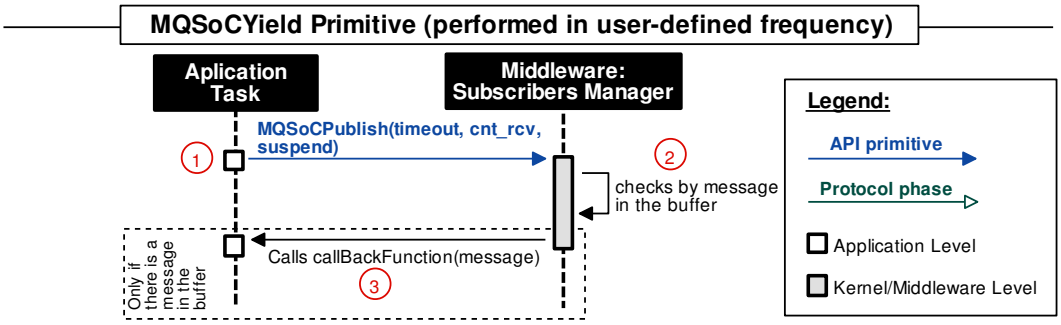


Figure 4.8: Sequence diagram of the MQSoCYield primitive.

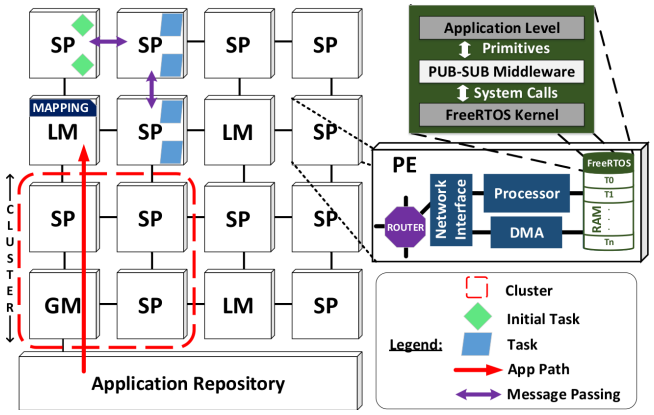


Figure 4.9: Modified FreeRTOS-based MPSoC 4x4 platform instance, adapted from [AMR+16].

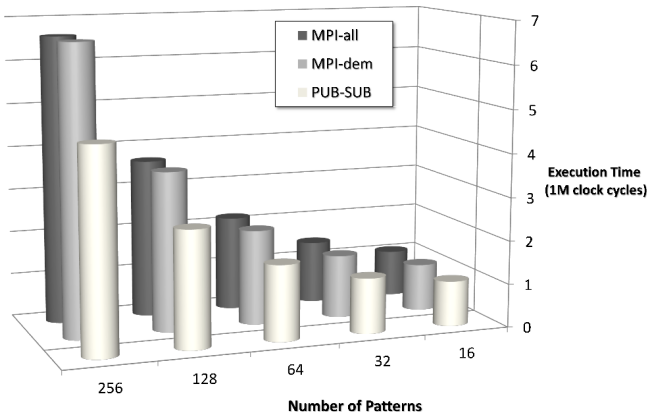


Figure 4.10: DTW execution time using MPI and PUB-SUB.

was executed. The figure also presents two lines representing the MPI and the PUB-SUB execution trace. Since both *MPI-all* and MPI-dem had the same behavior, only one is illustrated. The figure is divided into the three main phases of DTW application: *setup*, *data fork*, and *data join*.

The *setup phase* for MPI is the time between the first (1) and last (3) task allocation. The setup phase for PUB-SUB is longer because it allocates the tasks (1 to 2), and it also performs the topic advertisement, concluded at (4).

The *data fork phase* represents the time the *bank* task sends the first message to a *worker*. The PUB-SUB (4) starts later than MPI (3) due to the advertisement time. This graph shows that, once PUB-SUB starts the data fork phase (4), it starts to present advantage because the *bank* task performs only one System Call to send the message to all *workers*. On the other hand, the *bank* task of the MPI version waits for an NI caused by a message request from each *worker*, then the message is sent to the *worker* that requested it.

The *data join phase* represents the time between the first and the last message received by the *recognizer* task from a *worker* task. In the MPI version, when the *recognizer* task executes an *MPI_receive(target from)* primitive, it sends a message request to a single *worker*, going to a suspended mode until the message is received. The message request causes an NI at the *worker*. If the data is ready, the data message is sent by *worker* to the *recognizer* task, causing another NI at the destination. However, if the data is not ready, the *recognizer* task remains in suspended mode. These steps are repeated for each worker. On the other hand, in the PUB-SUB version, all *workers* publish on the same topic, and the *recognizer* task has a callback function to treat these messages. The stair case behavior observed in (5) is repeated whenever the *recognizer* task goes to suspended mode. It is resumed when a new message arrives. The *recognizer* task does not need to

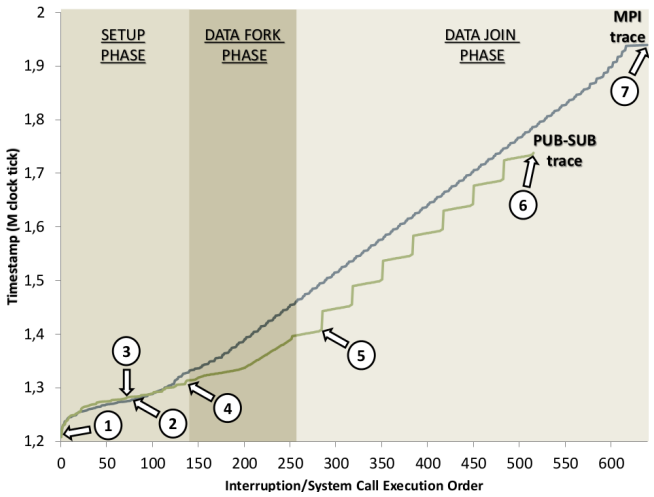


Figure 4.11: MPI vs PUB-SUB time spent in System Calls and NIs.

request data, and it does not need to block waiting any *worker*. The application is finished in (6) to PUB-SUB and (7) to MPI.

Although the PUB-SUB advertisement step adds some initial latency to start the task communication, this pays off because both publishers and subscribers know that the broker connects each other when they are ready to start. This way, the message request-response and the blocking receive executed in the MPI for each transaction are not required. Furthermore, when the application has collective communication patterns, PUB-SUB generates fewer system calls.

In terms of memory footprint, the proposed publish-subscribe middleware with FreeRTOS kernel has 23kB, which corresponds to a 7kB increase compared to the MPI-based FreeRTOS kernel.

In this initial design of the middleware proposed in this thesis, we implement all the middleware logic in C programming language, without any object-oriented technique or design pattern. The next evolutionary design of the middleware derives the publish-subscribe protocol phases, the API primitives and the general aspect of the non-object oriented middleware implementation presented in this section.

4.3. Object-Oriented Middleware

This section describes the design and evolution of the middleware designed following the object-oriented programming paradigm. Although following the initial publish-subscribe protocol phases detailed in Section 4.2.3, the middleware source code is fully re-factored in C++ programming language, making use of already widespread design patterns to become the software more modular and reusable. These software characteristics will explored in the support for the development of self-adaptive systems to be detailed in Chapter 5. The middleware is evaluated in the FreeRTOS-based MPSoC platform detailed in Section 4.1. Section 4.3.1 explains the motivation of using the proposed object-oriented middleware for the development of applications and self-adaptive systems. Section 4.3.2 presents related works. Section 4.3.3 provides recommended design patterns and best practices regarding the middleware design. Section 4.3.4 presents the the proposed object-oriented middleware architecture. Section 4.3.5 presents the experimental setup and discusses the results compared to the middleware implementation presented in Section 4.2.

4.3.1. Motivation

The constant increase of applications' complexity along with technologies constraints (e.g., power and memory wall) influenced the evolution of the embedded systems from a single core to multiprocessor architectures. MPSoC architectures provide parallel processing capabilities, aiming at covering the increasing requirements of emerging applications. Resulting complexity calls for a flexible and self-adaptive system, which must handle critical design constraints to address multiple complex applications competing for resources in the multiprocessor system.

Self-adaptive approaches have been proposed to cover the requirements of operating systems and applications at runtime [RC10, AS15, BBS15, LF15, SKK+14, AMM+17]. Software modularity, object-oriented programming, and software design patterns are examples of techniques used to provide systems with self-adaptive property. The software modularity aims to separate the software design from the other system elements that are architecture dependents. Object-oriented programming provides known advantages such as code reuse and encapsulation. Software design patterns are widespread solutions to common problems in operating systems, which rely on portable code that can be reused in many different situations. Some works target to improve the software adaptability in embedded systems, focusing on software development [LF15] and security issues [AMM+17]. Although well-known in several segments, the applicability of these techniques is not well explored in the multiprocessor systems domain.

Middleware approaches leverage patterns and techniques to bridge the gap between the functional requirements of applications and the underlying architecture [SB03]. This approach is used by solutions in the most diverse distributed environments, such as the DDS[6] for real-time systems environments and ROS[7] for robotic environments. Although the client nodes are designed to memory-constrained environments, these solutions typically make use of more burdensome infrastructure for centralized management roles (e.g., broker), which are usually hosted on a node without memory restrictions.

In this context, we present a new middleware architecture aiming to provide more modular software and replaces the previous middleware implementation. In this way, the new middleware provide the same API for communication between the tasks of an application. The modular characteristic for the software designed using the middleware is achieved by incorporating best practices of object-oriented programming, the publish-subscribe programming model, and design patterns which are suitable to MPSoCs with small memory.

4.3.2. Related Works

Several approaches have proposed the use of design patterns and modular techniques to address software modularity and support for development of self-adaptive services in different computing systems.

The authors in [RC10] conduct a study comprising project implementations to harvest adaptation-oriented design patterns that support the development of adaptive systems in general domains. A subset of collected design patterns is evaluated in an adaptive news web-server case study. In this context, the authors in [AS15] propose a set of design patterns for modeling and designing self-adaptive software systems based on IBM MAPE-K multiple control loop issues. To evaluate the applicability of the design patterns implemented in the environment, they present

[6] http://www.omg.org/spec/DDS/

[7] http://www.ros.org

Table 4.2: Related Works Comparison

Reference	Domain	Approach	Case Study	Impact Study
[RC10]	General	Collection of adaption patterns for self-adaptation expertise reuse.	News Web Server	No
[AS15]	General	Collection of design patterns for self-adaptive systems based on IBM MAPE-K multiple control loop issues.	E-learning Web Server	No
[BBS15]	General	Collection of design patterns for developing policies for self-adaptive systems at multiple levels of abstraction.	Smart Home System	No
[LF15]	Embedded Systems	Collection of design patterns with focus on software development for embedded applications.	No	No
[SKK+14]	Embedded Systems	Propose five patterns to model a self-adaptive system, comprising the Monitor, Analyzer, Decision-Making, Acting and Assessing patterns.	Object Tracking and Resource Allocation Control Engine	No
[AMM+17]	Embedded Systems	Systematic pattern-based approach that interlinks safety and security pattern engineering workflow.	Automotive System	No
OO-MQSoC (this work)	MPSoC	Middleware based on collection of Design Patterns and best practices of object-oriented programming for embedded systems, with focus on MPSoC domain.	Homogeneous MPSoC	Applications Execution Time and Memory Footprint

some case studies through an e-learning system. Furthermore, Berkane et al. [BBS15] present an approach based on design patterns for developing policies for self-adaptive systems at multiple levels of abstraction. Such system considers feedback loops modeled in a modular way, and evaluates the execution in a smart home case study scenario. Although these works [RC10, AS15, BBS15] present innovated approaches to achieve reusable design, they are designed with the focus on particular software development which comprises specific resource constraints.

Recent works propose the use of design patterns along with adaptive techniques to address the hard design constraints of embedded systems. Lakhani and Faisal [LF15] present a review regarding the evolution of design patterns developed for building architectures to diverse applications with a special focus on software development for embedded systems. Aiming to achieve performance and to cover real-time constraints, Said et al. [SKK+14] propose five design patterns used to model a loop-based self-adaptive embedded system. Further, Amorim et al. [AMM+17] present a systematic design patterns-based approach that interlinks safety and security patterns, considering an automotive use case scenario. All these approaches consider embedded systems and target particular designs that cover specific design constraints. However, none of them focus on embedded MPSoC platforms. While these platforms provide parallel capabilities, the constraints include all embedded design restrictions, even more strict, increased by the complexity of management of multiple resources and applications.

Table 4.2 shows a comparison between the proposed approach and related works, emphasizing the performed impact at each work in the respective domain. Our main contribution is to present a middleware designed using design patterns and object-oriented languages aiming to improve software modularity in the MPSoC domain.

4.3.3. Best Practices Implemented in the Proposed Middleware

The usage of design patterns allows the reuse of established solutions for known problems. In this section, we introduce the design patterns and best practices in object-oriented programming

for embedded systems that we have used in the design of the new middleware structure that will be detailed in Section 4.3.4.

Selected Design Patterns

Aiming design flexibility, the *Container* design pattern comprises a holder object that stores a collection of other objects (its elements). Containers are implemented as class templates, supporting several data types. We have used two categories of containers: *Sequence Container* and *Associative Container*. *Sequence Container* stores objects and its elements in a strict linear order, providing an interface for accessing them. Examples of Sequence Container implementations are List, Queue and Deque. *Associative Container* stores objects based on keys (indexes), differing from sequence container since it does not provide insertion at a specific position. Examples of Associative Container implementations are Map, Multimap, Flat-map and Flat-multimap. We use the Queue container to implement the Message Buffer component and the Flat-map and Flat-multimap containers to implement the managed topic tables in the proposed middleware (see Figure 4.12).

Regarding design patterns for self-adaptive systems, the *Factory* design pattern [RC10] allows the decoupling of high-level elements (e.g., monitors, decision makers and actuators) from those elements that are target-dependent (e.g., processing elements and other low-level hardware/-software components). This design pattern creates a standard interface that can be called in order to, for example, require information of a distributed monitoring infrastructure. We use the Factory design pattern to implement the Sensor/Decision Maker/Actuator interfaces detailed in Chapter 5. The *Broker* design pattern [Tar12] decouples the communication between the communicating elements in a publish-subscribe system. The *Observer* design pattern [Tar12] defines a one-to-many dependency in a system with multiple both monitoring and actuation services. With it, all dependent objects are notified about a changing of a state in an object under observation. We use the Broker and Observer design patterns to implement the publish-subscribe programming model.

Regarding software modularity, the *Hardware Abstraction Layer* (HAL) pattern [EHL+09] abstracts the underlying hardware/software structure from the rest of the system by implementing a driver with an abstract interface. We use the HAL pattern to enable portability in the proposed middleware.

Selected Programming Language

Most of the operating systems or middlewares for embedded systems use C programming language because of the run-time efficiency and the high availability of compilers for a wide range of processors. Although design patterns can be implemented in a non-object oriented programming language [Dou10], it leads to an awkward code, difficult to maintain. Recently, there have been efforts to use C++ in embedded systems [Whi11, Mia15, Kor18]. Most recent versions of C++, such as C++11[8] and C++14[9] standards, have enhanced features like type traits, operator overloading, static assertion, constant expression and concurrency support, enabling part of the C++ language support for embedded systems. The use of C++ in embedded systems with severe memory limitation (about tens or few hundreds of KBytes per processor) requires the use of techniques and best practices to reduce the code size generated by the compiler on the target platform. Following, we present some of these techniques applied to the proposed middleware. The impact of using these techniques is demonstrated in the Sec 4.3.5.

[8] https://www.iso.org/standard/50372.html

[9] https://www.iso.org/standard/64029.html

Static Memory Allocation

Dynamic memory allocation is another issue in environments with constrained memory size. The *Fixed Memory Allocation* technique [EHL+09] allows the static allocation of the memory, with its maximum size defined at compilation time, avoiding unexpected behavior at runtime. We use this technique to improve runtime predictability and reduce the software code size when the dynamic allocation and adjacent library are used to compile the source code.

Placement new

The default *new* operator in C++ allocates memory in the kernel heap area and constructs an object in the allocated memory in runtime. This approach is usually not suitable for embedded systems. The *new* operator can cause unpredictable behavior in the lack of available heap memory space. Limiting the object's maximum allocation space to a fixed amount of memory at compile time is an alternative approach. Besides that, the default *new* operator increases the code size generated by the addition of all inherent methods of this operator, such as *malloc*, *mallocr* and *free*. *Placement new* approach [Gun16] reimplements the *new* operator passing a pre-allocated memory area pointer and building the object in the given memory at compile time.

Avoid Exception Handling

Exception C++ feature adds to the code a large number of functions even when the exception feature is not used. So, in addition to not explicitly use exception handling in the code, it is advisable to include "*-fno-exceptions*" in the compiler options to disable this feature.

Compiler Optimization Options

The GCC and G++ compilers provide a set of options to control sorts of optimization. When used, they attempt to improve the performance and/or code size. The available set of optimization options depends on the target and how the compiler is configured. For example, when the primary goal is small memory size, the compiler could be instructed to optimize for size using the "*-Os*" flag in the compilation command. Other options are "*-O1*", "*-O2*" and "*-O3*" to performance optimization in exchange, generally, for large code size.

Embedded Template Library

The Standard Template Library[10] offers a set of well-tested design patterns implementations. However, it does not fit well in platforms with limited resource requirements. The Embedded Template Library[11] (ETL) is a worthwhile alternative designed for environments with restrict memory resources, since it provides containers with fixed capacity and static memory allocation. In the middleware presented in this section, we use the following ETL containers patterns: *queue*; *map*; *multimap*; and its alternatives aiming size memory optimization - *flat_map* and *flat_multimap*.

[10] https://www.sgi.com/tech/stl/

[11] http://www.etlcpp.com

4.3.4. Proposed OO-MQSoC Middleware

The proposed middleware, named Object-Oriented Message-Queuing System-on-Chip (OO-MQSoC), incorporates the publish-subscribe protocol phases presented in Section 4.2.3. In addition, we are proposing a new middleware structure based on an object-oriented approach improved with design patterns and programming best practices for embedded systems. The middleware structure also includes a Hardware/Software Abstraction Layer (HSAL), decoupling the middleware from the Operating System (OS) kernel and the hardware components.

Figure 4.12 shows the OO-MQSoC architecture, which is focused on middleware communication, containing the basic components that can be used to implement services and applications on the middleware. From the previous implementation of middleware detailed in Section 4.2, we only derived the phases of the publish-subscribe protocol. The rest of the middleware structure has been completely redesigned. The proposed middleware presents modules related to the management of publishers, subscribers and brokers, detailed in the following. Containers represent data structures that store the data of the topics handled by each management module. A message buffer, implemented through the Queue Container, retains incoming messages before delivering them to the application layer.

The Middleware Application API presents primitives that can be used to design applications using the publish-subscribe programming model. The applications run at the application level. Section 4.3.5 presents an experiment where we evaluate three applications that use this API.

The Middleware Extension API presents primitives that can be used for development of middleware extensions modules (named "Middleware Extension" in Figure 4.12) using the publish-subscribe programming model. These extension modules run at the middleware level. Chapter 5 presents an implementation of a middleware extension which provides the support for the development of self-adaptive systems on the middleware.

The Hardware/Software Abstraction Layer (HSAL) presents a set of primitives that aims to facilitate the middleware portability in other platforms. It assures the middleware portability by the implementation of a specific driver for the target platform. The middleware code remains the same. The proposed HSAL has standardized functions to interface with the Kernel (i.e., to create, destroy, suspend, and resume system tasks), NoC (to write or read in the NoC interface) and MPSoC manager (i.e., to know if a node is the broker of the system). APPENDIX A describes the minimal list of primitives of the HSAL that need to be provided to incorporate the middleware to a given platform, including kernel and hardware modules.

The Kernel is responsible for features like low-level communication primitives, task scheduling, DMA, and treatment of software system call and hardware interruption. The middleware is responsible for features like publish-subscribe protocol management, publish-subscribe messages delivery guarantees (QoS Manager), and interface with application level, Distributed Services, and kernel level.

From the general description of the middleware architecture, we detail as follows each one of the OO-MQSoC components regarding the middleware's communication and their interfaces.

PublishersManager

The PublishersManager component has an interface that enables other architecture components to publish a message to a topic. Note that before a publisher component publishes a message to a topic, it must advertise the topic, as explained in Section 4.2.3. Figure 4.13 shows the class diagram of the PublishersManager component (**PublishersManager**).

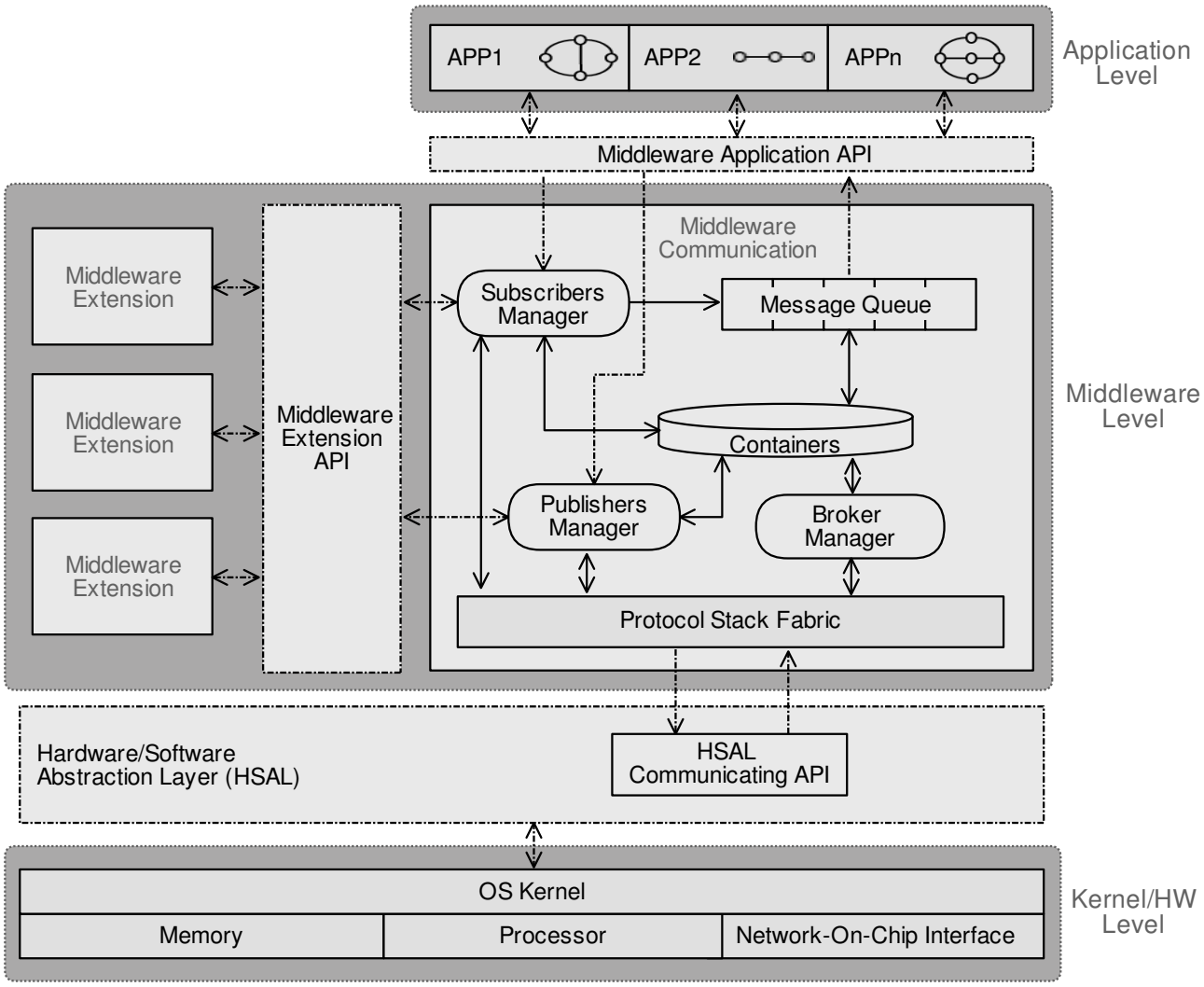


Figure 4.12: OO-MQSoC Architecture.

The other components of the middleware use the PublishersManager methods to advertise/unadvertise a topic and publish messages to a topic. The Publisher Manager also processes subscription and unsubscription requests came from Broker Manager (Section 4.3.4).

The table of publishers (**publishers**) contains a register for each element that has called the **advertiseHandler** method. The index of this table is the topic identification. The PublishersManager uses the table of publishers to manage the advertise message acknowledgment feature. Also, when a component calls the **unadvertiseHandler** method, the PublishersManager erases the respective register in the table of publishers. Only one publisher can advertise a given topic. Therefore, the **etl::flat_map** library implements the table of publishers.

The table of subscribers (**subscribers**) contains the set of subscriber clients that have requested subscription in the topics advertised by the publisher clients of the current node. The index of this table is the topic identification. External components call the **subscriptionHandler** method to request a subscription in a topic. When this happens, the PublishersManager inserts a new register in the table of subscribers. In the same way, the PublishersManager erases a register when receives an **unsubscriptionHandler** method calling. There may be more than one subscriber node for the same topic. In this way, the table of subscribers must allow the insertion of more than one register with the same index (topic), accomplished by the library etl::flat_multimap. The **publishHandler** method publishes a message in a topic encapsulating the given message payload and the specific message header for this type of message (see Figure 4.17). The **publishHandler**

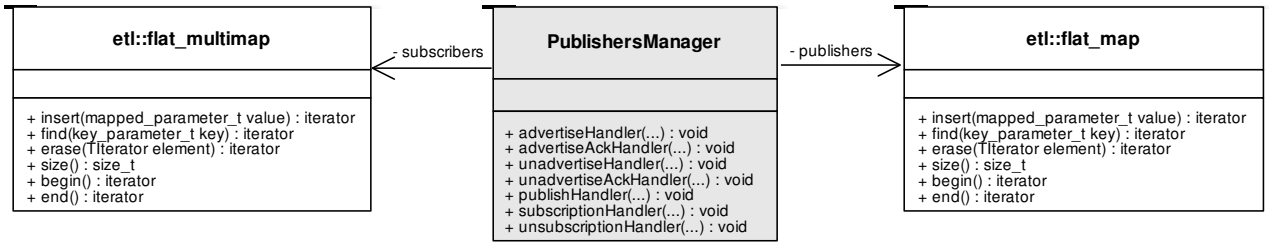


Figure 4.13: PublishersManager Component.

method searches by subscriber clients in the table of subscribers to know whom to send the message. The messages are sent directly to the subscriber node.

SubscribersManager

The SubscribersManager component has an interface that enables other architecture components to subscribe to a topic. Figure 4.14 shows the class diagram of the SubscribersManager component (**SubscribersManager**).

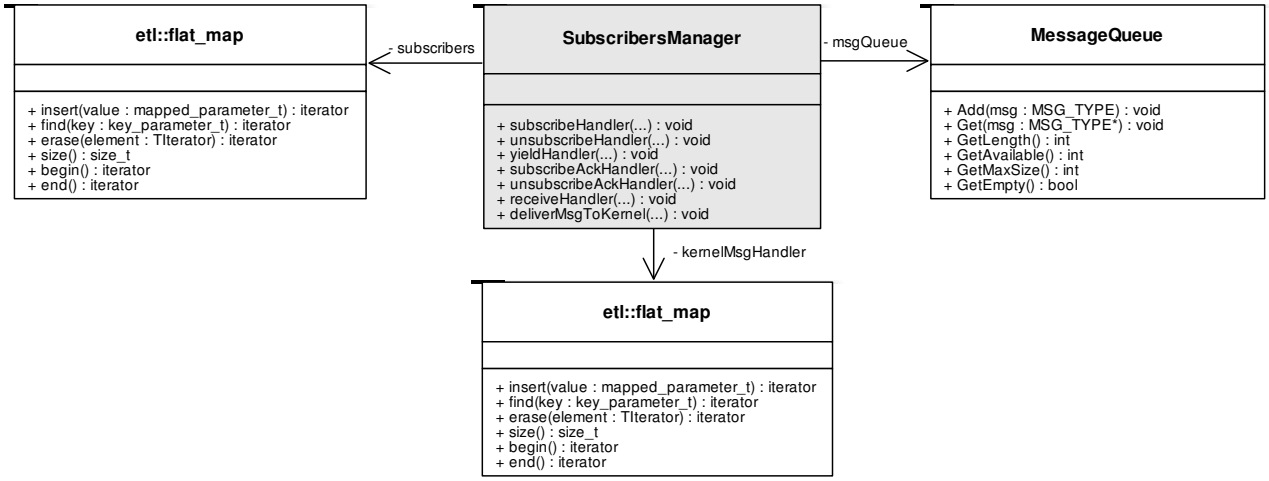


Figure 4.14: SubscribersManager Component.

The other components of the middleware use the SubscribersManager methods to subscribe/unsubscribe to a topic and to check for received messages. The SubscribersManager has methods to handle network interruption (NI) events generated by the **PSLayer** component. The **receiveHandler** method handles received messages in a subscribed topic. When the received message is addressed to a topic subscribed by an application task, the method stores the message in the **MessageQueue** component. When to a service running in the kernel level, the method calls the respective function callback to process the message. The **subscribeHandler** method stores in the table of Kernel Handlers (**kernelMsgHander**) the function callback informed by the kernel service in the method parameter. Only one subscriber kernel service can subscribe to a given topic. In this way, the **etl::flat_map** library implements the table of Kernel Handlers. The **subscribeAckHandler** and **unsubscribeAckHandler** methods handle messages acknowledgment for the subscription and unsubscription protocol phases. The **yieldHandler** method verifies if there is some message in the **MessageQueue** for a given topic and consumes them.

The table of subscribers (**subscribers**) contains a register for each element that has called the **subscribeHandler** method. The index of this table is the topic identification. The SubscribersManager uses the table of subscribers to manage the subscribe message acknowledgment feature.

Also, when a component calls the **unsubscribeHandler**, the respective register in the table of subscribers is erased. Only one subscriber can advertise a given topic. In this way, the **etl::flat_map** library implements the table of subscribers.

Broker Manager

The Broker Manager component has an interface that enables the synchronization of the topics in all the publish-subscribe system. Figure 4.15 shows the class diagram of the Broker Manager component (**Broker**). The Broker Manager methods interact with the methods of the PublishersManager and SubscribersManager components. The **advertiseHandler** and **unadvertiseHandler** methods perform inserting and erasing of topic registers in the table of the publishers (**publishers**). These methods are consequences of the calls to the **advertiseHandler** and **unadvertiseHander** methods of the PublishersManager component (Section 4.3.4). In the same way, the **subscribeHandler** and **unsubscribeHandler** methods perform inserting and erasing of topic registers in the table of the subscribers (**subscribers**). These methods are consequences of the calls to the **subscribeHandler** and **unsubscribeHander** methods of the SubscribersManager component (Section 4.3.4). The tables of publishers and subscribers can store topics of the same index. In this way, the **etl::flat_multimap** library implements both tables.

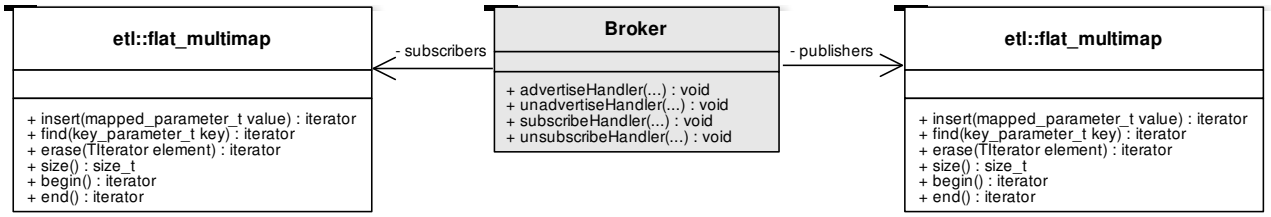


Figure 4.15: Broker Manager Component.

Protocol Stack Fabric

The Protocol Stack Fabric component composes the elements of the protocol stack used in the middleware. Figure 4.16 shows the class diagram of the Protocol Stack Fabric component. The **ProtocolStackFabric** class performs the external interface of the Protocol Stack Fabric component. In order to become the implementation of protocol layers more flexible, the Protocol Stack Fabric component uses three design patterns[12] to perform the protocol stack: Protocol Stack, Protocol Layer, and Protocol Packet. Two protocol layers are organized in adjacent layers (upper and lower), and there are no dependencies between them. In the future, new layers can be developed and added to the protocol stack without interventions at existent layers.

The Protocol Stack design pattern implemented by the **ProtocolStack** class maintains a doubly linked list of protocol layers. It provides an interface composed of the following methods: Transmit (invoked by an external element to send messages using the protocol stack); Receive (invoked by the kernel to pass received messages to the protocol stack); Add_Layer (invoked by the **ProtocolStackFabric** to add a protocol layer at a specific position in the protocol stack); Remove_Layer (invoked by the **ProtocolStackFabric** to remove a layer from the protocol stack).

The Protocol Layer design pattern implemented by the **ProtocolLayer** class aims to decouple adjacent protocol layers. It provides a standard interface for implementing different layers of a protocol stack. We implement two protocol layers in the middleware: **PSLayer** (Publish-Subscribe Layer) and **NetworkLayer**. Figure 4.17 shows the layers that compose the protocol stack and the

[12] https://www.eventhelix.com/RealtimeMantra/PatternCatalog/

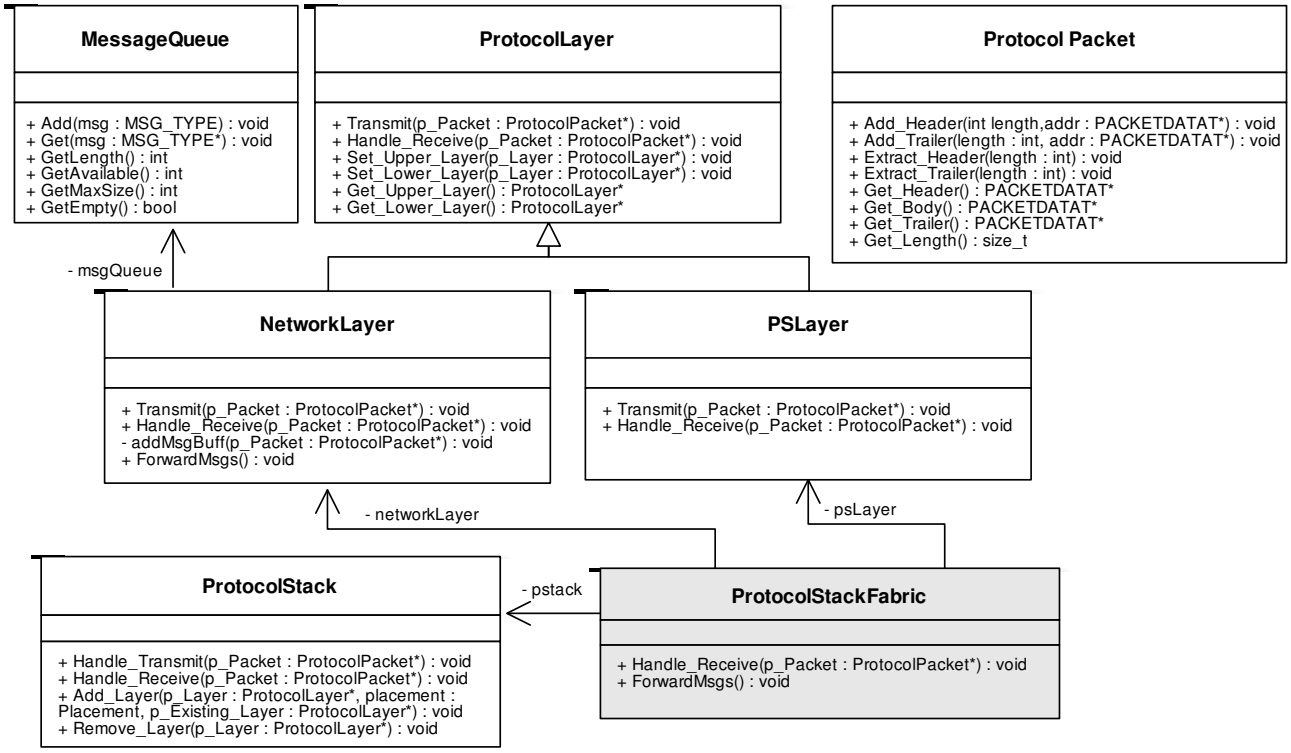


Figure 4.16: Protocol Stack Fabric Component.

format of the handled packet. The layers implement the virtual methods of the **ProtocolLayer** class: **Transmit** and **HandleReceive**. The **ProtocolStack** class invokes the **Transmit** method of a lower layer to pass a packet to the upper layer. The **ProtocolStack** class invokes the **HandleReceive** method of an upper layer to pass a packet to the upper layer.

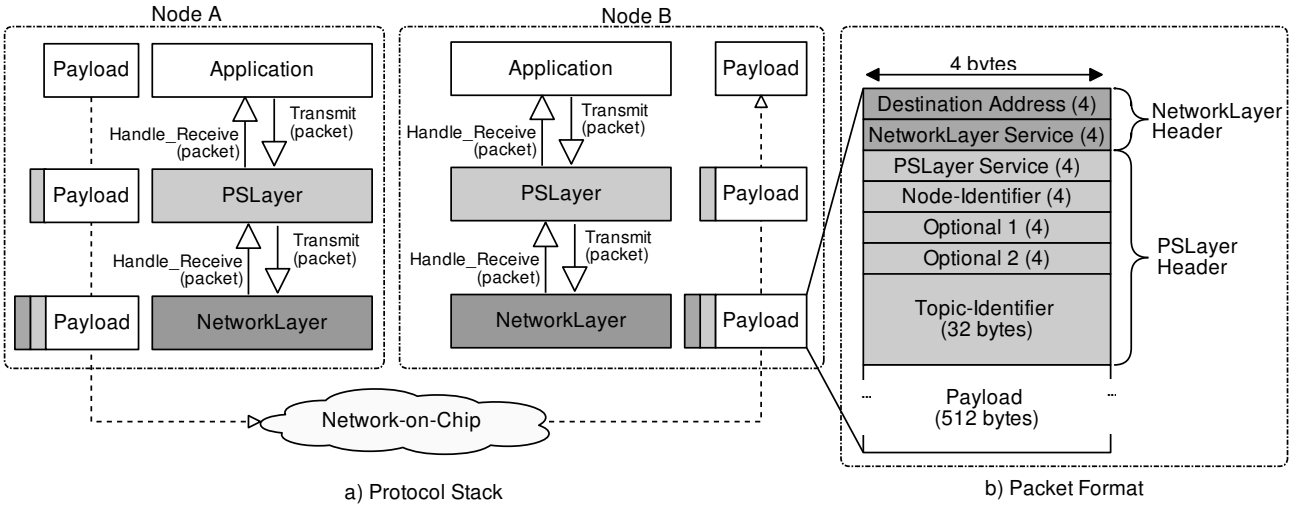


Figure 4.17: Protocol Stack and Packet Format.

The **PSLayer** class implements in the **Handle_Receive** method the network service handlers managed by the publish-subscribe protocol, one for each phase of the protocol. Both the PublishersManager, SubscribersManager and Broker Manager components of the middleware use the **Transmit** method of the **PSLayer** class to send messages through the publish-subscribe protocol. In the same way, the service handlers implemented in the **Handle_Receive** method call the respective methods in those components.

The Protocol Packet design pattern implemented by the **ProtocolPacket** class handles the packet encapsulation, providing an interface for adding and removing of headers and trailers in the packet. The packet size is updated whenever headers and trailers are added or removed. The **ProtocolPacket** class contains a raw buffer that is dynamically handled across the layers. The goal is to reduce the overhead that could be occasioned by buffer copies in each layer. Instead, the packet buffer location remains the same throughout the protocol stack, and only the header, body, and trailer pointers are manipulated while the packet is forwarded between the layers. We do not use the trailer region to implement the Publish-Subscribe header. Figure 4.17-b shows the packet format used by the Publish-Subscribe Layer. The first four bytes comprise the physical address of the destination node. The kernel defines the physical address from a given network address. The PUB-SUB Service field identifies the message type for multiplexing service handler purposes. The Node-Identifier comprises the identification of the source node. The Optional 1 and 2 are additional fields used by the service handlers. The Topic-Identifier field identifies the publish-subscribe topic of the message. The size of this field is parameterizable at design-time. As detailed in Section 5.3, the topic is represented as a string, where each character occupies 1 byte. Therefore, the size of this field defines the maximum number of characters of the topic. The following 512 bytes (configurable size) represent the load region encapsulated by the serialization process.

Figure 4.18 shows a sequence diagram for the receiving of a packet in the middleware protocol stack. When receiving, an external component (e.g. Kernel) calls the Handle_Receive method of the ProtocolStackFabric class passing the packet as parameter (1). The ProtocolStack-Fabric class calls the Handle_Receive method of the ProtocolStack class (2), which checks for the lowest layer of the protocol stack (3) and forward the packet for the NetworkLayer class calling the Handle_Receive method (4). The Handle_Receive method extracts its respective header and checks who is the following layer to forward the packet by checking the NetworkLayer Service field of the header (5). In this example, the message is addressed for the PSLayer, who receives the message through its Handle_Receive method (6). The PSLayer class extracts the PSLayer header and verifies who is the service handler to deliver the body message by checking the PSLayer Service field of the header (7). In this example, the body message is addressed for the AdvertiseHandler method of the Broker class (8), who receives the body message and process it (9).

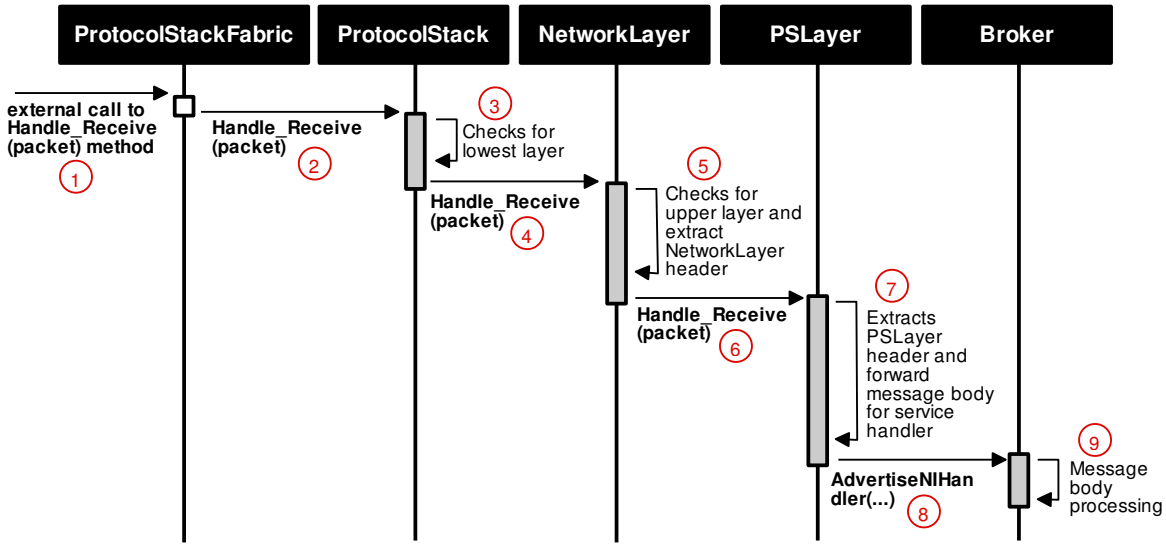


Figure 4.18: Receiving a packet in the Protocol Stack used in the Middleware.

Figure 4.17-b shows the Packet Format. The **PSLayer Service** field of a PSLayer packet contains the type of the message, corresponding to the publish-subscribe protocol phases detailed in Section 4.2.3. The PSLayer Class implements the service handlers for each message type, that

are: *BrokerAdvertise*, *PublisherAdvertiseAck*, *BrokerUnadvertise*, *PublisherUnadvertiseAck*, *Brokersubscription*, *SubscriberSubscriptionAck*, *BrokerUnsubscription*, *SubscriberUnsubscriptionAck*, and *SubscriberRcv*. This field is checked always that the PSLayer receives a packet (step 7 in Figure 4.18. Both the *Node-Identifier* and *Topic-Identifier* header fields are used by all types of messages. However, the *Optional 1* and *Optional 2* header fields differ according to the message type. Each message type is associated to a phase of the publish-subscribe protocol. Next, we describe the characteristics of each message type in turn:

BrokerAdvertise - When a publisher node advertises a topic, the *advertiseHandler* method in the *PublshersManager* class generates a *BrokerAdvertise* message that is forwarded to the respective broker node and processed by the *advertiseHandler* method in the *Broker* class.

PublisherAdvertiseAck - To acknowledge the receiving of the *BrokerAdvertise* message, the *advertiseHandler* method in the *Broker* class generates a *PublisherAdvertiseAck* message that is processed by the *advertiseAckHandler* method in the *PublishersManager* class.

BrokerUnadvertise - When a publisher node unadvertises a topic, *unadvertiseHandler* method in the *PublshersManager* class generates a *BrokerUnadvertise* message that is forwarded to the respective broker node and processed by the *unadvertiseHandler* method in the *Broker* class.

PublisherUnadvertiseAck - To acknowledge the receiving of the *BrokerUnadvertise* message, the *unadvertiseHandler* method in the *Broker* class generates a *PublisherUnadvertiseAck* message that is processed by the *unadvertiseAckHandler* method in the *PublishersManager* class.

BrokerSubscription - When a subscriber node subscribes to a topic, the *subscribeHandler* method in the *SubscribersManager* class generates a *BrokerSubscription* message that is forwarded to the respective broker node and processed by the *subscribeHandler* method in the *Broker* class.

SubscriberSubscriptionAck - To acknowledge the receiving of the *BrokerSubscription* message, the *subscribeHandler* method in the *Broker* class generates a *SubscriberSubscriptionAck* message that is processed by the *subscribeAckHandler* method in the *SubscribersManager* class.

BrokerUnsubscription - When a subscriber node unsubscribes a topic, the *unsubscribeHandler* method in the *SubscribersManager* class generates a *BrokerUnsubscription* message that is forwarded to the respective broker node and processed by the *unsubscribeHandler* method in the *Broker* class.

SubscriberUnsubscriptionAck - To acknowledge the receiving of the *BrokerUnsubscription* message, the *unsubscribeHandler* method in the *Broker* class generates a *SubscriberUnsubscriptionAck* message that is processed by the *unsubscribeAckHandler* method in the *SubscribersManager* class.

Figure 4.19 shows the sequence diagram for the transmission of a packet in the middleware protocol stack. A component must call the *Transmit* method of the *PSLayer* class to send a packet (1). The *PSLayer* class adds the PSLayer header using the parameters passed in the method call, checks for the lower layer (2) and forwards the packet to it calling the respective *Transmit* method (3). The *NetworkLayer* class, that is the lower layer of the *PSLayer* class, adds the NetworkLayer header (4) and send the packet (5) using the *HSAL_OS_COMM_send* function of the Hardware/Software Abstraction Layer. The *HSAL_OS_COMM_send* function implements the low level primitive to send the packet through the NoC.

4.3.5. Evaluation

We present in this section a middleware comparison contrasting the proposed middleware and the previous middleware implementation.

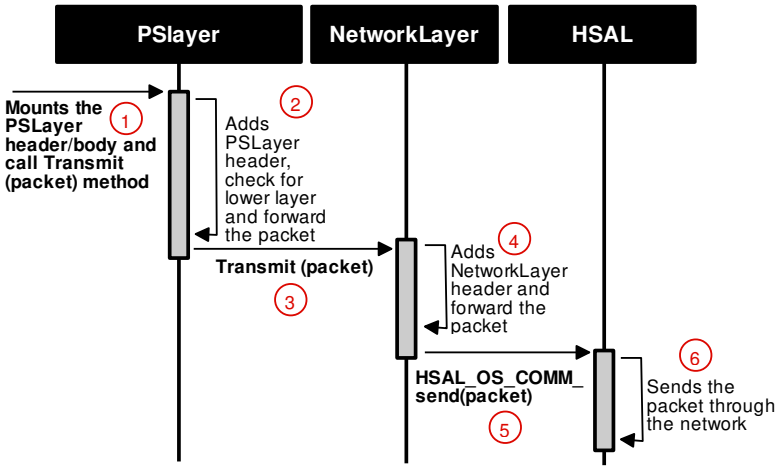


Figure 4.19: Transmitting a packet in the Protocol Stack used in the Middleware.

Experimental Setup

In order to compare the proposed middleware with the previous middleware implementation, we have used a benchmark composed of the MPEG, Producer-Consumer (PROD-CONS) and DTW (Dynamic Time Warping) applications. Figure 4.20 shows these applications represented through a task graph using the publish-subscribe programming model. A directed arrow between two tasks (blocks in the figure) means that the first task sends data to the second one. A message encapsulates the data transferred between two tasks with a maximum payload size of 512 bytes in the experiments. The number of exchanged messages by an application depends on the workload data configured at compile time, in the test scenario.

We compare two equivalent implementations of the middleware: a previous middleware implemented using C programming language (detailed in Section 4.2); and the proposed C++ based middleware (respectively C MIDD and C++ MIDD in Figure 4.21). Both implementations use the same base platform and programming model. The evaluated key metrics are memory footprint and application execution time. The used compilers are the *arm-none-eabi-gcc* and *arm-none-eabi-g++*, version 4.9.3. All scenarios use a single-cluster 4x4 MPSoC, with each PE executing a single task. Tasks are mapped in the same PE in both C and C++ evaluation scenarios.

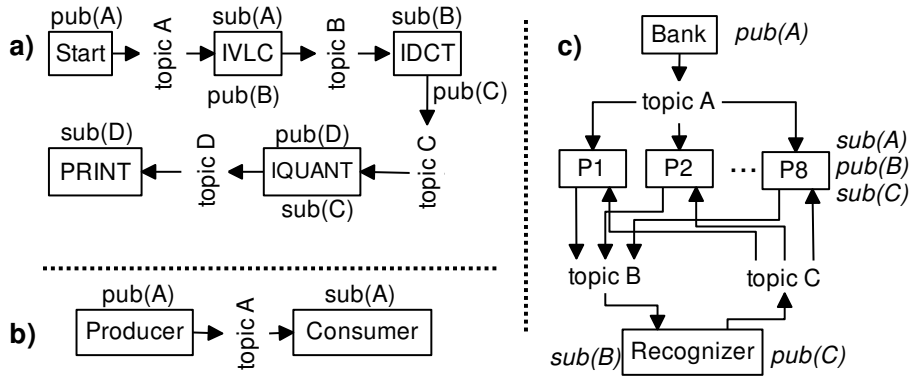


Figure 4.20: Task graph of the a) MPEG, b) PROD-CONS, and c) DTW applications.

Results

The first experiment evaluates the achieved memory footprint size (kernel + middleware) of the C++ middleware implementation by using each technique and best practices for embedded

C++, as presented in Section 4.3.3. Table 4.3 shows the footprint size of the initial version with no optimization (first line) and the footprint size achieved after using each technique, in the order in which they appear in the table. The total memory footprint size achieved in the final version of C++ the middleware is 17.03KB, being 11.06KB related to the kernel size and 5.97KB related to middleware size. Compared to the C-based middleware detailed in Section 4.2 which has been recompiled using the same features present in the C++ middleware, the proposed C++ middleware represents an overhead of 1.37 KB (8.7%).

Table 4.3: Total Memory Footprint (Kernel+Middleware) improvement

Version	**Footprint (KB)**	**Reduction**
No optimization (INITIAL VERSION)	106	-
+ Using "-Os"	75.29	29%
+ Using "-fno-exceptions"	75.01	0.4%
+ Replacing Map/Multimap by Flat-map/Flat-Multimap	72.32	3.6%
+ Using "Placement new" (FINAL VERSION)	17.03	76.4%

The second experiment evaluates the application execution time measured through a timing model [AMR+16] that capture the executed instructions for each processor, generating an execution time from total executed instructions. For this experiment, we use the final version (last line in Tab. 4.3) of the C++ middleware, which incorporates all cited optimizations. The execution time of both DTW, MPEG and PROD-CONS applications were evaluated ranging the workload data from 8 to 64 packets. Figure 4.21 shows the results for each scenario. In all the simulated scenarios the C++ middleware presents better application execution time. Figure 4.21 also shows the percentage of gain obtained in the C++ middleware implementation compared with the C implementation (minor graph). It reduces the execution time ranging from 4% to 13.4% in DTW, from 3.4% to 19.5% in PROD-CONS, and from 3% to 6.7% in MPEG, depending on the workload. The gain decreases when the workload increases because the highest gain is obtained on topic advertise/subscribe steps, which occurs at the beginning of the application execution. In these steps, the use of containers reduces the time spent with the insertion and search of objects in the managed topic tables.

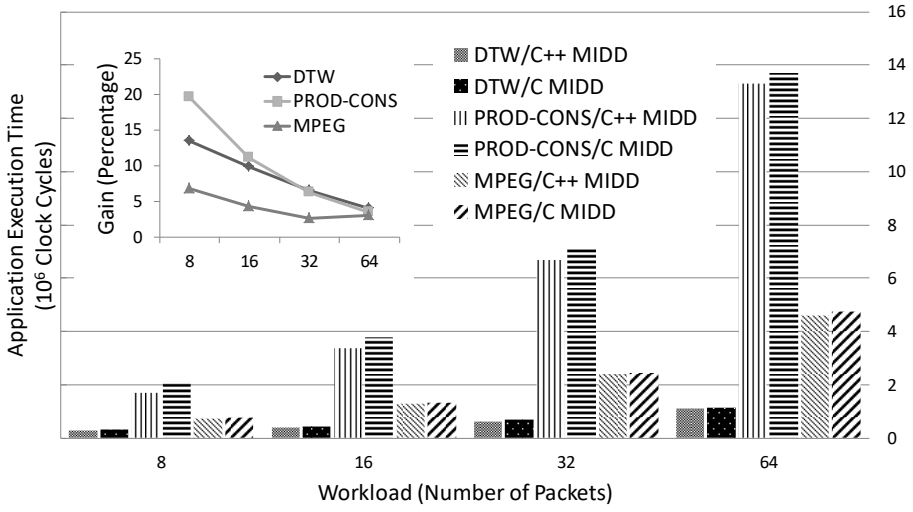


Figure 4.21: Application execution time for C and C++ scenarios.

The performed experiments demonstrate that the proposed middleware is well tailored to MPSoC domain since it presents low memory usage and improved applications execution time.

4.4. Serialization/Deserialization into the Middleware

This section describes a key additional feature of the middleware that is the incorporation of the serialization/deserialization feature into the middleware in order to ease the encapsulation of data transferred between components of self-adaptive services. This new feature has been incorporated to the previous object-oriented middleware presented in Section 4.3. The contributions detailed in this section have been published in part at ICECS'18 [HDFGM18]. Section 4.4.1 explains the motivation of using the serialization-deserialization feature into the middleware design. Section 4.4.2 presents related works. Section 4.4.3 presents the list of serialization libraries and discusses some of their applicability in the context of this work. Section 4.4.4 presents the experimental setup used to compare the evaluated serialization libraries. Section 4.4.5 presents the results and discussion.

4.4.1. Motivation

In MPSoCs, the memory organization is usually based on the No Remote Memory Access (NoRMA) approach, where the embedded processors indirectly access the remote address via messages sent via a NoC. As any other distributed system based on NoRMA, the communication infrastructure and protocol stack play an essential role in the system architecture.

Although the community widely debated on-chip physical communication infrastructure in the past [BM06], discussions about how to build a protocol stack for MPSoCs are rarer to be found. Regardless of the type of distributed domain, in typical protocol stacks, each of the layers has a set of protocols, which provide different services to the upper layer. The uppermost layer is the application layer while the lowermost one is the physical layer. Layers in between physical and application layers may vary from system to system. For instance, the OSI model suggests the implementation of five other layers: data link, network, transport, session, and presentation.

Despite the many different typical services for a protocol stack, serialization service is one of the most common among them. Two approaches for serialization are commonly found in the literature: binary and textual. In both, some serialization service is responsible for transforming some applications data structure into serial data, which can be either a stream of bytes (binary serialization) or merely a string (e.g., XML, JSON). Deserialization is the reverse process, where a stream of bytes (or a string) is received and converted to a copy of the data structure that originated it. It is important to note that serialization and deserialization are often called *serialization*, for short. We use this terminology for the rest of this Thesis.

There are dozens of serialization libraries available. However, these libraries are designed for a particular application domain that has requirements to be met. For example, application domains related to mobile phones they require interoperability, low power consumption, among others. Another example is web applications where, usually, textual serialization formats (e.g., XML, YAML, and JSON) are preferred. Thus, the approach for serialization methods is bound to the domain of application.

In MPSoCs, where the amount of memory available for each PE is minimal (about tens or hundreds of KBytes), some programming platforms (e.g., Java, Python) cannot be used. It also excludes approaches that rely on large external dependencies (e.g., LibBoost). Besides, serialized

data must be as smaller as possible to save bandwidth in the NoC, reducing network contention and energy consumption. For these reasons, string-based serialization (e.g., XML, JSON) are not valid options. Lastly, the serialization and deserialization must be fast, because the MPSoCs are usually based on small and simple processors. For instance, such processors work with 32-bit data path, fixed-point arithmetic, and three or five pipeline stages. Lastly, some applications have real-time constraints, although we do not address them in this Thesis.

The requirements for MPSoC excludes a substantial number of available serialization methods. We present an evaluation of the performance and resource usage of the few adequate solutions we could find, against the requirements mentioned before. As given in Section 4.4.2, only a few studies present similar comparisons, but the solutions they evaluate have no use for MPSoCs since the domains of applications are radically different and less constrained when compared to the MPSoC domain.

4.4.2. Related Works

Although some studies on the comparison of serialization libraries exist in the literature, none of them are focused on serialization for highly memory constrained embedded platforms such as MPSoCs. The most related works evaluates serialization with a focus on the Internet of Things domain [PBYP17]. However, they assume the Beagle Bone Black (https://beagleboard.org/black) or Odroid (https://www.hardkernel.com) hardware platforms. Both platforms have more than 512MB of memory and use a complete Linux-based OS, which has much more resources than the individual processors of our target MPSoC architecture. Thus, most of the evaluated serialization methods cannot be applied for MPSoCs because, for instance, the evaluated methods require from 1MB to 22MB of memory, which is more memory than the total amount of memory available for individual processors in the target MPSoC platform.

Maeda [Mae11, Mae12] presented similar studies comparing several serialization libraries for Java. The requirement of languages such as Java and Python is also a limitation for processors with few KBytes of memory. Also, most of the serializers work with textual formats such as XML, YAML, and JSON. Even though these formats present advantages concerning readability and interoperability, they present the drawback of bigger serialized data size compared to the binary data formats. In the MPSoC domain, bandwidth usage in the NoC is much more important than readability. For this reason, binary serialization is more suitable for MPSoC.

Sumaray and Makki [SM12] present a similar comparison for data size, serialization speed and ease of use of serialization libraries. However, they focus on Android-based platforms, which fall on the same issues as the previous references.

Our main contribution is to present an experiment-based comparison of serialization libraries applied to the context of embedded platform highly constrained in memory.

4.4.3. Serialization Libraries

In programming languages such as Java and C#, serialization is implemented by extending some interface of the built-in API. In other programming languages, such as C and C++, serialization must be implemented almost from scratch. The community developed several libraries, in the hope of mitigating efforts during the implementation of serialization.

Most of the libraries support both serializations of basic types and complex types. For the later, the support mostly is provided through schemas, which approach varies from library to library. Depending on the approach, the required resources may go over the available for resource constrained platforms. In our comparison we considered the following libraries.

MsgPack-c (version 2.1.5) is an implementation of the MsgPack (msgpack.org) serialization format with support to C and C++ language. MsgPack-c requires both a C++03 or C++11 compatible compiler and code annotation. MsgPack-c is available at github.com/msgpack/msgpack-c.

MsgPuck (v. 2.0) is a compact implementation of MsgPack library, written in C. MsgPuck repository announces interesting characteristics like zero-cost abstractions and zero overhead. All necessary library code is written in a pair of .c and .h files. In addition to support the base types, MsgPuck also has support to *arrays* as representation of a sequence of objects and *maps* for key-value pairs of objects. MsgPuck requires a C89+ or C++03 compatible compiler and it does not use schemas or code annotation. Available at github.com/rtsisyk/msgpuck.

MPack (v. 0.8.2) is a third implementation of MsgPack format, also written in C, without libc requirement. MPack does not use schemas or code annotation. Available at github.com/ludocode/mpack.

FlatBuffers (v. 1.8.0) allows that the data can be accessed without unpacking serialized data. But this features does not come without a cost, as we show in the results discussed in Section 4.4.5. Flatbuffers is based on Protocol Buffer format (github.com/google/protobuf), also known as ProtoBuf. FlatBuffers requires both a C++11 compatible compiler and schemas definition. Available at github.com/google/flatbuffers.

NanoPB (v. 0.3.9) is an implementation of ProtoBuf that targets embedded systems. The serialization rely on schemas. Schemas of NanoPB are written into proto files, and their syntax is very similar to C's struct syntax. Nanopb should compile with most ansi-C compatible compilers, but it requires implementations of the strlen, memcpy and memset functions. Available at github.com/nanopb.

YAS (v. 5.0.1) is a replacement for Boost Serialization library (www.boost.org). Advantages of YAS include it is header-only library and does not depends on external libraries and endianess. It requires both a C++11 compatible compiler and schemas definition. Available at github.com/niXman/yas.

4.4.4. Evaluation

We perform the experiments in FreeRTOS-based MPSoC Plataform detailed in Section 4.1. The network infrastructure counts with a three-layer protocol stack that interacts with the underlying NoC by system calls to the kernel. In the first layer resides application-level protocols. These protocols are supported by a middleware implemented over the publish-subscribe programming model.

Application Case Study

The experiments use a Producer-Consumer application, which contains two tasks: a producer and a consumer. The pub-sub middleware is used to perform the communication between the two tasks. A topic identifies the message flow. We use three distinct data types that represent the application data in each test scenario. The goal is to evaluate each serialization library using from straightforward to more complex data types. Figure 4.22 shows the used data types. The size of

the structs A, B and C is, respectively, 8, 60 and 92 bytes, not considering the size of types that are dynamically sized. In our experiments, the size of all vectors is 1, that is, each vector has only 1 inserted element. The application data is delivered to the middleware level by using the provided API [HAR+17]. The middleware serializes the application data using a serialization library and transfers the serialized payload to the low level of the protocol stack until it is transmitted over the NoC. One message is sent by producer to consumer task in all scenarios.

```
/* Struct A */
struct Temperature{
  int32_t timestamp;
  float temp;};

/* Struct B */
struct InstrCnt{
  char[32] name;
  int32_t arith;
  int32_t logical;
  int32_t branch;
  int32_t jump;
  int32_t load;
  int32_t store;
  int32_t nop;};

enum TempLevel{High=1,Medium,Low};

/* Struct C */
struct AllSensors{
std::string name;
Temperature temp;
float calib;
int16_t processor_usage;
int16_t processor_usage;
std::vector<uint8_t> occupancy;
TempLevel  tempLevel;
std::vector<InstrCnt> processors;
InstrCnt instrCnt;
std::vector<Temperature> History;};
```

Figure 4.22: Data structures used within the experiment.

Metrics

Three metrics are extracted from each scenario: serialization and deserialization execution time, data size, and code size. A scenario is a combination of a serialization library and one of the three structures presented in Figure 4.22. Thus, a total of 16 scenarios were run, since YAS and NanoPB do not support the serialization of vectors, which are used in *Struct C*. Although the MsgPuck and MPack libraries do not directly support vectors, the available API allows you to serialize the vectors as object arrays and include them in the serialized data.

The **serialization execution time** corresponds to the total time spent by the producer process to perform the serialization of data. The **deserialization execution time** is the time spent by the consumer process to perform the deserialization of the received data. The measurement unit is the number of *clock cycles* measured through a timing model [ROR+14] that capture the executed instructions for each processor, generating an execution time from total executed instructions. The **data size** (DS) corresponds to the number of bytes required to encapsulate the serialized object into the packet payload, that is, the total size of the payload data of the application layer. The **code size** (CS) corresponds to the amount of memory required in PE to store the software code. This metric considers the size of kernel, middleware and serialization library, together. The measurement unit is *bytes*. For comparison purposes, the size of the software code (kernel plus middleware) without any serialization library is 22.5KB.

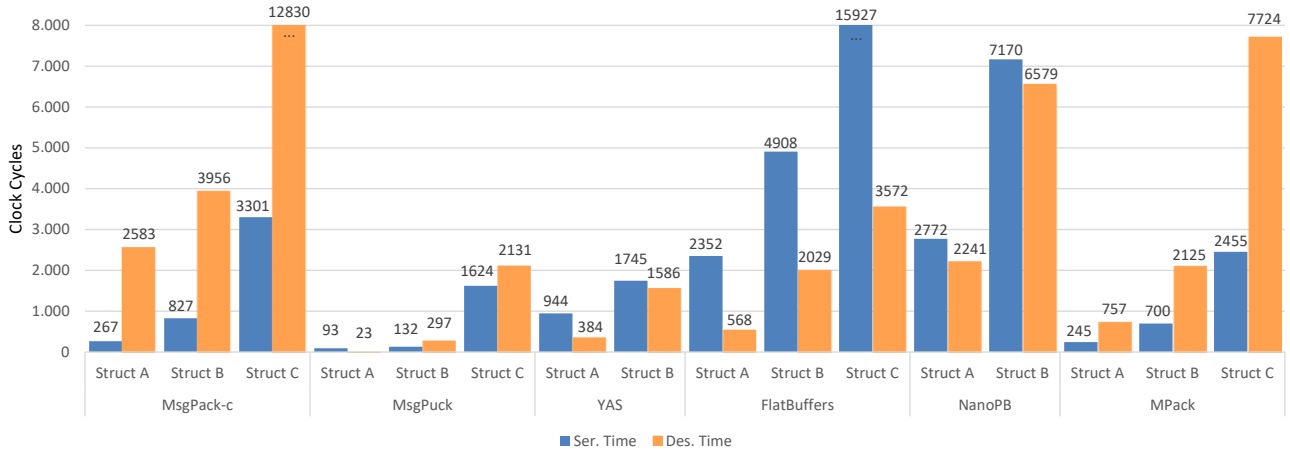


Figure 4.23: Serialization and deserialization execution time for each evaluated library and serialized data structs.

4.4.5. Results

The performed experiment evaluates memory code size (CS) and payload data size (DS) achieved at each data struct type. Table 4.4 shows the results. Regarding CS and DS, the MsgPuck library has achieved the fittest result for all three data structures. All libraries present a larger code size for *Struct C* because this struct has elements with standard types to represent vectors and strings (*std::vector* and *std::string*). Consequently, the code size is significantly increased with additional methods to handle these types. An alternative to representing these object types would be the use of specialized libraries for embedded system, such as Embedded Template Library (http://www.etlcpp.com). FlatBuffers library needs a larger number of bytes to represent the serialized data, in addition to presenting the largest code size. FlatBuffers stores metadata of complex types into memory in a way that it serves as pointers to parts of the serialized data. In general, libraries that use schemas to represent data structures end up producing a larger code size. They use a run-time type identification (RTTI), which is a feature of the C++ programming language that exposes information about an object's data type at runtime. On the other hand, the libraries with smaller generated code size are those that require the explicit definition of the serialize/deserialize method for each object that composes the struct. We observe a trade-off between ease of use of the library and the amount of memory necessary to store the software. The system designer must keep this in mind when choosing the serialize library that fits into your design.

In order to demonstrate an example of ease of use, we show the code snippet necessary to serialize the *Struct B* in both the YAS library (Figure 4.24), that uses schemas, and the MsgPuck library (Figure 4.25), that requires the explicit declaration of serialization process for each struct element.

We observed that some serialization libraries (MsgPack-c, MsgPuck and MPack) consume more clock cycles for deserialization than serialization. In the Producer-Consumer application, specifically, this is very undesired, since, if the producer process is faster than the consumer, there will be a point in application's lifetime in which the consumer buffer will be full, and thus the producer will be unable to produce and deliver more packets. This behavior may slow the system, which is not tolerable in some domains (e.g. real-time applications). When the serialization time is greater than the deserialization time, the system is also slowed, but the node that hosts the consumer application will not be compromised in case it has other tasks to care of.

Table 4.4: Memory size for each analyzed library.

Library	Struct A		Struct B		Struct C	
	CS[1]	DS[2]	CS[1]	DS[2]	CS[1]	DS[2]
MsgPack-c	168.7	9	269.1	15	272.6	67
MsgPuck	22.7	8	23	14	263.2	62
YAS	328.9	15	329.7	49	N/S[3]	N/S[3]
Flatbuffers	333.6	24	334	72	336	224
NanoPB	33.6	10	33.7	24	N/S[3]	N/S[3]
MPack	34.8	9	35	15	272.3	67

[1] CS = Memory Code Size (in KBytes)
[2] DS = Data Size of the message payload (in Bytes)
[3] N/S = No support for vector of structs

```
yas::mem_ostream os;
yas::binary_oarchive<yas::mem_ostream> oa(os);
oa & AppData;
this->message.msg_len = os.get_intrusive_buffer().size;
this->message.msg = (char*) os.get_intrusive_buffer().data;
-----------------------------------------------------------
template<typename Ar>
void serialize(Ar &ar, const AppDataClass &t) {
ar & YAS_OBJECT_NVP( "InstCnt", ("n",t.InstrCnt.name),
("a",t.InstrCnt.arith), ("l",t.InstrCnt.logical),
"b",t.InstrCnt.branch), ("j",t.InstrCnt.jump),
("ld",t.InstrCnt.load), ("s",t.InstrCnt.store),
("n",t.InstrCnt.nop)); }
```

Figure 4.24: Code snippet of the YAS serialization process and required schema. AppData object contains an InstrCnt member corresponding to the Struct B.

```
char buf[MAX_PAYLOAD_SIZE];
char *w = buf;
w = mp_encode_str(w, AppData.InstrCnt.name,
strlen(AppData.InstrCnt.name));
w = mp_encode_int(w, AppData.InstrCnt.arith);
w = mp_encode_int(w, AppData.InstrCnt.logical);
w = mp_encode_int(w, AppData.InstrCnt.branch);
w = mp_encode_int(w, AppData.InstrCnt.jump);
w = mp_encode_int(w, AppData.InstrCnt.load);
w = mp_encode_int(w, AppData.InstrCnt.store);
w = mp_encode_int(w, AppData.InstrCnt.nop);
this->message.msg_len = strlen(buf);
this->message.msg = buf;
```

Figure 4.25: Code snippet of MsgPuck serialization process for the Struct B.

5. MIDDLEWARE EXTENSION FOR SELF-ADAPTIVE SYSTEMS

This chapter presents extension of the middleware that provides support for the development of self-adaptive systems in MPSoC platforms. Section 5.1 describes the model of self-adaptive system that we are proposing. Section 5.2 presents the extension of the middleware that implements the proposed model providing the support for development self-adaptive systems. A set of classes and features comprises this extension, called *Modules*, separating the functional logic of the self-adaptive system from the adaptive logic. Functional logic refers to the composition of the basic components that orchestrate the communication between the elements of the self-adaptive system. We have already detailed in Section 4.3.4 the functional logic of the middleware. Adaptive logic refers to the final logic for which the adaptive system is developed, that is related to the system adaptability, without concern with aspects of communication. Additionally, we present in Section 5.3 the topic name scheme used in the Modules extension. Finally, in Section 5.4, we present the basic guidelines for creating the objects of a self-adaptive system, together with a tool to automate this process.

5.1. Self-Adaptive System Model

A self-adaptive system is modeled in our proposal following the Observe-Decide-Act loop computing paradigm (see Section 2.4). We describe a self-adaptive system (SAS) as the set $SAS=\{ASE_1, ASE_2, ..., ASE_n\}$, where $ASE=\{S, M, D, A, E, T, SM, AE\}$ is the adaptive service set. In the set ASE, $S=\{s_1, s_2, ..., s_n\}$ is the sensor set, $M=\{m_1, m_2, ..., m_n\}$ is the monitor set, $D=\{d_1, d_2, ..., d_n\}$ is the decisor[1] set, $A=\{a_1, a_2, ..., a_n\}$ is the actuator set, $E=\{e_1, e_2, ..., e_n\}$ is the effector set, and $T=\{t_1, t_2, ..., t_n\}$ is the topic set. The set $SM=\{\{s_i,m_i,t_i\}, \{s_j,m_j,t_j\}, ..., \{s_n,m_n,t_n\}\}$ represents the relations between the sensor/monitor pairs (e.g. $\{s_i,m_i\}$) identified by a given topic (e.g. $\{t_i\}$). The set $AE=\{\{a_i,e_i,t_i\}, \{a_j,e_j,t_j\}, ..., \{a_n,e_n,t_n\}\}$ represents the relations between the actuator/effector pairs (e.g. $\{s_i,m_i\}$) identified by a given topic (e.g. $\{t_i\}$). Each decisor $d_i=\{M, A\}$ has your own set of monitors M and actuators A. This means that a decisor composites one or more monitors and actuators. A decisor has no topic identification since the programming model abstract it. A self-adaptive system composes one or more adaptive services. Each adaptive service performs adaptation for a desired objective[2]. Figure 5.1 shows two examples of Self-Adaptive System models.

In the proposed model, the set of sensors (S) and effectors (E) makes the interface with local observable and configurable resources (hardware or software) in the MPSoC platform. The set of monitors (M) and actuators (A) are the respective components that a decisor (placed at any place of the architecture) uses to communicate with sensors and effectors.

5.2. Middleware Extension: Modules

The Modules is an extension of the middleware that implements the Self-Adaptive System model described in Section 5.1. It provides a way for the development of adaptive services composed of software and/or hardware elements distributed by the network-on-chip composed a set of sensors,

[1] To simplify, we call the decision-making component as "decisor".

[2] Additional mechanisms not included in the model should address issues of multi-objective adaptability or when there are distributed decision-making (more than one decisor for the same set of actuators).

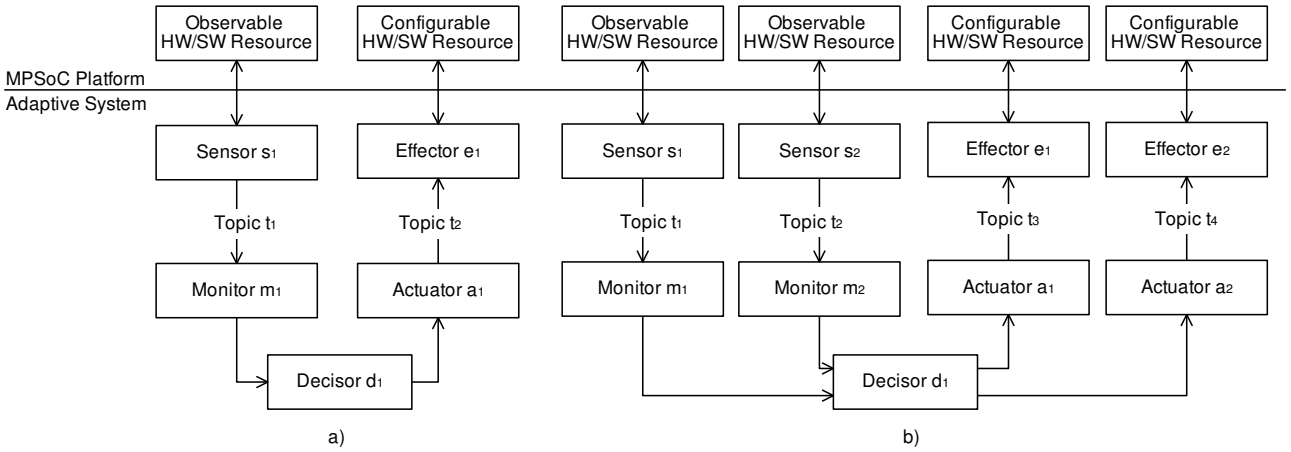


Figure 5.1: Examples of self-adaptive system models: a) $SAS=\{S=\{s_1\}, M=\{m_1\}, D=\{d_1=\{m_1,a_1\}\}, A=\{a_1\}, E=\{e_1\}, T=\{t_1, t_2\}, SM=\{\{s_1,m_1,t_1\}\}, AE=\{\{a_1,e_1,t_2\}\}\}$; b) $SAS=\{S=\{s_1,s_2\}, M=\{m_1,m_2\}, D=\{d_1=\{\{m_1,m_2\},\{a_1,a_2\}\}\}, A=\{a_1,a_2\}, E=\{e_1,e_2\}, T=\{t_1,t_2,t_3,t_4\}, SM=\{\{s_1,m_1,t_1\}\}, \{\{s_2,m_2,t_2\}\}, AM=\{\{a_1,e_1,t_3\}\},\{\{a_2,e_2,t_4\}\}\}$

decision-making methods, and actuators. Note that it is not the purpose of this Thesis to provide a set of adaptive services or a set of sensors, decision-making methods, and actuators. Nevertheless, we provide a middleware to make developing self-adaptive systems easier, more extensible, easier to maintain and decoupled from kernel software and the underlying hardware level (making it easily portable). A case study of a self-adaptive system using this middleware extension is presented in Chapter 6.

Figure 5.2 shows the OO-MQSoC architecture enhanced with the extension Modules, represented in the figure by the component named "Modules". The Modules component presents a set of classes and object-oriented techniques that applied together give support for the development of adaptive services. The set of classes are designed following an object-oriented paradigm and implements the five components of the Self-Adaptive System model: Sensor, Monitor, Decisor, Actuator and Effector. The publish-subscribe paradigm is explored to perform the communication between the communicating pairs. A topic identifies the communication performed by a communicating pair. As detailed in Section 5.1, the communicating pair can be a pair of sensors and monitors or a pair of actuators and effectors. Note that there is two types of components named monitors and effectors. A monitor is a component that subscribes to data from a sensor at the publish-subscribe system. That is, for every sensor component, there must be a monitor component. The same happens for the effector, which is a component that subscribes to the data of an actuator. Different topics identify the communication between the sensor/monitor pair and actuator/effector pair. A decisor that wants to receive data from a given sensor, it must instance the respective monitor component and then use the sensor data as a local variable of the monitor object. To apply a given system configuration, a decisor must instantiate the respective actuator component, store the decision in a local variable of the actuator object and call a primitive that will trigger the actuation over the NoC via publish-subscribe protocol. A decisor can instantiate one or more monitors and actuators (Figure 5.1-b). The remaining logic of the publish-subscribe protocol is performed by middleware. We detail this mechanism in the following sections.

The main advantage of using Modules is that the user[3] does not need to know details of the communication primitives of the middleware communication API. Instead, the user handles the ODA components as local variables. The base classes of the Modules component perform all the underlying communication with the lower-level components of the middleware architecture (e.g.

[3]We call as "user" the programmer of the adaptive service.

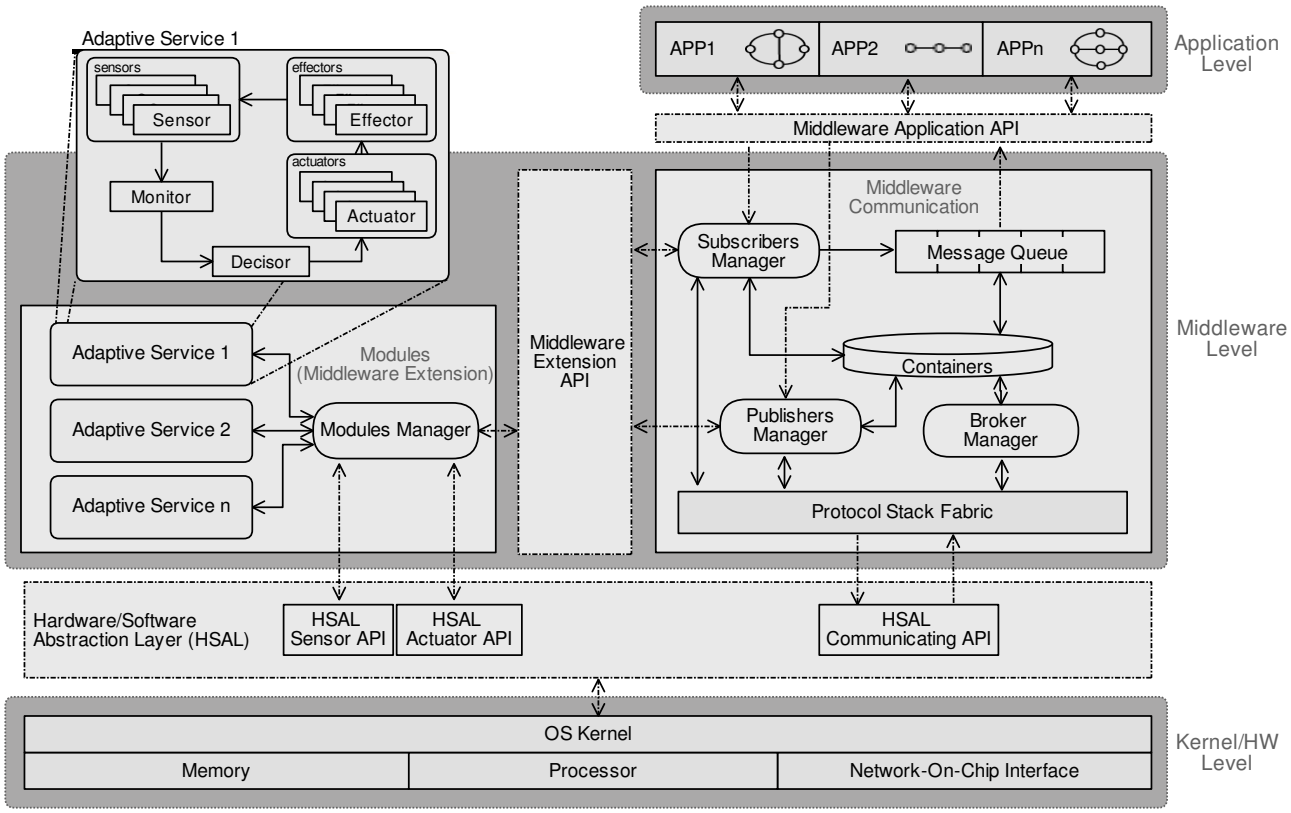


Figure 5.2: OO-MQSoC architecture enhanced with the *Modules* middleware extension.

PublishersManager, SubscribersManager, Broker, ProtocolStack). Following, we describe the base class of the Modules component.

5.2.1. Modules Component

The *Modules* component (*Modules* class, in gray in the class diagram showed in Figure APPENDIX B.1), composes the set of derived classes that represent the elements of a given self-adaptive system. The self-adaptive system could be composed of one or more adaptive services. The derived classes are based on the following base classes: *Sensor*, *Monitor*, *Actuator*, *Effector*, and *Decisor*. The base classes contain virtual methods (those highlighted in bold) that must be implemented in their derived classes. Figure APPENDIX B.1 shows the class diagram of the *Modules* Component. The *ModuleInterface<T>* class is a template class that performs communication with low level primitives of the middleware, defining invariant properties of the adaptive service components. The *ModuleInterface<T>* class has a specialized implementation for each one of the base classes: *ModuleInterface<Sensor>*, *ModuleInterface<Monitor>*, *ModuleInterface<Actuator>*, and *ModuleInterface<Effector>*. Following we explain each one of these specialized classes. The *Decisor* class has no implementation of *ModuleInterface<>* because a decisor does not need to use the communication primitives of the middleware. A derived class from *Decisor* is fully decoupled from the middleware logic, referring only to which monitors and actuators it uses, as explained in the decisor example in Section 5.4.3. The *Modules* class presents a set of methods, as follows:

- *enable()* - General code to be run when the Modules object is enabled.
- *enableSensors()* - Enables the set of sensors.
- *enableMonitors()* - Enables the set of monitors.

- *enableActuators()* - Enables the set of actuators.
- *enableEffectors()* - Enables the set of effectors.
- *enableDecisors()* - Enables the set of decisors.
- *updateSensors()* - Calls the *update method* of the set of sensors. The Algorithm 1 shows a pseudo-code of the *updateSensors() method*, where $sensors_{set}$ is the set of sensors.
- *updateDecisors()* - Calls the *decide* method of the set of decisors. The Algorithm 2 shows a pseudo-code of the *updateDecisors() method*, where $decisors_{set}$ is the set of decisors.

Algorithm 1 updateSensors() method pseudo-code

```
1: Input: sensors_set
2: for each sensor_i ∈ sensors_set do
3:     if sensor_i.isEnabled() then
4:         sensor_i.update()
5:     end if
6: end for
```

Algorithm 2 updateDecisors() method pseudo-code

```
1: Input: decisors_set
2: for each decisors_i ∈ decisors_set do
3:     if decisors_i.isEnabled() then
4:         decisor_i.decide()
5:     end if
6: end for
```

5.2.2. Modules Base Classes

In this section we detail the base classes used by Modules component detailed in Section 5.2.1, as follows:

ModuleInterface<Sensor>

The *ModuleInterface<Sensor>* class is a specialized template class of the *ModuleInterface<T>* class presenting the composition of the methods necessary to implement a sensor object in the middleware. Figure APPENDIX B.2 shows the class diagram of the *ModuleInterface<Sensor>* class. It is a derived class from both *Sensor* and *ModuleInterfacePublisher* classes. The *ModuleInterfacePublisher* class contains methods to interface with low-level communication primitives of the middleware, detailed in Section 5.2.2. In Figure APPENDIX B.2, *ExampleASensor* and *ExampleBSensor* are effective implementations of sensors, that must implement their own *update() method* - derived from the*Sensor class* - and *updateStatus() method* - derived from the *ModuleInterface<Sensor>* class. Section 5.4.1 describes an example of how implement an effective sensor class. The *ModuleInterface<Sensor>* class contains two public methods described as follows:

- *updateStatus()* - Virtual method declaration that must be implemented in the derived sensor class; the implementation must update the sensor data with current values using HSAL primitives in order to access kernel or hardware sensors or any other method to access the observed sensor data;
- *updateData(data : ModuleType)* - Virtual method implementation that: i) calls the *updateStatus* method to update the sensor data with current value; ii) serializes the updated sensor data calling the *serialize* method of the derived *ModuleType* class; iii) and calls the transmit method of the *ModuleInterfacePublisher* class to invoke underlying publish communicating primitive.

ModuleInterface<Monitor>

The *ModuleInterface<Monitor>* class is a specialized template class of the *ModuleInterface<T> class*. The *ModuleInterface<Monitor>* class presents the composition of the methods necessary to implement a monitor object in the middleware. Figure APPENDIX B.3 shows the class diagram of the *ModuleInterface<Monitor>* class. It is a derived class from both *Monitor* and *ModuleInterfaceSubscriber* classes. The *ModuleInterfaceSubscriber* class contains the methods to interface with low-level communication primitives of the middleware, detailed in Section 5.2.2. Figure APPENDIX B.2, *ExampleAMonitor* and *ExampleBMonitor* are effective implementations of monitors, that must implement their own *enable method* derived from the *Monitor* class and *doit method* derived from the *Callback* class. Section 5.4.1 describes an example of how implement an effective monitor class. The *ModuleInterface<Monitor>* class does not contain any particular method, only derives the methods of the base classes.

ModuleInterface<Actuator>

The *ModuleInterface<Actuator>* class is a specialized template class of the *ModuleInterface<T>* class. This class presents the composition of the methods necessary to implement an actuator object in the middleware. Figure APPENDIX B.4 shows the class diagram of the *ModuleInterface<Actuator>* class. It is a derived class from both *Actuator* and *ModuleInterfacePublisher* classes. The *ModuleInterface<Actuator>* class contains a public method described as follows:

- *driveData(data : ModuleType)* - Virtual method implementation that: i) serializes the updated actuator data calling the *serialize* method of the derived *ModuleType* class; ii) calls the *transmit* method of the *ModuleInterfacePublisher* class to invoke underlying publish communicating primitive

.

ModuleInterface<Effector>

The *ModuleInterface<Effector>* class is a specialized template class of the *ModuleInterface<T>* class. The *ModuleInterface<Effector>* class presents the composition of the methods necessary to implement an effector object in the middleware. Figure APPENDIX B.5 shows the class diagram of the *ModuleInterface<Effector>* class. It is a derived class from both *Effector* and *ModuleInterfaceSubscriber* classes. In Figure APPENDIX B.2, *ExampleAEffector* and *ExampleBEffector* are effective implementations of effector objects, that must implement their own *enable* method derived from the *Effector* class and *doit* method derived from the *Callback class*. Section 5.4.2 describes an example of how implement an effective effector class.

ModuleInterfacePublisher

The *ModuleInterfacePublisher* class contains the methods necessary to handle the communication primitives of the middleware as a publisher client. Figure APPENDIX B.6 shows the class diagram of the *ModuleInterfacePublisher* class. It also contains private member variables representing the message to be transmitted and the topic that identifies the data flow. Both *ModuleInterface<Actuator>* and *ModuleInterface<Sensor>* are derived classes from *ModuleInterfacePublisher* class. The *ModuleInterfacePublisher* class derives the methods of the *Publisher* class (Section 5.2.2), which presents the last level of abstraction before calling the *PublishersManager* class methods. The *ModuleInterfacePublisher* class contains two public methods described as follows:

- *transmit()* - Virtual method implementation that calls the *publish* method of the derived *Publisher* class passing the topic and message variable members as parameter;
- *enableAsPublisher()* - Virtual method implementation that calls the *advertise* method of the derived *Publisher* class passing the topic variable member as parameter.

ModuleInterfaceSubscriber

The *ModuleInterfaceSubscriber* class contains the methods necessary to handle the communication primitives of the middleware as a subscriber client. Figure APPENDIX B.7 shows the class diagram of the *ModuleInterfaceSubscriber* class. It contains private member variables representing the message to be received and the topic that identifies the data flow. Both *ModuleInterface<Monitor>* and *ModuleInterface<Effector>* are derived classes from *ModuleInterfaceSubscriber* class. The *ModuleInterfaceSubscriber* class derives the methods of the *Subscriber* class, which presents the last level of abstraction before calling the *SubscribersManager* class methods. The *ModuleInterfaceSubscriber class* contains one public method described as follows:

- *enableAsSubscriber(callBackP : CallbackPtr)* - Virtual method implementation that calls the *subscribe* method of the derived *Subscriber* class passing *callBackP* as parameter for *CallbackPtr*.

Publisher

The *Publisher* class implements the last level of abstraction before calling the *PublishersManager* class methods. Therefore, the *Publisher* class is responsible for performing, at last level, the communication with the middleware for components of an adaptive service modeled as a publisher client. The *Publisher* class contains two public methods described as follows:

- *advertise(t : Topic)* - Virtual method implementation that calls the *advertiseHandler* method of the *PublishersManager* class passing the *t* as parameter for *Topic*;
- *publish(t : Topic, mqMessage : MQSoCMessage)* - Virtual method implementation that calls the *publishHandler* method of the *PublishersManager* class passing the *t* as parameter for *Topic* and *mqMessage* as parameter for *MQSoCMessage*.

Subscriber

The *Subscriber* class implements the last level of abstraction before calling the *SubscribersManager* class methods. Therefore, the *Subscriber* class is responsible for performing, at last level, the communication with the middleware for components of an adaptive service modeled as a subscriber client. The *Subscriber* class contains one public method described as follows:

- *subscribe(t : Topic, callBackP : CallBackPtr)* - Virtual method implementation that calls the *subscribeHandler* method of the *SubscribersManager* class passing the *t* as parameter for *Topic* and *callBackP* as parameter for pointer to *CallBackPtr*. This pointer to a method of the subscriber node is called by the *SubscribersManager* class when it receives a message to respective topic.

UpDownActuator

The *UpDownActuator* class is design pattern implementation that handles a special set of actuation commands in an adaptive service. It standardizes the interface for a set of actuators that has the following behavior: sends actuation commands which can be represented by different levels of actuation. To use the *UpDownActuator* class, the levels of actuation of a given actuator must be able to be represented as an enumerator class. The enumerator class must implement both *operator++* and *operator--* methods that are handled by the *UpDownControl* class. The *UpDownControl* class implements the standard interface provided by the *UpDownActuator* class. Figure APPENDIX B.8 shows the class diagram of the *UpDownActuator* class. As represented, the *UpDownActuator* class is a derived class from *ModuleInterface<Actuator>* and *UpDownControl* classes. The *UpDownActuator* is implemented as a template class that receives from its derived class the enumerator class type declaration and the values that represent the first, last and initial values of this enumerator class. The *UpDownActuator* class contains the following public methods derived from their base classes:

- *up()* - Increases the actual value of the enumerator class by one level;
- *down()* - Decreases the actual value of the enumerator class by one level;
- *get()* - Returns the actual enumerator class value;
- *advance(n : int)* - Increases (positive *n* value) or decreases (negative *n* value) the actual value of the enumerator class by *n* levels;
- *to_first()* - Advances to the first value defined for the enumerator class;
- *to_last()* - Advances to the last value defined for the enumerator class;
- *at_first()* - Returns *true* if the actual value corresponds to the first value defined for the enumerator class;
- *at(val : T)* - Returns *true* if the actual value corresponds to *val*.

5.2.3. Atomic Operations among Sensors, Monitors, Actuators and Effectors Objects

External components (e.g. *Modules* component) can invoke the following atomic operations of objects of an adaptive service, detailed as follows: a) *Sensor* - *enable()* and *update()*; b) *Monitor* - *enable()*; c) *Actuator* - *enable()* and *drive()*; d) *Effector* - *enable()*. We show as follows the operations triggered when these methods are called.

Enabling a Sensor, Monitor, Actuator or Effector object

Figure 5.3-a shows the sequence diagram representing what happens when an external component calls the *enable()* method (1 in Figure) of a hypothetical *Sensor* object named *ExampleASensor*. The atomic operation invokes the respective methods (2 and 3) of the base classes detailed earlier until performing the sensor enabling through the call of the *advertiseHandler (topic)*

method of the *PublishersManager* component (4). The same happens for enabling the objects of a *Monitor* (Figure 5.3-b), an *Actuator* (Figure 5.3-c), and an *Effector* (Figure 5.3-d).

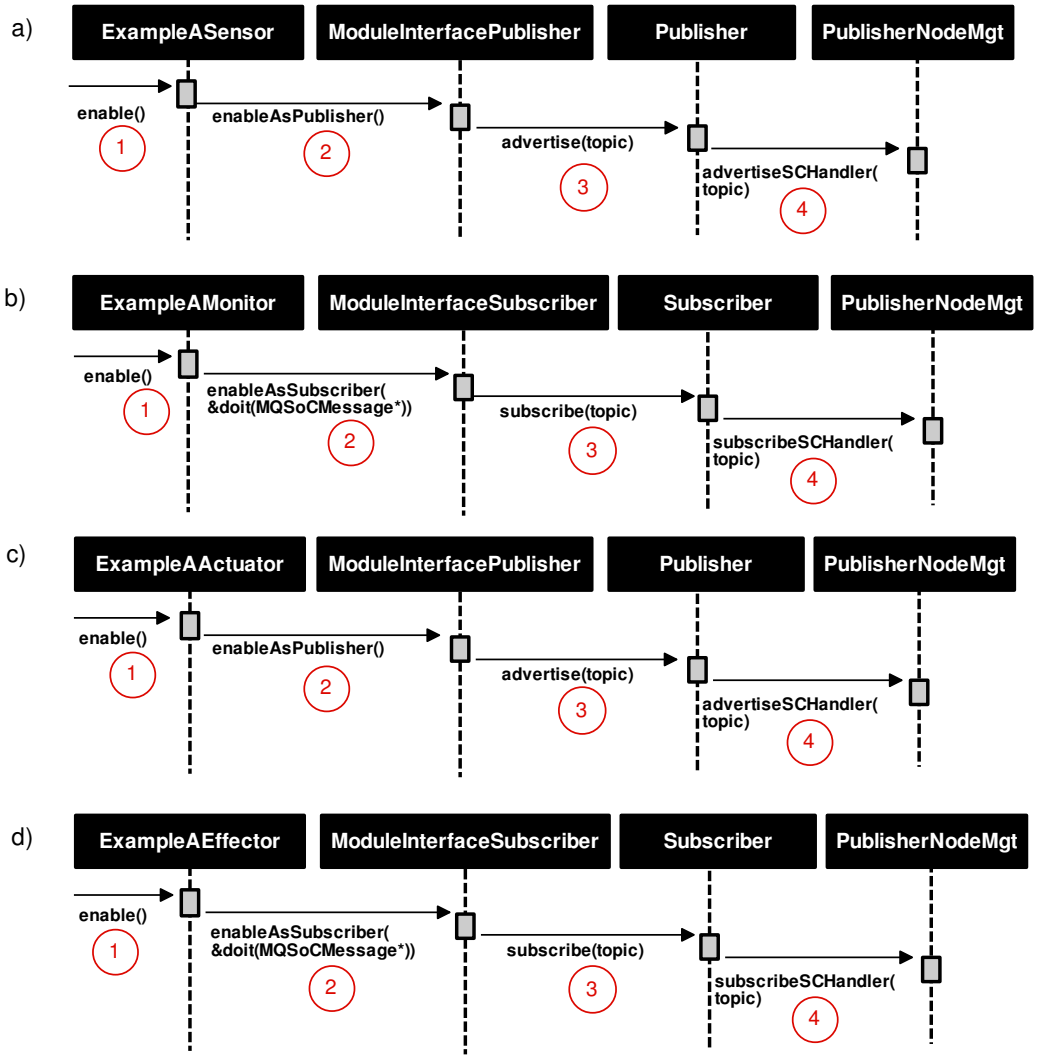


Figure 5.3: Sequence diagram of the *enable()* atomic operation of a hypothetical: a) *Sensor*; b) *Monitor*; c) *Actuator*; d) *Effector*.

Updating the data of Sensor object

Figure 5.4 shows the sequence diagram representing what happens when an external component calls the *update ()* method (1 in Figure) of a hypothetical sensor object named *ExampleASensor*. The *update* method of the sensor object calls the *updateData* method (2) of the *ModuleInterface<Sensor>* base class, passing the sensor data as parameter. The *updateData* method calls the *updateStatus* method (3) of the *ExampleASensor* object to update the sensor data with an actual value (4). Then, the *updateData* method calls the *serialize* method (5) of the sensor type defined in the example by the *ExampleAModuleType* class in order to serialize the sensor data into the message payload (6). After the serialization process, the *updateData* method calls the *transmit* method (7) of the *ModuleInterfacePublisher* base class, which invokes the *publish* method (8) of the *Publisher* base class. Finally, the *publish* method in the *Publisher* base class calls the *publishHandler* method of the *PublishersManager* component passing the topic and the message as parameters. The *publishHandler* method is responsible for performing the underlying message processing before transmit the message (e.g., adds the message header).

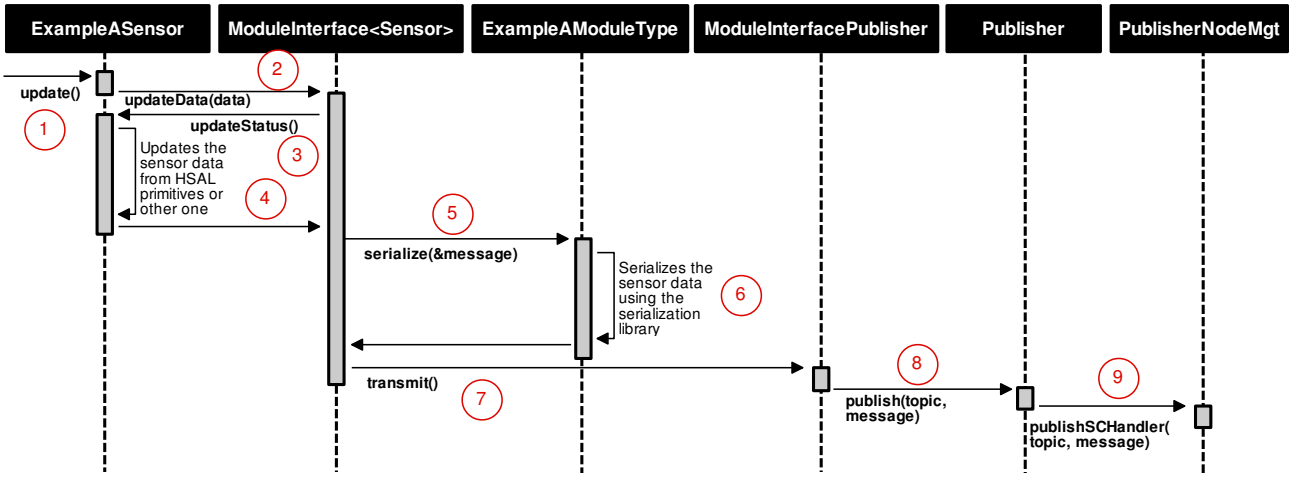


Figure 5.4: Sequence diagram of the *update()* atomic operation of a hypothetical *Sensor* object named *ExampleASensor*.

Receiving a monitored sensor data in a Monitor object

Figure 5.5 shows what happens when the message with the sensor data is delivered to the *SubscribersManager* component through a call from the *receiveHandler* method (1 in Figure). Note that this method could be called by the *PSLayer* component whether the message is received from the NoC, or by the *PublishersManager* component whether the message is received locally. The *receiveHandler* method checks the received message topic, searches for the callback pointer of that topic in its records, and calls the respective callback method named *doit* passing the message as parameter (2). The *doit* method, implemented in the monitor object named the *ExampleAMonitor*, calls the *deserialize* method (3) implemented in the respective monitor type defined in the example as the *ExampleAModuleType* class (the same *module type* class of the respective sensor pair). After deserialized (4), the local variable of the ExampleAMonitor object contains the current value of the monitored sensor data (5).

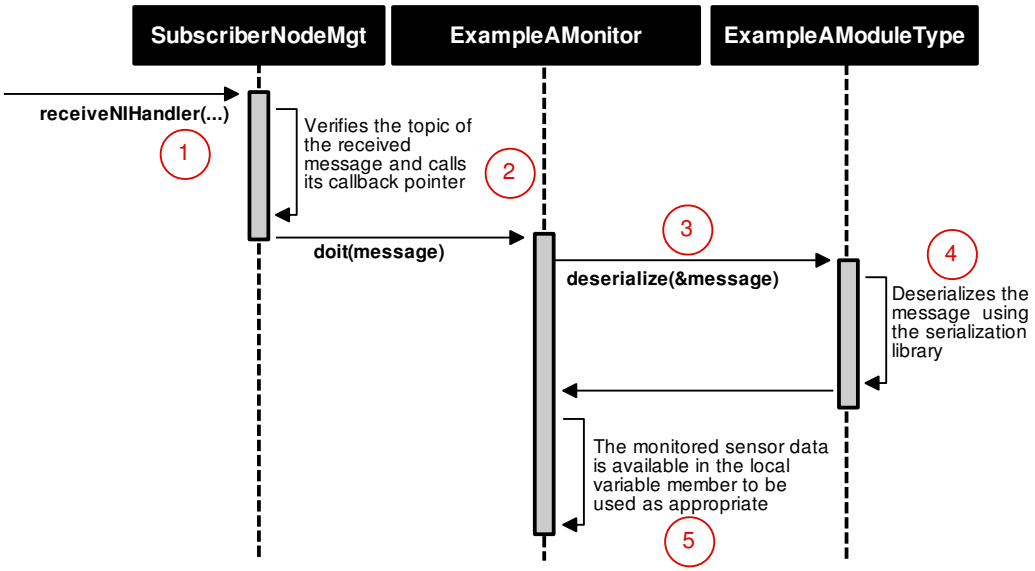


Figure 5.5: Sequence diagram for receiving a sensor data message in a hypothetical monitor object named *ExampleAMonitor*.

Actuating in an actuator data of an Actuator object

Figure 5.6 shows the sequence diagram representing what happens when an external component calls the *drive()* method (1 in Figure) of a hypothetical actuator object named *ExampleAActuator*. The *drive* method of the sensor object calls the *driveData* method (2) of the *ModuleInterface<Actuator>* base class, passing the sensor data as parameter. The *driveData* method calls the *serialize* method (3) of the actuator type defined in the example by the *ExampleAModuleType* class in order to serialize the actuator data into the message payload (4). After the serialization process, the *driveData* method calls the *transmit* method (5) of the *ModuleInterfacePublisher* base class. From here, the message transmission follows the same steps of the sensor data transmission. The *transmit* method (5) of the *ModuleInterfacePublisher* base class calls the *publish* method (6) of the *Publisher* base class. Then, the *publish* method in the *Publisher* base class calls the *publishHandler* method of the *PublishersManager* component passing the topic and the message as parameters. The *publishHandler* method delivers the message for the destination subscriber node.

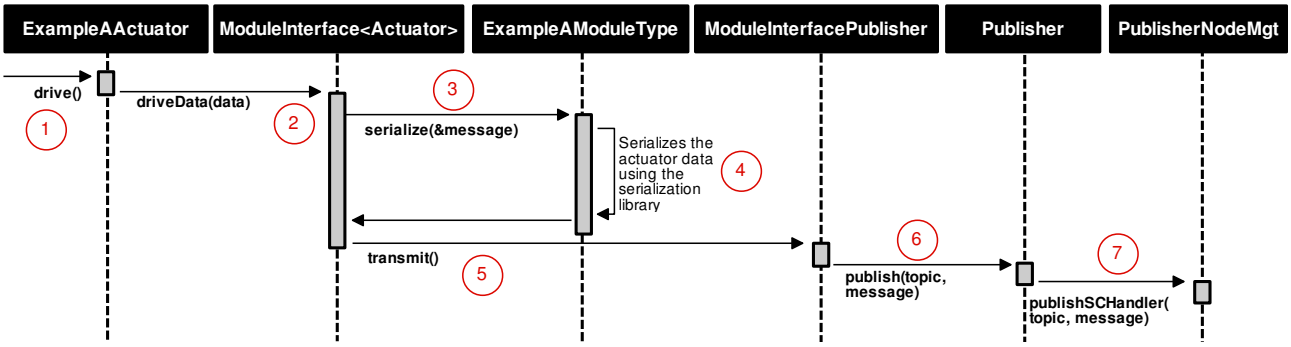


Figure 5.6: Sequence diagram of the *drive()* atomic operation of a hypothetical actuator object named *ExampleAActuator*.

Receiving an actuator data in an Effector object

Figure 5.7 shows what happens when the message with the actuator data is delivered to the *SubscribersManager* component through a call from the *receiveHandler* method (1 in Figure). Likewise receiving a monitored sensor data in a monitor object, the *receiveHandler* method could be called by the *PSLayer* component whether the message is received from the NoC, or by the *PublishersManager* component whether the message is received locally. The *receiveHandler* method checks the received message topic, searches for the callback pointer of that topic in its records, and calls the respective callback method named *doit* passing the message as parameter (2). The *doit* method, implemented in the effector object named *ExampleAEffector*, calls the *deserialize* method (3) implemented in the respective module type defined in the example as the *ExampleAModuleType* class (the same module type class of the respective actuator pair). After deserialized (4), the local variable of the ExampleAEffector object contains the current value of the actuator data and can be used to effect a given configuration of the system (5).

5.3. Topic-Name Scheme

A monitor-rich system presents a large amount of different information to be made available. Likewise, system elements need to know about the existence of resources on the network. In the publish-subscribe model, a *topic* represents the atomic information made available in the

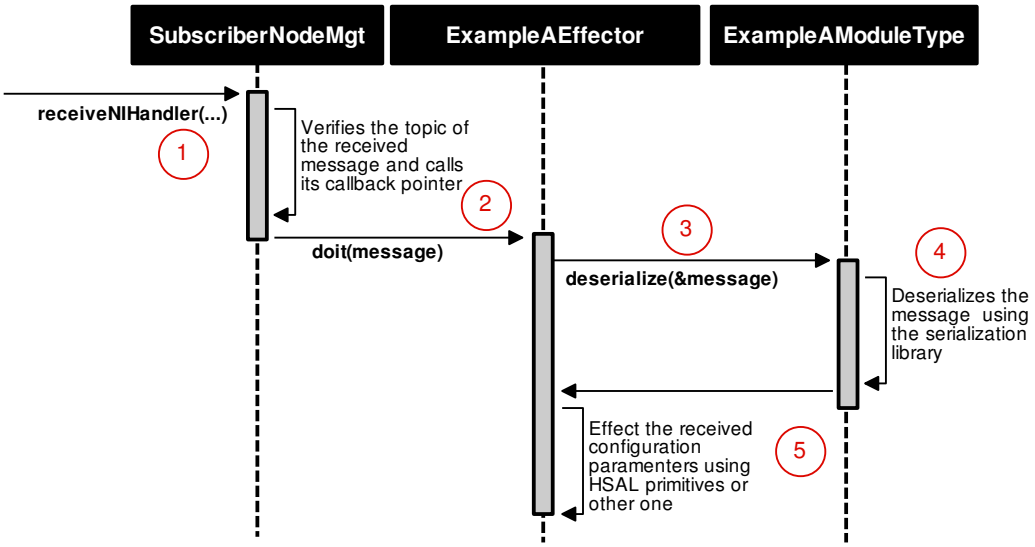


Figure 5.7: Sequence diagram for receiving a actuator data message in a hypothetical effector object named *ExampleAEffector*.

system (e.g. energy spent by a processing element). An organized way of maintaining a topic index available for query and subscription is used by the *ROS* environment [ROS16b], which manages a hierarchical graph of resource names [ROS16a]. In this way, we implement a method of registration of resources (topics) names based on *ROS*, organized in the form of a hierarchical graph. In addition, we implement a wildcard-based subscription method based on Mosquitto environment [Ecl16]. With these two methods, nodes can subscribe to one or more topics with a single call to subscribe. For subscription on more than one topic of the same hierarchy level or underlying hierarchy level, the user may use the wildcards "+" or "#", respectively. For example, for a topic graph with four levels of hierarchy (Figure 5.8), the following subscriptions could be performed:

/temperature/pe1 –> subscription in temperature topic of the processing element 1

/temperature/+ –> subscription in temperature topics of all processing elements

/msg_lost_cnt/router1/w –> subscription in the counter of lost messages of router1 west port

/msg_lost_cnt/router1/+ –> subscription in all counters of lost messages of all router1 ports

/msg_lost_cnt/# –> subscription in all counters of lost messages of all routers

/# –> subscription in all topics of the system

The topic-name scheme must be refined according to the purpose of the publish-subscribe system. However, the developer must follow the hierarchical graph to name the topics of the system, where the different levels are separated by the symbol "/". Figure 5.8 shows the topic-name scheme used in this work, following a four-level hierarchical graph. The first level identifies if the topic is a sensor or an actuator topic. The second and third level are related to the location of the sensor or actuator topic, could be one of this locations: pe (processor element), memory, router or cluster. The third level is the identification of that location. For example, a topic named "/sensor/pe/3/." is related to the processor element sensor identified by 3. In the same way, a topic named "/sensor/pe/6/..." is related to the sensor of the processor element identified by 6. The fourth level defines the type of sensor or actuator topic: i) slacktime and energy for a pe sensor; ii) injection, congestion, and energy for a router sensor; iii) energy for a cluster sensor; iv) DVFS (Dynamic Voltage and Frequency Scaling) for a PE actuator; v) powerMode for a cluster actuator.

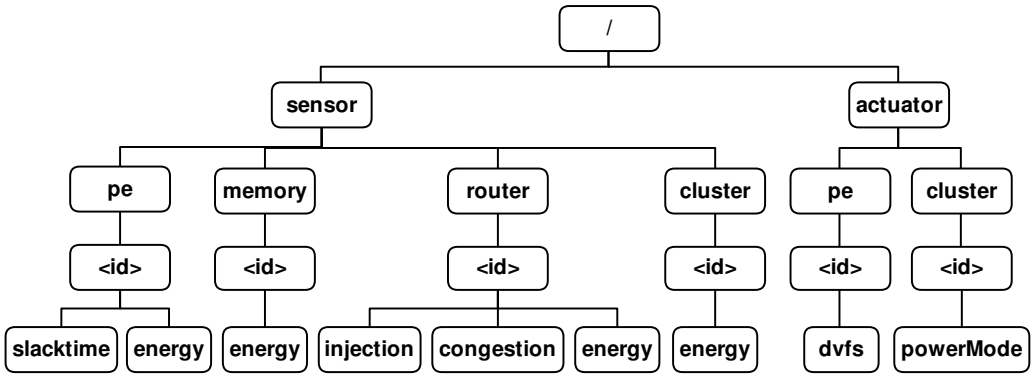


Figure 5.8: Topic-name scheme employed in this Thesis following the hierarchical graph.

Additionally, we have developed a topic domain scheme to define the reach of a topic. In this way, a topic name is always related to a topic domain. Aiming the use of the middleware in clusterized MPSoCs, we define two domains: i) *intra-cluster*, for communication between elements within the same cluster; ii) *inter-clusters*, for communication between elements that are not in the same cluster. When a publisher or subscriber node advertises or subscribes to a topic, it must also inform which is the domain of that topic. For this purpose, we have defined two domains: i) SLAVES (for intra-cluster communication); ii) MANAGERS (for inter-cluster communication). The middleware implements a broker structure by domain to handle the advertise and subscribe requests in the topics. The local master node of the each cluster handles the broker structure in the "SLAVES" domain. For the "MANAGERS" domain, global master PE handles the broker structure. Note that while the "SLAVES" domain balances the broker processing by taking advantage of the clusterization feature, the "MANAGERS" domain has a scalability issue because the broker of this domain centralizes all the communication of the advertise and subscribe operations. In this way, the "MANAGERS" domain must be used carefully concerning the issue scalability.

5.4. Creating the Objects of the Adaptive Service

This section describes the classes and methods that the user must create to implement an adaptive service following the ODA model in the middleware. First, we describe in Sections 5.4.1, 5.4.2, and 5.4.3 how to create the necessary classes and their methods manually. Additionally, we present in APPENDIX F a tool to automate the process of creating the necessary classes and their methods. With the tool, the user needs to manipulate only the content of the methods, incrementing them as desired.

Firstly, it is essential to understand the directory structure of the Modules extension of the middleware. The *Modules* extension directory, showed in APPENDIX D, contains all class files that build the *Modules* extension. At the first level of the hierarchy, the *Modules* extension directory named *modules* contains the *common* directory, which holds the base classes detailed in Section 5.2.2, and the *modules_ports* directory. The *modules_ports* contains the specific class implementations to perform an adaptive service into a given platform. That is, *modules_ports* contains the classes that will be customized from one implementation to another. The user must create a new directory inside *modules_ports* for each adaptive service.

In the example showed in APPENDIX D, the directory named *dvfs* represents an adaptive service of a DVFS mechanism. Figure 5.9 shows the components of the adaptive service implemented in this example. Sections 5.4.1, 5.4.2 and 5.4.3 demonstrate how to create the *Sensor*, *Monitor*, *Decisor*, *Actuator*, *Effector*, and *Types* classes that represent this example.

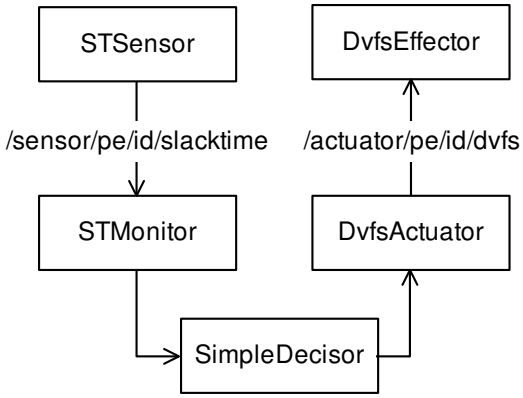


Figure 5.9: Components of the adaptive service implemented as example.

5.4.1. Sensor/Monitor pair and Type classes

We show as follows how to manually create the variable and methods of the *Sensor*, *Monitor* and *Type* classes.

Sensor class

Beginning by the sensor class implementation, in the example that we are going to demonstrate, the sensor distributes the slack time data of the PE in a topic named */sensor/pe/<id>/slacktime*, where *<id>* is the identification of the PE. The periodicity of updating this data depends on the periodicity in which the *update()* method is called externally. The user must give a name for the sensor class (we call *STSensor*), and implement the *update()* and *updateStatus()* methods in its source file, in addition to defining the *topic name* and *domain* in the class constructor. This class must have declared in the header file a public member named *data* with its type defined by *Type* class to be specified following. Figures APPENDIX C.1 and APPENDIX C.2 show the code snippet demonstrating the minimal implementation of the header and source files of the *STSensor* class.

Monitor class

For the *Monitor* class implementation, the user must give a name for the *Monitor* class (*STMonitor* in this example), and implement the *enable()* and *doit(MQSoCMessage* pMQMessage)* methods in its source file. The *enable()* method must define the *topic name* and *domain* of the observable sensor data. We implement the *STMonitor* class to receive the sensor data of slacktime of all PEs that have advertised to the respective topic. The *Modules* feature allows representing this interest through the subscription to the topic named */sensor/pe/+/slave*, using the *wildcard* feature (see Section 5.3). The *STMonitor* header file contains an additional class declaration to store the sensor data of all sensors, named *STMonitor_T*. When a sensor data is received, the *doit* method deserializes the message and stores the received data to an array declared in the *STMonitor_T* class. The *Decisor* class, detailed in Section 5.4.3, uses the *STMonitor_T* class to access the observable sensor data. Figures APPENDIX C.3 and APPENDIX C.4 show the code snippet demonstrating the minimal implementation of the header and source files of the *STMonitor* class.

Type class

The *Type* class must implement the *serialize* and *deserialize* methods used by the sensor and monitor classes. In this example, we named the type class as *STType*. Figures APPENDIX C.5 and APPENDIX C.6 show the code snippet demonstrates the minimal implementation of the header and source files of the *STType* class. The code uses the MsgPuck serialization library (see Section 4.4.3) to serialize and deserialize the members that represents the observable sensor data. Note that the user could use the desired serialization method by following the signature of the *serialize* and *deserialize* methods of the *MessageType* base class.

5.4.2. Actuator/Effector pair and Type classes

We show as follows how to manually create the constructors and methods of the *Actuator*, *Effector* and *Type* classes.

Actuator class

In the demonstrated example, the *Decisor* class detailed in Section 5.4.3 actuates in the system through a DVFS mechanism. For this, the *Decisor* class instances an object of the *Actuator* class named *DvfsActuator*. The *Actuator* class header file must contain the declaration of the enable method, while the source file must contain the implementation of the enable method. The method implementation must define the topic name and domain that identify the actuation data, named in this example as */actuator/pe/<id>/dvfs*, where *<id>* is the identification of the PE in which the decisor will actuate. The *Decisor* class implementation that instances the *DvfsActuator* class must enable the set of actuator objects passing the *id* of the PE in the *enable* method parameter. Figures APPENDIX C.7 and APPENDIX C.8 show the code snippet demonstrates the minimal implementation of the actuator header and source classes. The *Actuator* class implemented in this example uses the *UpDownActuator* design pattern class implementation.

Effector class

The pair *effector* of the *DvfsActuator* class, named *DvfsEffector*, must contain the declaration of the *enable* and *doit* methods in the header file and their implementation in the source file. The *enable()* method must define the *topic name* and *domain* used by the *DvfsActuator* class. The *doit* method must call the *deserialize* method and uses the received actuation data to configure the system. In this example, we apply the received actuation data using the primitive *HAL_OS_EFFECTORS_setDVFS(value)* that applies the value parameter in the processor voltage/frequency configuration (the DVFS mechanism is detailed in Section 6.1.3). Figures APPENDIX C.9 and APPENDIX C.10 show the code snippet demonstrates the minimal implementation of the effector header and source files.

Type class

The *Type* class used by the *DvfsActuator* and *DvfsEffector* class must implement the *serialize* and *deserialize* methods used by the sensor and monitor classes. In this example, we named the *Type* class as *DvfsType*. Figures APPENDIX C.11 and APPENDIX C.12 show the code snippet demonstrates the minimal implementation of the header and source files of *Type* class.

5.4.3. Decisor class

The *Decisor* class as implemented in this hypothetical example, named *SimpleDecisor*, must contain the *enable* method declaration and declare a pointer to the desired *Monitor* class to access its monitored sensor data. Also, it must instance the *DvfsActuator* class. The *enable* method implementation must enable all the instanced monitors calling the *addCallbackMonitor* method and passing a pointer to respective member method, named *notifySTMonitor* in the example. The *notifySTMonitor* method is called when the variable that stores the observable sensor data is updated. In the example, the *notifySTMonitor* calls the *decide()* method which contains the decision-making logic according to the observable data of the slacktime sensor. Note that the decision is made whenever the slacktime data of a PE is updated. Figures APPENDIX C.13 and APPENDIX C.14 show the code snippet demonstrates the minimal implementation of the header and source files of the *Decisor*.

5.4.4. Experiment

We perform an experiment to verify the effectiveness of middleware coordination to perform communication between the pairs *Sensor/Monitor*, and *Actuator/Effector*. The main goal of the experiment is to confirm that the adaptive service works correctly for the purpose described in the source codes of its components. We use a platform composed of a non-clusterized 3x3 NoC-based MPSoC with homogeneous PEs, as detailed in Section 6.1. Each PE has a MIPS-like processor, DMNI module, private random access memory (RAM) and router. The frequency/voltage scaling actuation mechanism (DVFS) operates only on the processor, memory and DMNI, as detailed in Section 6.1.3. There are three possible DVFS level, following a decreasing order of performance: *vf(7)*, *vf(6)*, and *vf(4)*. The platform contains sensors of CPU[4] (Central Processing Unit) utilization at hardware level measured according to the equation 6.11. Section 6.1.3 details the DVFS mechanism. The experimental setup consists of a 3x3 MPSoC configuration running an AES application with 5 tasks. Figure 5.10 shows the setup of the experiment with the performed application (a), and the MPSoC platform setup with the location of the objects (b) of the evaluated adaptive service (c). We use an AES application - MPI-based - to impose a workload for the MPSoC in order to observe a variation in the CPU utilization. Each slave PE has one instance of the *STSensor* and *DvfsEffector* components. The master PE has one instance of the *SimpleDecisor* and *STMonitor* components, besides eight instances of the *DvfsActuator* component (one for each slave PE).

Figures 5.11 and 5.12 show the CPU utilization over time in the PE. The graph shows samples of CPU utilization every 250ms, which is the time window to sample new sensor data. The graphs represent the SP's view of the sensor data and also show the actuation commands received by the respective SP (blue line). As noted, the PE allocates the task before that instant 1000 Kticks in both PEs. At this time, both PEs have a CPU utilization of around 80% regarding the initial processing-load spent mapping the task. All PEs start execution on the highest performance voltage/frequency pair (represented by *vf(7)*). The *SimpleDecisor* object (Section 5.4.3) implements a decision logic where the DVFS is scaled to down when CPU utilization is less than 33% and to up when CPU utilization is greater than 66%. We can observe that the actuation occurs in six instants for PE 1 in Figure 5.11. The DVFS is scaled to down at the time instants marked by the first, second, fourth, and sixth blue, horizontal lines, and to up at the time instants marked by the third

[4] The target platform uses the term CPU to represent the processor.

and fifth blue, horizontal lines. For PE 4 in Figure 5.12, the DVFS is scaled firstly to down when the CPU utilization is lower than 33% and to up after reaching 66%.

The goal of this experiment is to demonstrate the actuation of the evaluated adaptive service against the sensor data. Chapter 6 demonstrates a complex case study of a self-adaptive system composed of two adaptive services that actuate at two levels: slaves PE and cluster.

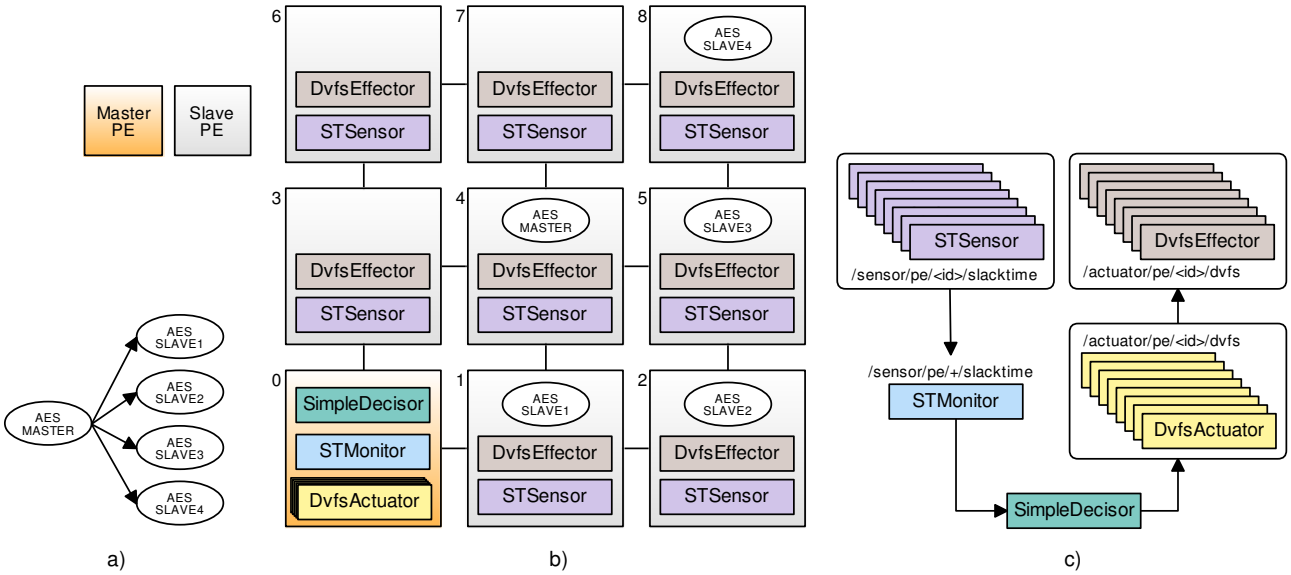


Figure 5.10: Experiment setup: a) AES task graph; b) 3x3 MPSoC platform setup; c) Evaluated adaptive service.

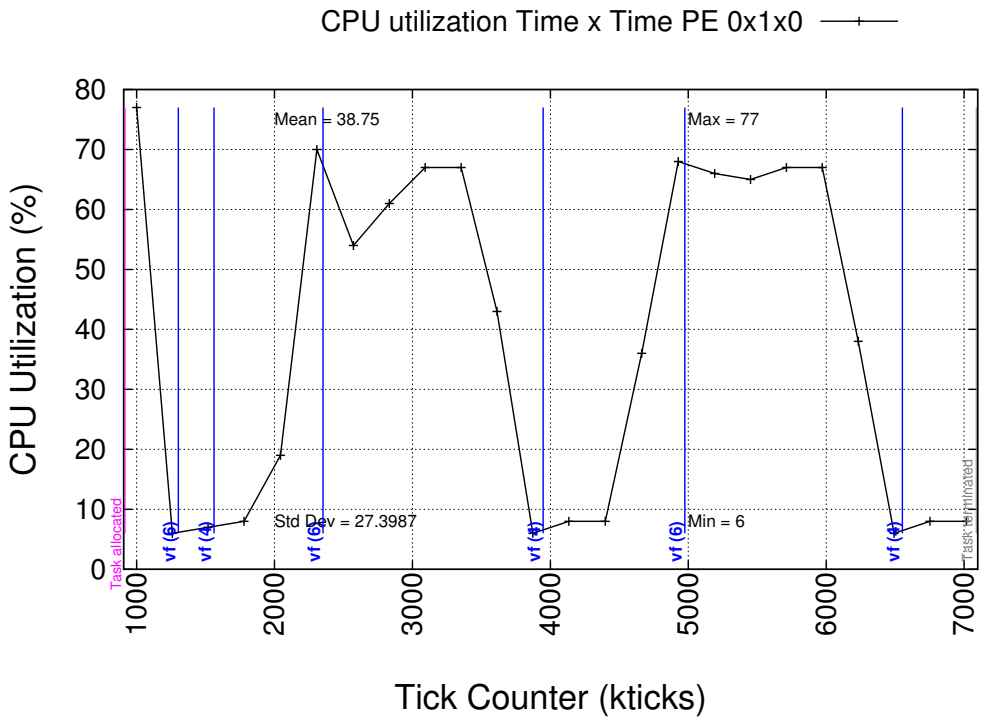


Figure 5.11: CPU utilization over time for the PE 1.

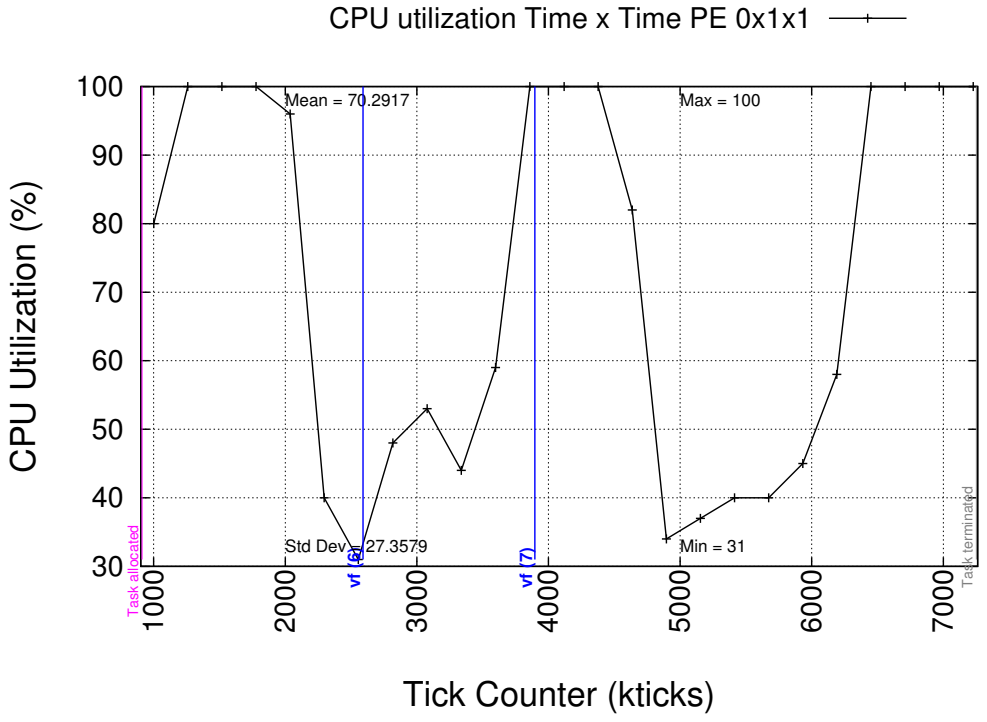


Figure 5.12: CPU utilization over time for the PE 4.

6. CASE STUDY OF A SELF-ADAPTIVE SYSTEM

This chapter presents a case study in which we compare two versions of a multilevel self-adaptive system on an MPSoC platform. The objective is to compare the results obtained according to performance, energy and software quality metrics, aiming to contrast the approaches used for the development of these self-adaptive systems.

In this way, the chapter is organized as follows. Section 6.1 presents the baseline MPSoC platform and baseline self-adaptive system proposed at [MdSR+19]. Section 6.2 presents the same platform, however refactoring the whole self-adaptive system with the proposed middleware. Section 6.3 presents the results.

6.1. Baseline Platform

The baseline platform used for the self-adaptive system case study presented in this chapter is the HeMPS[1] platform. HeMPS is an open-source many-core architecture framework used to implement and evaluate software and hardware modules for many-cores. Figure 6.1 shows the many-core architecture of the baseline platform.

The platform is composed of a NoC-based MPSoC that embeds homogeneous processing elements (PE). Each PE (Figure 6.1-b) is equipped with a Plasma CPU[2] (MIPS-like) [Ope14], Direct Memory Network Interface (DMNI) module [RLMM16], a private Local Memory, and packet-switch based router. The DMNI contains a direct memory access interface incorporated into a network interface, which allows the router to access the Local Memory without using the processor. The applications enter in the MPSoC through an extra memory module - called Application Repository - attached to the PE located in the bottom-left position. At the system initialization, the applications' static code is loaded into PEs' local memories to be run. When all resources are busy, the manager system maintains the application in the application repository and loads the object code when there is an available execution resource.

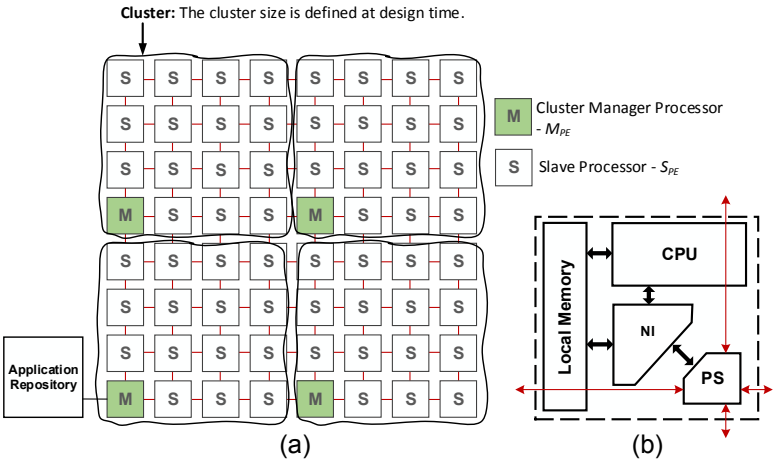


Figure 6.1: Baseline HeMPS MPSoC: (a) system architecture; (b) PE architecture [RM18, p. 35].

The baseline platform organizes the PEs in clusters. Each cluster has a Manager PE (M_{PE})[3] that assumes features of system management such as task mapping and application admission

[1] http://www.inf.pucrs.br/hemps/

[2] The target platform uses the term CPU to represent the processor.

[3] The M_{PE} is also called as *LM* in this Thesis.

control. The Slave PEs (S_{PE})[4] receive the applications' static code from the Application Repository and run them in multitasking mode, managed by a kernel software embedded in each PE. The M_{PE} and S_{PE} have different kernel software. Additionally, there is also one specific version of the M_{PE} kernel, called SM_{PE} (System Master PE). The SM_{PE} implements the same features of the M_{PE}, added with functions of overall cluster management and control of the interface with the application repository with correlated features to application mapping on the cluster. More details about the application admission management can be found at [RM18, p. 35].

6.1.1. Baseline Self-Adaptive System

This section presents the baseline self-adaptive system used for comparison purposes with the self-adaptive system designed using the middleware approach, presented in Section 6.2. The baseline self-adaptive system, called Multi-Objective Resource Management (MORM), has been designed by Martins and Moraes [MdSR+19]. For comparison purposes, we reference its implementation in the present case study as MORM-C. MORM aims to operate in an environment of dynamic execution of applications, respecting a limit of power consumption, called *power cap*. MORM works by managing the execution of applications should avoid exceeding the power cap in addition to maintaining the system operation in the valley of power consumption. It has been designed to act in a clusterized MPSoC, where each cluster can be configured to work in two operation modes as follows: i) performance mode, where the maximum performance is scaled up to reach the power cap; ii) energy mode, where performance is reduced to reach the valley of power.

MORM is classified as multi-objective management because it concomitantly addresses power, energy, and performance goals [MdSR+19]. MORM considers both communication and computing sensoring for actuation.

Figure 6.2 shows the overview structure of the MORM self-adaptive system following the hierarchical organization of the clusterized MPSoC, where *SP* is a Slave PE, CM is the Cluster Manager PE (one by cluster) and GM is the Global Manager PE (one for the system as whole). MORM follows the observe-decide-act paradigm [DJS15], performing both system observation, decision making and actuation at the system/cluster level (inter-cluster domain) and processing element level (intra-cluster domain).

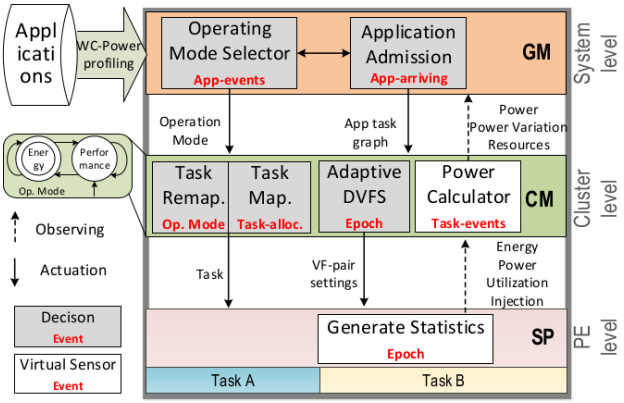


Figure 6.2: General MORM overview [MdSR+19, p. 4].

The **system observation** follows a bottom-up direction. At the PE level, the SPs send observed data (energy, CPU utilization, NoC congestion) to their CMs. In turn, each CM sends the

[4] The S_{PE} is also called as *SP* in this Thesis.

data observed in its cluster (power consumption) to the GM. The **system decision making** follows the hierarchical cluster organization, with the GM deciding on the observed inter-cluster data and the CMs deciding on the observed intra-cluster data. The **system actuation** follows a top-down direction. An actuation method acts on the PE level setting the voltage-frequency pair of one or more SPs. In turn, another actuation method acts on the system-level changing the operation mode of a given cluster. The following subsections detail the observation, actuation, and decision making methods used by MORM.

6.1.2. Observation Methods

MORM uses a power and energy characterization methods to infer the power dissipation and energy consumption for each PE, cluster, and system as a whole. The methods consider the processor, NoC (Network-on-Chip) and memory to estimate the static and dynamic power dissipation. The method to estimate power and energy is general because it is based on a calibration process to define the energy/power values. The characterization process employs a synthesizable VHDL description of the base platform in RTL level. The next subsections describe the power and energy characterization methods for the processor, router and memory. The method does not consider the power and energy for the DMNI because it is a small module compared to the processor, router, and memory.

Processor Power/Energy Characterization

The processor power characterization incorporated into MORM was proposed by Martins at [MSC+14], comprising five steps:

1. Grouping all the instructions of the processor into instruction classes according to their type (e.g., arithmetic, logical, shift, move, nop, branches, jumps and load-store). In this step, an assembly program is implemented with the instructions of the class to be used in step 4.
2. Simulation at RTL level of each instruction class, counting the number of clock cycles to execute it for validation purposes in step 5.
3. Logic synthesis of the processor defining the processor frequency for a given technology. This step generates a netlist with annotated delay data to be used in step 4.
4. Simulation at gate-level of each assembly program created in step 1 using the netlist generated in step 3. This step generates the switching activity at the gate level as well as traces for functional validation with step 2.
5. Power measurement from the switching activity traces generated in step 4. Table 6.1 shows the power and energy measured at the end of the processor power characterization.

The energy for each instruction class (E_{class}) is measured following the Equation 6.1, where P_{class} is the power measured for a given instruction class, CPI is the number of clock cycles for that instruction class and T is the clock period[5].

$$E_{class} = P_{class} * CPI * T \tag{6.1}$$

The total energy consumption for the processor ($E_{processor}$) and total power dissipation for the processor ($P_{processor}$) are measured following the equations 6.2 and 6.3, where $n_{instructions_i}$ is the

[5]The clock period used in this case study is 4ns

Table 6.1: Power characterization results and energy estimation for each instruction class of the processor. Library CORE65GPSVT (65nm), 1.1V, 25ºC (T=4ns).

Class	**Avg. Power (mW)**		**Energy per inst. (pJ)**	
	Leakage	Dynamic	Leakage	Dynamic
Arithmetic	0.452	5.894	1.808	23.58
Logical		5.176		20.70
Shift		4.940		19.76
Move		4.768		19.07
Nop		3.331		13.32
Branches		5.723		31.70
Jumps		4.175		18.56
Load-store		5.507	3.616	43.15

number of executed instructions for the instruction class *i* in a given execution sample, E_{Class_i} is the energy measured for the instruction class *i* (constant, from Equation 6.1) and P_{Class_i} is the power measured for the instruction class *i* (constant, from Table 6.1).

$$E_{processor} = \sum_{i=0}^{n_{classes}} n_{instructions_i} \times E_{class_i} \tag{6.2}$$

$$P_{processor} = \sum_{i=0}^{n_{class}} n_{instructions_i} \times P_{class_i} \tag{6.3}$$

The processor power characterization considers that the processor consumes only static power when no task is running in the processor or when it is awaiting state (e.g., waiting for data from another task).

Martins [MSC+14] has validated the calibration process for the energy and power measurements using different benchmarks. This validation generates an error on the power and energy processor measured at the RTL level, compared to the gate level. In the related experiments, the error was from -9% to 8% for energy consumption and -9% to 10% for power dissipation, depending on the benchmark.

Router Power/Energy Characterization

The router that composes the PE architecture (Figure 6.1-b) includes internal components such as input buffers, crossbar, and control logic for arbitration and routing purposes. The router characterization process has been proposed by Martins at [MSC+14]. It is similar to the processor characterization, comprising four steps:

1. Generation of traffic in all 5 ports of the router for maximizing the switching activity. This step creates 6 testcase scenarios varying the injection rate from 0% to 50% of the link bandwidth.

2. Logic synthesis of the one instance of a 5-port router, generating a netlist to be used in step 3. The energy consumption of the wires (links) considers the wire capacitance between two routers (considering the distance of 1 mm between the routers).

3. Simulation of the 6 scenarios created in step 1 in a 3x3 NoC instance, replacing the router RTL description by the netlist generated in step 2. Each testcase scenario generates a switching activity file at the gate level for the respective injection rate.

4. Power measurement from the switching activity file generated in step 3. Despite the 6 injections rates performed in the characterization method, the final characterization adopts two rates: 100% - representing the active mode of the router (worst case); and 0% - representing the idle mode of the router. Table 6.2 shows the power characterization for the two considered router modes. The second column of the table represents the dynamic average power consumption for one buffer. The combinational logic represents the power consumption of the remaining router components. The last column represents the leakage power of the router.

Table 6.2: Router Average Power. Library CORE65GPSVT (65nm), 1.1V@4ns, 25ºC.

Traffic Rate	**One buffer**	**Combinational Logic**	$\mathbf{P^{leak}_{router}(n_{ports} = 5)}$
0% - idle	364.64 μW	575.64 μW	223.08 μW
100% - active	755.56 μW	2655.25 μW	

The energy consumption characterization for the router is divided in dynamic active energy to receive one flit (named E^{active}_{router}) and dynamic idle energy (named E^{idle}_{router}). They are measured respectively following the equations 6.4 and 6.5, where n_{ports} is the number of ports of the router (constant, 5), $P^{active}_{component}$ is the power consumption of a given component in the active mode, $P^{idle}_{component}$ is the power consumption of a given component in the idle mode, and T[6] is the period of the sample timing window.

$$E^{active}_{router} = [((n_{ports} - 1) \times P^{idle}_{buffer}) + P^{active}_{buffer} + P^{active}_{comb}] \times T \tag{6.4}$$

$$E^{idle}_{router} = [(n_{ports} \times P^{idle}_{buffer}) + P^{idle}_{comb}] \times T \tag{6.5}$$

Memory Power/Energy Characterization

The memory power/energy characterization uses the CACTI-P tool [LCA+11] enabling the characterization of the energy consumption of the PE local memory. Table 6.3 shows the *access time* that corresponds to the period used to characterize the processor and the router (rounded to 4ns), the P^{leak}_{memory} that is the leakage power of the memory, the E_{load} that is the dynamic energy spent per read access in the memory, and the E_{store} that is the dynamic energy spent per store access in the memory.

Table 6.3: CACTI-P Report for a Scratchpad Memory (65nm, 1.1V, 25ºC).

Access time	$\mathbf{P^{leak}_{memory}}$	$\mathbf{E_{load}}$	$\mathbf{E_{store}}$
3.98 ns	0.66 mW	67 pJ	38 pJ

Processing Element (PE) Power/Energy Characterization

MORM uses data of energy consumption for the processing element as a whole. For this, an observing module sums locally the total energy provided by the processor, router and memory energy characterization, supported by a set of additional hardware and software components.

The additional hardware components include a set of registers into the PE for counting events: a) n_{class} - number of executed instructions for the respective instruction class; b) $cycles_{active}$

[6] The sample timing window (T) used in this case study is 250ms

- number of clock cycles in which the router is transmitting flits; c) $cycles_{total}$ - number of clock cycles executed by the PE at the sample timing window. The sample timing window is called an *epoch* - an execution timing interval defined by a periodic hardware interruption. At the end of an *epoch*, all counters are read and reset to zero.

The additional software components include functions to read the counters at the end of an *epoch* and, from this, estimate the PE power and energy. The following equations sum to already related equations to perform the PE energy estimation, where $E_{processor}$, E_{memory} and E_{router} are the dynamic energy spent respectively by the processor, memory and router at an *epoch*, $E_{leakage}$ is the energy from leakage power of the whole PE and E_{PE} is the total energy spent by the PE at an epoch.

$$E_{processor} = \sum_{i=0}^{n_{classes}} n_{instructions_i} \times E_{dynclass_i} \tag{6.6}$$

$$E_{memory} = n_{instructions_{load}} \times E_{load} + n_{instructions_{store}} \times E_{store} \tag{6.7}$$

$$E_{router} = E^{idle}_{router} \times (cycles_{total} - cycles_{active}) + E^{active}_{router} \times cycles_{active} \tag{6.8}$$

$$E_{leakage} = [P^{leakage}_{processor} + P^{leakage}_{memory} + P^{leakage}_{router} \times n_{ports}] \times cycles_{total} \times T \tag{6.9}$$

$$E_{PE} = E_{processor} + E_{memory} + E_{router} + E_{leakage} \tag{6.10}$$

Others PE observed data

MORM also uses others observed data to perform the decision making described in Section 6.1.4. Additional hardware registers and software functions provide observed data from the router and processor, which are as follows: a) $Utilization_{processor}$ - percentage of clock cycles where the processor is running a task or kernel routine, calculated from the equation 6.11; b) *router injection* - buffer utilization of the local port of the router; c) *router congestion* - buffer utilization of the non-local ports (north, south, east and west). Hardware counters in the router provide the measurements of *router injection* and *router congestion*, which are measured by the average message flow in the respective router ports at an *epoch*.

$$Utilization_{processor} = \frac{cycles_{active}}{cycles_{idle}} \tag{6.11}$$

6.1.3. Actuation Methods

The base platform provides some actuation methods that are used by the MORM to achieve the desired adaptability. Hardware modules or software features implements the provided actuation methods. Figure 6.3 shows the actuation methods used by MORM and the classification according to its implementation, software or hardware, described as follows.

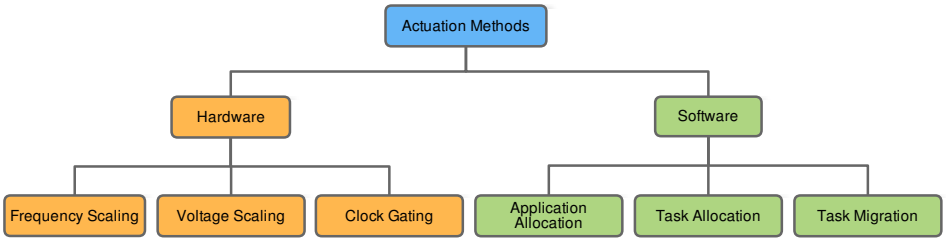


Figure 6.3: Classification of actuation methods adopted by MORM

Dynamic Voltage/Frequency Scaling - DVFS

The frequency scaling actuates on the processor, memory and DMNI hardware components of the PE. The router works in a nominal frequency to avoid that a processor at higher frequency stall due to PEs that are running at a lower frequency. The DMNI synchronizes the hardware modules which work at different frequencies using two bi-synchronous FIFOs. The additional hardware necessary to implement the frequency scaling at the baseline platform is a clock-generator and the bi-synchronous FIFOs at the DMNI. The additional FIFO produces, at average, an overhead of 6.55% in the applications execution time. More details about the hardware modifications to perform the frequency scaling can be found at [MdSR+19].

The voltage scaling employed on the baseline platform follows a method for modeling supply voltages supported by the system. The method evaluates the processor netlist for 1.0V and 0.9V since nominally the processor works at 1.1V-250MHz. The same process is performed for the memory and router. The goal is to measure the minimum period to obtain an acceptable slack concerning processor, memory, and router, without interfering with the operation of these hardware components.

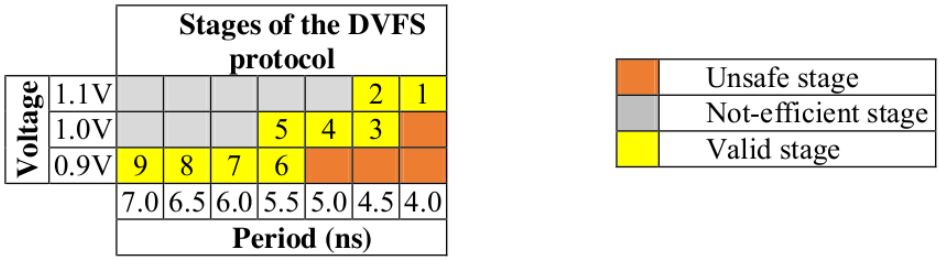


Figure 6.4: Secure voltage/frequency pairs [MM18, p. 74].

Figure 6.4 shows the results of the performed evaluation, listing the minimum period for safely scaling the voltage and frequency. The yellow boxes define safe Voltage/Frequency pairs, numbered from 1 to 9 that correspond to scale, respectively, from highest to lower performance modes. The hardware necessary to implement the voltage scaling is a voltage regulator that presents a latency of 100ns (25 clock cycles at 250MHz) and an energy overhead of 10% [CCK07].

Clock-Gating

The clock gating employed on the baseline platform considers a model that affects only the processor and memory. According to the adopted model, the processor clock signal is disabled when it is in idle mode, saving dynamic power. The processor does not perform memory operations at the idle mode. Consequently, the memory does not accumulate dynamic power in the idle mode (since the dynamic power comes from load and store operations only). The router continuously

spends dynamic power in the idle and active mode. When in idle, the router considers the dynamic power from buffers (equation 6.5). The timing overhead of the clock gating actuation is considered negligible [MdSR+19].

Application Allocation

The application allocation method assigns the tasks of an application to a chosen cluster. The application allocation follows a protocol triggered when the application repository set the respective hardware interruption in the GM. The GM runs an algorithm to select[7] a cluster to map the application tasks, informing the respective CM of its choice. The CM also runs an algorithm to select the SPs to map the application tasks, reporting it to the GM that is the only PE with physical access to the application repository. Then, the GM stars the task allocation protocol detailed next. Decision making methods at the GM and CM kernel implement algorithms to select, respectively, the cluster and SPs that will receive the tasks. They are detailed in Section 6.1.4.

Task Allocation

The task allocation method assigns the tasks of an application to the chosen SPs. The task allocation method only coordinates the tasks to the respective SPs. There is no making decision method in this process since the application allocation method does it. The task allocation begins with the GM getting a task code from the application repository and sending it directly to the SP informed by the CM in the application allocation. After receiving the task, the SP notifies this to its CM, which releases the execution in the respective SP.

Task Migration

The task migration method remaps the tasks of an application inside the same cluster. That is, the task migration process moves a task from a source SP to a destination SP. The component that uses the task migration method is responsible for the decision making about who are the involved SPs. The task migration protocol migrates all sections of the task object code directly from the source to the destination SP, without using checkpoints. The migrated task enters in running mode at the destination SP after the task migration process for all tasks finishing.

6.1.4. Decision Making Methods

MORM employs decision-making methods that work synchronically to adapt the system to multi-objective. Multi-objective means that MORM addresses power, energy, and performance concomitantly [MdSR+19]. MORM uses the clusterization feature of the base platform to shift the goals according to the workload behavior.

MORM is the composition of a set of observation, actuation and decision-making methods. However, it is important to note that what differentiates MORM from other self-adaptive system is the decision-making methods used by it. The employed decision-making methods are responsible by decisions that are taken at two different domains: i) inter-cluster domain, to handle the power cap, application mapping and to choose the cluster operation modes; ii) intra-cluster domain, to control the DVFS, task mapping, and task migration. They are described as follows.

[7] The GM does not admit an application execution if there are no sufficient available resources (number of free pages minor than the number of application tasks) in any cluster, waiting until they are released.

MORM Inter-Cluster Decision-Making Method

The decision-making method employed in the inter-cluster domain is responsible by the admission of an application in the system. Figure 6.5 shows a general schema of the MORM Inter-Cluster Decision-Making Method.

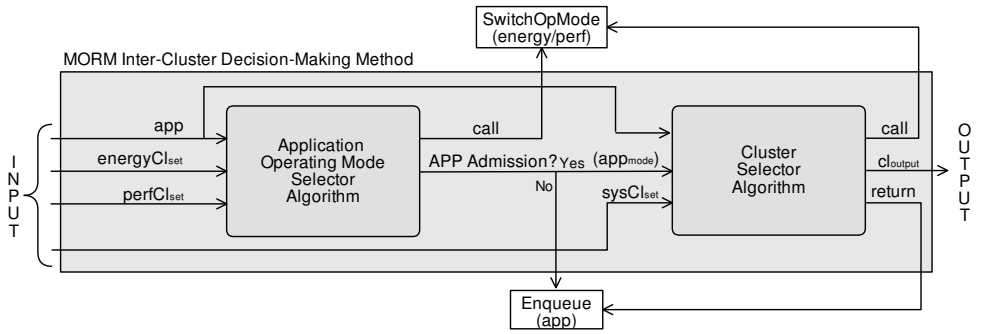


Figure 6.5: General Schema of the MORM Inter-Cluster Decision-Making Method.

The method comprises two algorithms to: i) select the *application operating mode* (Algorithm 3); ii) select the *cluster to admit the application* (Algorithm 4). The GM performs both algorithms in the kernel level.

Algorithm 3 shows a pseudo-code of the process to define the *application operating mode*. The algorithm can shift the operation modes of the clusters while respecting the system power cap based on the amount of power disturbance induced by application events [MdSR+19]. The algorithm receives as input the application description, the set of clusters operating in energy mode ($\text{energyCl}_{\text{set}}$) and the set of clusters operating in performance mode ($\text{perfCl}_{\text{set}}$), generating as output the selected application mode (app_{mode}). The application description (*app*) comes from an application power profiling performed at design-time. At design-time, simulations of the application set measure the power consumption in two scenarios: i) when the application tasks are executing in different PEs without CPU sharing and at the nominal voltage and frequency parameters; ii) when the application tasks are executing in the same PE at the most energy efficient voltage and frequency parameters. More details about the application profiling can be found at [MM18, p. 109]. The algorithm uses the power measurements of the application profiling to infer the power dissipation in the performance (*app.pwrPerformance*) and energy (*app.pwrEnergy*) modes.

The execution of the algorithm is triggered when an application requests admission into the system or when an application finishes its execution. All clusters operate in performance mode at the beginning of the system execution.

When an application requests its admission (line 2 in Algorithm 3), the algorithm allows admission if the estimated power for running the application in performance mode, added to the current system power, does not exceed the system power cap (lines 3-5). Otherwise, the algorithm estimates the power increment to allow application in energy mode (line 7). If the estimate exceeds the power cap (line 8), the algorithm switches the operating mode of clusters running in performance mode to energy mode until the estimation falls below the power cap (lines 9-14). If the estimation does not exceed the power cap (line 15), the application is admitted in energy mode (line 16). Otherwise, the application is queued for later acceptance (line 18). If the estimation calculated to admit the application in energy mode (line 7) does not exceed the power cap (line 20), the algorithm allows the application to be admitted in energy mode (lines 24 and 26). If there is no cluster in energy mode by then (line 21), the algorithm switches the operating mode from performance to energy (line 23) of that cluster that has the largest number of processors available (line 22). When an application finishes its execution (line 30), the algorithm switches to performance (line 34) the

Algorithm 3 MORM Application Operating Mode Selector [MdSR+19, p. 114]

```
1:  Inputs: app, energyCl_set, perfCl_set
2:  if app is arriving then
3:      newPwr ← sys.pwr + app.pwrPerformance
4:      if newPwr < sys.pwrCap then
5:          Allows the admission of the application in performance mode
6:      else
7:          newPwr ← sys.pwr + app.pwrEnergy
8:          if newPwr > sys.pwrCap then
9:              for each cl_i ∈ perfCl_set do
10:                 newPwr ← newPwr + cl_i.pwrVariation
11:                 if newPwr > sys.pwrCap then
12:                     shiftOpMode(cl_i, energy)
13:                 end if
14:             end for
15:             if newPwr < sys.pwrCap then
16:                 Allows the admission of the application in energy mode
17:             else
18:                 Application enqueued to be admitted later
19:             end if
20:         else
21:             if energyCl_set = ∅ then
22:                 cl_output ← maxAvailSPs(perfCl_set, performance)
23:                 shiftOpMode(cl_output, energy)
24:                 Allows the admission of the application in energy mode
25:             else
26:                 Allows the admission of the application in energy mode
27:             end if
28:         end if
29:     end if
30: else                                          ▷ app finished its execution
31:     for each cl_i ∈ energyCl_set do
32:         newPwr ← sys.pwr + cl_i.pwrVariation
33:         if newPwr < sys.pwrCap then
34:             shiftOpMode(cl_i, performance)
35:         end if
36:     end for
37: end if
```

operating mode of that cluster that is in energy mode (line 32) and whose power increment does not exceed the power cap (lines 32-33).

After selecting the operating mode of the application, GM selects the cluster to admit the application based on the available resources. Algorithm 4 shows a pseudo-code of the process for selecting the cluster to admit the application. The algorithm takes as input the application description, the output of the Application Operating Mode Algorithm (app_{mode}), and the set of system

clusters (sys.cl_{set}). The algorithm outputs the cluster (cl_{output}) selected to receive the application. The algorithm initially calculates (line 4 in Algorithm 4) the number of SPs required to run the application based on the application description and the application mode selected by Algorithm 3. The number of SPs is equal to the number of application tasks for performance mode or less than that for energy mode, since the tasks can share the CPU in the energy mode. The algorithm then checks on clusters running in the selected application mode for those that have enough SPs to run the application (lines 5-9). Among these clusters, the algorithm returns the one with the maximum number of SPs running without tasks (lines 10-13). If there are not enough clusters available with enough SPs to run the application, the algorithm selects the performance mode cluster with the maximum number of available SPs (line 14), changes the application mode to energy (line 15), and updates the number of SPs required to run the application in energy mode (line 16). If the number of free SPs for the selected cluster is sufficient to receive the application (line 17), the algorithm switches the operating mode (line 18) of the cluster and returns the selected cluster identification (line 19). Otherwise, no cluster is available to run the application, and the application is queued to run later.

Algorithm 4 MORM Cluster Selector [MdSR+19, p. 115]

1: **Inputs**: app, app_{mode}, $sys.cl_{set}$
2: **Outputs**: cl_{output}
3: $cl_{set} \leftarrow \emptyset$
4: $SP_{min} \leftarrow$ **getMinSPsAdmitApp**(app, app_{mode})
5: **for each** $cl_i \in sys.cl_{set}$ **do**
6: **if** $app_{mode} = cl_i.mode$ **and** $cl_i.freeSP \geq SP_{min}$ **then**
7: $cl_{set} \leftarrow cl_{set} \cup cl_i$
8: **end if**
9: **end for**
10: **if** $cl_{set} \neq \emptyset$ **then**
11: $cl_{output} \leftarrow$ **maxAvailSPs**(cl_{set}, app_{mode})
12: **return** cl_{output}
13: **end if**
14: $cl_{output} \leftarrow$ **maxAvailSPs**$(sys.cl_{set}, performance)$
15: $app_{mode} \leftarrow energy$
16: $SP_{min} \leftarrow$ **getMinSPsAdmitApp**(app, app_{mode}) ▷ update SP_{min}
17: **if** $cl_{output}.freeSP \geq SP_{min}$ **then**
18: **shiftOpMode**$(cl_{output}, energy)$
19: **return** cl_{output}
20: **else**
21: **return** $\emptyset$
22: **end if**

MORM Intra-Cluster Decision Making Method

The decision-making method employed in the intra-cluster domain is responsible for selecting the voltage-frequency pair of the SPs within a cluster in addition to choose the SPs that will receive the tasks of an application. The method comprises two mechanisms described as follows: i) adaptive DVFS; ii) task mapping/remapping. The CM of each cluster performs both mechanisms in the kernel level.

Algorithm 5 shows a pseudo-code of the process responsible for selecting the voltage-frequency pair (vf-pair) for a given SP, enabling the DVFS adaptive feature. It is a threshold-based algorithm that receives as input the cluster operating mode (cl_{opMode}), the message injection rate of an SP ($SP_{injection}$), and the processor utilization of that SP ($SP_{utilization}$) to decide the vf-pair to be applied to that SP. The algorithm can generate as output one of these three values for the vf-pair: i) nominal (VF_{perf}) - highest voltage/frequency pair; ii) low power (VF_{min}) - lowest voltage/frequency pair; iii) EDP (VF_{EDP}) - more energy efficient voltage/frequency pair (intermediate value between nominal and low power). The algorithm executes whenever the CM receives an observation sample message from an SP containing the message injection rate and the processor utilization of that SP. The algorithm starts by setting the vf-pair to nominal (line 3). If the cluster is operating in energy mode (line 4), the algorithm changes the vf-pair to low power in two situations (lines 6-7): i) high injection rate ($SP_{injection} > 75\%$) - the SP is injecting messages into the network at a higher speed than the destination consumes them; ii) low processor utilization ($SP_{utilization} < 25\%$) - the SP is in an idle state for most of the time waiting for messages. Otherwise, the algorithm switches the vf-pair to EDP (line 5). In order to define the levels of high and low injection rate, the algorithm considers that the injection rate is the average utilization of the input bu[U+FB00]er in the local port.

Algorithm 5 MORM Adaptive DVFS

```
1:  Input: cl_opMode, SP_injection, SP_utilization
2:  Outputs: VF_pair
3:  VF_pair ← VF_perf
4:  if cl.opMode = ENERGY then
5:      VF_pair ← VF_EDP
6:      if SP_injection > 75 or SP_utilization < 25 then
7:          VF_pair ← VF_min
8:      end if
9:  end if
10: return VF_pair
```

The process of tasks mapping/remapping is an algorithm for selecting the SPs that will receive the application tasks. The algorithm receives as input the application profile and the application mode of operation (energy or performance). From the application operating mode, the algorithm decides by performance or energy mapping. Performance mapping maximizes application parallelism and optimizes execution time. It uses a single-task mapping where an SP does not share resources with other tasks. Energy mapping aims to use the minimum number of available processors to map an application. It uses a multi-task mapping where a processor shares resources between tasks. Tasks that communicate with each other can be mapped to the same SP. Parallel tasks do not share the same SP. The remapping mechanism uses two approaches - join and split - to perform adaptability when the operating mode of the cluster switches. Join remapping acts when a cluster is running in performance mode and receives an order to switch to energy mode. Its goal is to migrate the communicating tasks to execute them on the same processor, generating more idle processors that can be shut-down. Split remapping acts when a cluster is running in energy mode and receives an order to switch to performance mode. It aims to spread tasks across more SPs, optimizing task performance.

6.2. Self-adaptive system embedded in the Middleware

This section discusses the development of a self-adaptive system using the support provided by middleware. The self-adaptive system developed is based on the MORM self-adaptive system already detailed in Section 6.1.1. The goal is to refactor MORM using a middleware-based approach. The requirement is that the functionality for which MORM was developed is maintained.

As refactoring methodology, we identify the original MORM code inside the HEMPS's kernel software, and redesign your project using the classes of sensors, monitors, decision makers, actuators and effectors provided by the middleware's modules extension (Section 5.4). MORM is originally implemented in the C programming language and its source code is coupled to the kernel. Functions in the kernel implement the logic of the sensors (Section 6.1.2), actuators (6.1.3) and decision-makers (Section 6.1.4).

To identify the original MORM code inside the HEMPS's kernel software, we follow these steps: i) in the kernel software source code, we identify the functions that correspond to the logic of sensors, decision-makers and actuators; ii) for sensors and actuators, we disable the low-level kernel primitives used to send and receive messages; iii) for decision-makers, we completely disable the source code of the respective functions. We disable the functions and code snippets through the use of preprocessing directives (macros). The objective of using preprocessing directives is to allow the selection at design-time of which of the two self-adaptive systems could be enabled in the HEMPS system: i) original MORM in C, named MORM-C; ii) MORM refactored in the middleware, called MORM-MQSoC[8].

After the process of identifying the original MORM-C code, we proceeded with the modularization of that logic in the classes provided by the middleware's modules extension, described as follows: i) the Sensor and Monitor classes implement the logic of the MORM-C sensors; ii) the Actuator and Effector classes implement the MORM-C actuators; iii) the Decisor class implement the logic of the MORM-C decision-makers. Figures 6.6 and 6.7 demonstrate the adaptive services modeled on the middleware class structure for the Global Master PE and Local Master PE, respectively. Next, we discuss the design choices made for the implementation of both adaptive services. Figure 6.8 shows a MORM's overview for MORM-C and MORM-MQSoC implementations.

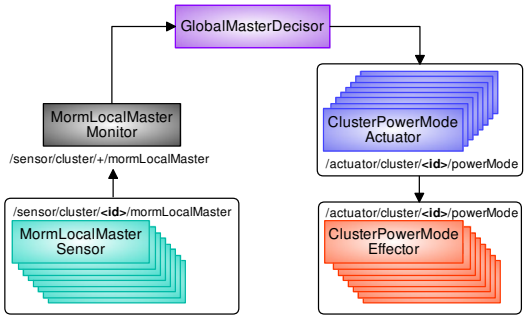


Figure 6.6: MORM-MQSoC Adaptative Service for the Global Master PE.

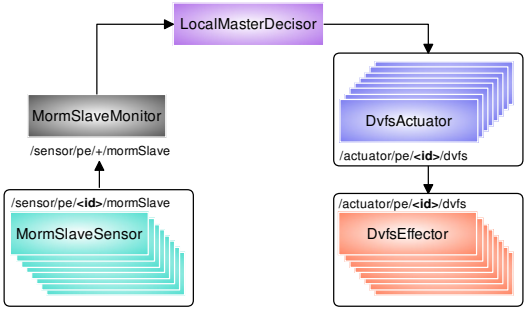


Figure 6.7: MORM-MQSoC Adaptative Service for the Local Master PE.

[8]We use the OO-MQSoC middleware version in all development and evaluation processes detailed in this Chapter. For simplicity, we call OO-MQSoC only as MQSoC in this Chapter.

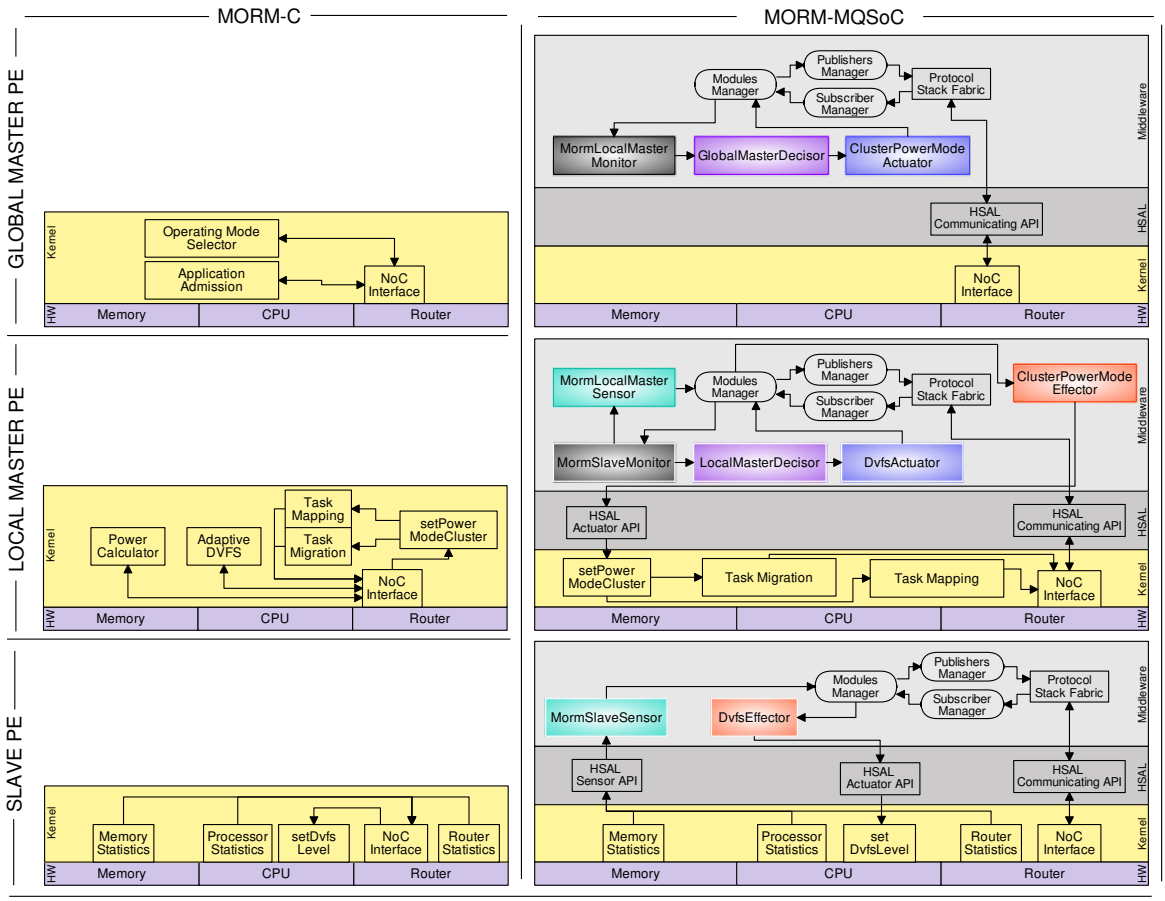


Figure 6.8: General MORM overview for the MORM-C and MORM-MQSoC implementations.

6.2.1. MORM-MQSoC Adaptative Service for the Local Master PE.

Many of the design choices we made regarding the modularization of MORM logic using the middleware classes aim to maintain similarity of functionality between the two approaches. For example, MORM-C encapsulates the sensor data of an SP in a single message, although it is more appropriate to separate the messages according to sensor type for modularization purposes. Differentiating memory's sensor data from router's sensor data would be indicated for cases where a monitor is only interested in one of these data. However, as the MORM-C encapsulates all sensor data from an SP in a single message, we follow this behavior. The adaptive service for the Local Master PE (LM) following our middleware-based approach is comprised of these classes, described below: ***MormSlaveSensor***, ***MormSlaveMonitor***, ***LocalMasterDecisor***, ***DvfsActuator***, and ***DvfsEffector***. Figure 6.7 shows a general overview of the relation between these classes representing an *ODA* loop.

MormSlaveSensor class defines the SP sensor object. In our middleware-based approach, a type class defines the format of the sensor data that will be serialized in the payload of the message to be sent. The *MormSlaveSensor* class uses the *MormSlaveType* class as the sensor data format. Following the detailed instructions in Section 5.4, the *MormSlaveType* class contains the *serialize* and *deserialize* methods, which encapsulate/decapsulate the sensor data in/from the message payload. The *MormSlaveSensor* class contains the *updateStatus* method, which defines where the current information of the sensor data is extracted to be serialized. Figure 6.12 shows the code snippet of the methods *serialize*, *deserialize* and *updateStatus* defined in the respective classes. Sensor data information is calculated at the kernel level by the same functions used by the

MORM-C implementation. The difference is that while the MORM-C sends the sensor data message using low-level kernel primitives, the MORM-MQSoC sends the message via the publish-subscribe primitives. An advantage highlighted here is that pub-sub decouples the source of the destination of a message. For example, in the communication model that uses low-level kernel primitives, if a sensor's data must be sent to more than one recipient, the developer of adaptive services must explicitly define this in the source code. In the publish-subscribe model, the developer does not need to define who is the recipient of the message, because the message is sent to a topic. Recipients who wish to receive the message from this sensor should show interest by subscribing to the topic. The *updateStatus* method of the *MormSlaveSensor* class accesses the sensor data through *HSAL* primitives. The topic named */sensor/pe/<id>/mormSlave* identifies the message generated by the *MormSlaveSensor* class, where *<id>* is the identification of each SP. The *Modules* class defines in its composition an instance of the *MormSlaveSensor* class for each SP.

```
void MormSlaveType::serialize(MQSoCMessage* pMQMessage)
{
  ...
  w = mp_encode_uint(w, this->id);
  w = mp_encode_uint(w, this->procSlacktime);
  w = mp_encode_uint(w, this->energyleak);
  w = mp_encode_uint(w, this->energyTotal);
  w = mp_encode_uint(w, this->realSamplingWindow);
  w = mp_encode_uint(w, this->routerInjection);
  w = mp_encode_uint(w, this->routerCongestion);
  ...
}

void MormSlaveType::deserialize(MQSoCMessage* pMQMessage) {
...
  this->id = mp_decode_uint(&r);
  this->procSlacktime = mp_decode_uint(&r);
  this->energyleak = mp_decode_uint(&r);
  this->energyTotal = mp_decode_uint(&r);
  this->realSamplingWindow = mp_decode_uint(&r);
  this->routerInjection = mp_decode_uint(&r);
  this->routerCongestion = mp_decode_uint(&r);
...
}

void MormSlaveSensor::updateStatus() {
...
  this->data.procSlacktime = HAL_OS_SENSORS_getSample_processorSlacktime();
  this->data.energyleak = HAL_OS_SENSORS_getSample_processorEnergyLeak() +
  HAL_OS_SENSORS_getSample_routerEnergyLeak() + HAL_OS_SENSORS_getSample_memoryEnergyLeak();
  this->data.energyTotal = this->data.energyleak + HAL_OS_SENSORS_getSample_processorEnergyDyn() +
  HAL_OS_SENSORS_getSample_routerEnergyDyn() + HAL_OS_SENSORS_getSample_memoryEnergyDyn();
  this->data.realSamplingWindow = HAL_OS_SENSORS_getSample_realSamplingWindow();
  this->data.routerInjection = HAL_OS_SENSORS_getSample_routerInjection();
  this->data.routerCongestion = HAL_OS_SENSORS_getSample_routerCongestion();
...
}
```

Figure 6.9: Code snippet of the *serialize* and *deserialize* methods of the *MormSlaveType* class, and the *updateStatus* method of the *MormSlaveSensor* class.

MormSlaveMonitor class is the pair of the *MormSlaveSensor* class. When subscribing to the sensor data topic of all SPs using the wildcard symbol "+", messages sent in this topic are received by the *MormSlaveMonitor* class. *MormSlaveMonitor* deserializes the received messages using the deserialize method of its instantiated *MormSlaveType* class. An array struct in the *MormSlaveMonitor* class stores the deserialized sensor data. Through this approach, decision-makers or other components that want to use the sensor data from the SPs must use a pointer to the *MormSlaveMonitor* object and read the array structure. As demonstrated in Section 5.4.1, when instancing a pointer to the *MormSlaveMonitor* class, the component must also pass a pointer to a method

that is invoked when the *MormSlaveMonitor's* array struct is updated. In the MORM-MQSoC implementation, the *MormSlaveMonitor* object is instantiated by the *MormLocalMasterSensor* and *LocalMasterDecisor* classes.

LocalMasterDecisor class implements the decision-making logic at the intra-cluster actuation level. While maintaining the same functionality, this class decouples from the kernel the MORM's decision-making logic. This approach allows, for example, that the decision logic to be replaced by another heuristic without interfering with the behavior of other elements of the adaptive service, such as sensors and actuators. The *LocalMasterDecisor* class incorporates in specific methods the Algorithm 5 which defines the MORM's decision-making logic. The *LocalMasterDecisor* class actuates in the SPs by instancing the *DvfsActuator* class, one for each SP in the cluster.

DvfsActuator class is a specialization of the *UpDownActuator* class (detailed in Section 5.2.2). The *UpDownActuator* class implements a standard interface that allows control of the applicable actuation levels in an actuator. The *DvfsActuator* class implements three actuation levels following the DVFS model detailed in Section 6.1.3. Figure 6.13 shows the class diagram of the *ClusterPowerModeActuator* class. As shown, it is derived from the *UpDownActuator* class, having as template parameters the *DVFSType* class as parameter for *DataType* and the *DvfsValue* enumerator class as parameter for *T*. The *DvfsActuator* class uses the *DvfsType* as the actuation data format. The topic named */actuator/pe/<id>/dvfs* identifies the message flow between the *DvfsActuator* and *DvfsEffector* pair, where *<id>* is the SP identification. As shown in Figure 6.8, both *MormSlaveMonitor*, *LocalMasterDecisor* and *DvfsActuator* classes are instantiated in the LM kernel.

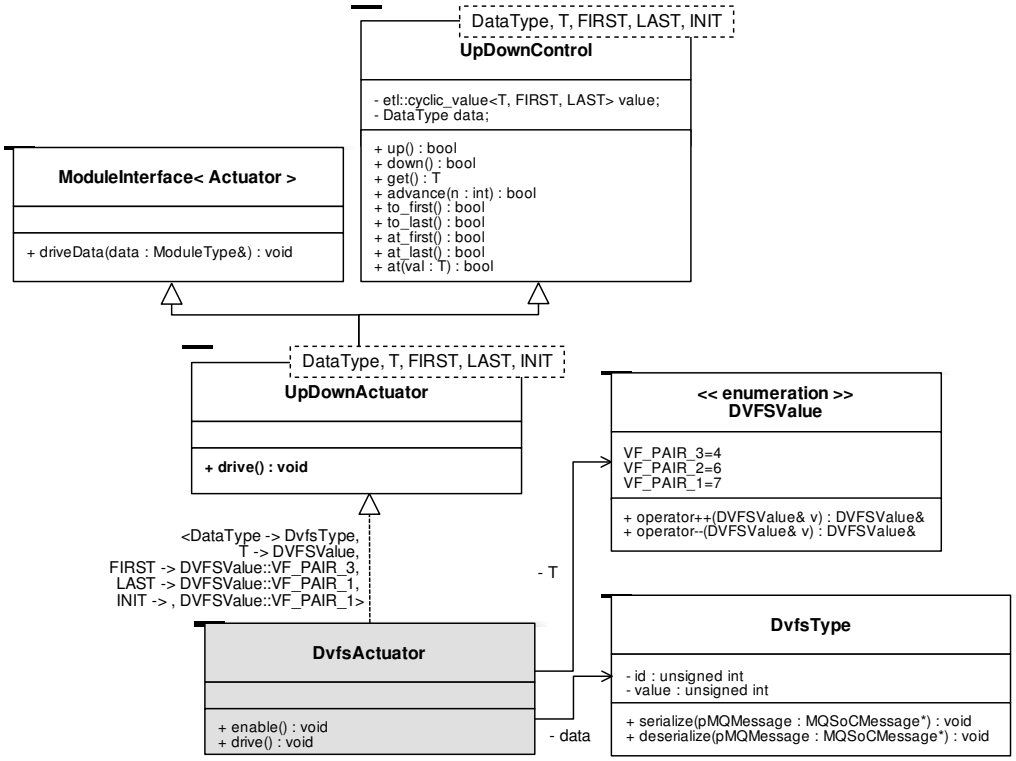


Figure 6.10: DvfsActuator Class Diagram.

DvfsEffector class is the pair of the *DvfsActuator* class. It receives the actuation value from the *DvfsActuator* object instantiated in the LM and applies this actuation value in the SP using an *HSAL* primitive. Figure 6.11 shows the code snippet of the *doit* method of the *DvfsEffector* class responsible for calling the *deserialize* method of the *DvfsType* class and applying the actuation value

stored by the variable named "value". The *Modules* class instantiates a *DvfsEffector* object for each SP.

```
void DvfsEffector::doit(MQSoCMessage* pMQMessage)
{
...
  this->data.deserialize(pMQMessage);
  HAL_OS_EFFECTORS_setDVFS(this->data.value);
..
}
```

Figure 6.11: Code snippet of the *doit* method of the *DvfsEffector* class.

6.2.2. MORM-MQSoC Adaptative Service for the Global Master PE.

The adaptive service that performs decision-making in the inter-cluster domain is located in the Global Master PE (GM). The Section 6.1.4 details the algorithms that execute the decision making in the implementation of the MORM-C. MORM-MQSoC implementation has the same functionality. The distinctions are in the model of communication between the elements of the adaptive service and the modularization of the codes using our middleware-based approach. APPENDIX G shows the directory tree with the set of files implemented by MORM-MQSoC. The adaptive service for the GM following our middleware-based approach is comprised of these classes, described below: ***MormLocalMasterSensor***, ***MormLocalMasterMonitor***, ***GlobalMasterDecisor***, ***ClusterPowerModeActuator***, and ***ClusterPowerModeEffector***. Figure 6.6 shows a general overview of the relation between these classes representing an *ODA* loop.

MormLocalMasterSensor class defines the LM sensor object. The *MormLocalMasterSensor* class uses the *MormLocalMasterType* class as the sensor data format. The *MormLocalMasterType* class contains the *serialize* and *deserialize* methods, which encapsulate/decapsulate the sensor data in/from the message payload. The *MormLocalMasterSensor* class contains the *updateStatus* method, which defines where the current information of the sensor data is extracted to be serialized. Figure 6.12 shows the code snippet of the *serialize*, *deserialize* and *updateStatus* methods defined in the respective classes. The *MormLocalMasterSensor* class updates the status of sensor data by accessing the respective data in variables stored by the *LocalMasterDecisor* object instance. It also calls the *HAL_OS_OS_predictPowerMigration* primitive via the *HSAL* interface which is handled by the LM kernel. This function estimates the power spent by a given migration of application tasks. The *GlobalMasterDecisor* class uses the sensor data through the instantiation of its *MormLocalMasterMonitor* pair. The topic name */sensor/pe/<id>/mormLocalMaster* identifies the message generated by the *MormLocalMasterSensor* class, where *<id>* is the identification of each cluster. The *Modules* class defines in its composition an instance of the *MormLocalMasterSensor* class for each LM.

MormLocalMasterMonitor class is the pair of the *MormLocalMasterSensor* class. When subscribing to the sensor data of all clusters using the "+" wildcard symbol, messages sent in this topic by the respective LM are received by the *MormLocalMasterMonitor* class. The *MormLocalMasterMonitor* deserializes the received messages using the *deserialize* method of the *MormLocalMasterType* class. An array struct in the *MormLocalMasterMonitor* class stores the deserialized LM's sensor data. Through this approach, decision-makers or other components that want to use the LM's sensor data must use a pointer to the *MormLocalMasterMonitor* object and read the array structure. The *MormLocalMasterMonitor* object is used by the *GlobalMasterDecisor* class.

```
void MormLocalMasterType::serialize(MQSoCMessage* pMQMessage)
{
  ...
  w = mp_encode_uint(w, this->id);
  w = mp_encode_uint(w, this->energyLeak);
  w = mp_encode_uint(w, this->energyTotal);
  w = mp_encode_uint(w, this->powerModeCluster);
  w = mp_encode_uint(w, this->powerEstimatorError);
  w = mp_encode_uint(w, this->slackTime);
  ...
}

void MormLocalMasterType::deserialize(MQSoCMessage* pMQMessage) {
...
  this->id = mp_decode_uint(&r);
  this->energyLeak = mp_decode_uint(&r);
  this->energyTotal = mp_decode_uint(&r);
  this->powerModeCluster = mp_decode_uint(&r);
  this->powerEstimatorError = mp_decode_uint(&r);
  this->slackTime = mp_decode_uint(&r);
...
}

void MormLocalMasterSensor::updateStatus() {
...
  this->data.energyLeak = pLocalMasterDecisor->totalPEEnergyHistory.getEnergyClusterDif_leakage();
  this->data.energyTotal = pLocalMasterDecisor->totalPEEnergyHistory.getEnergyClusterDif_total();
  this->data.powerModeCluster = pLocalMasterDecisor->getCurrentPowerMode();
  this->data.powerEstimatorError = HAL_OS_predictPowerMigration(HAL_OS_GetClusterUID(),
  pLocalMasterDecisor->getCurrentPowerMode());
  this->data.slackTime = pLocalMasterDecisor->processorHistory.get_average_slack_time_cluster();
...
}
```

Figure 6.12: Code snippet of the *serialize* and *deserialize* methods of the *MormLocalMasterType* class, and the *updateStatus* method of the *MormLocalMasterSensor* class.

GlobalMasterDecisor class implements the decision-making logic at the inter-cluster actuation level. While maintaining the same functionality, this class decouples from the kernel all the MORM's decision-making logic. The *GlobalMasterDecisor* class incorporates in specific methods the algorithms 4 and 3. We perform some modifications in the algorithms of the MORM-C decision logic in order to decouple those actuation values that are platform specific. With this design decision, the *GlobalMasterDecisor* class could be used as decision-making method of any actuation object derived from *UpDownActuator* class. The *GlobalMasterDecisor* class actuates in the clusters by instancing the *ClusterPowerModeActuator* class, one for each cluster in the system.

ClusterPowerModeActuator class is a specialization of the *UpDownActuator* class (detailed in Section 5.2.2). The *DvfsActuator* class implements two levels of actuation: *Performance* - where the SPs in the cluster are configured as single-task execution model; *Energy* - where the SPs are configured as multitask execution model, with the maximum number of tasks defined at design time. In both actuation levels, tasks migration can be performed to achieve one or another execution model, as detailed in the process of tasks mapping/remapping in Section 6.1.4. Figure 6.13 shows the class diagram of the ClusterPowerModeActuator class. As demonstrated, it is derived from UpDownActuator class, having as template parameters the *ClusterPowerModeType* class as parameter for *DataType* and *PowerModeValue* enumerator class as parameter for *T*. The *ClusterPowerModeActuator* class uses the *ClusterPowerModeType* as actuation data format. The topic named *"/actuator/cluster/<id>/powerMode"* identifies the message flow between the *ClusterPowerModeActuator* and *ClusterPowerModeEffector* pair, where *<id>* is the identification of the cluster. Both *MormLocalMasterMonitor*, *GlobalMasterDecisor* and *ClusterPowerModeActuator* classes are instances of the GM kernel.

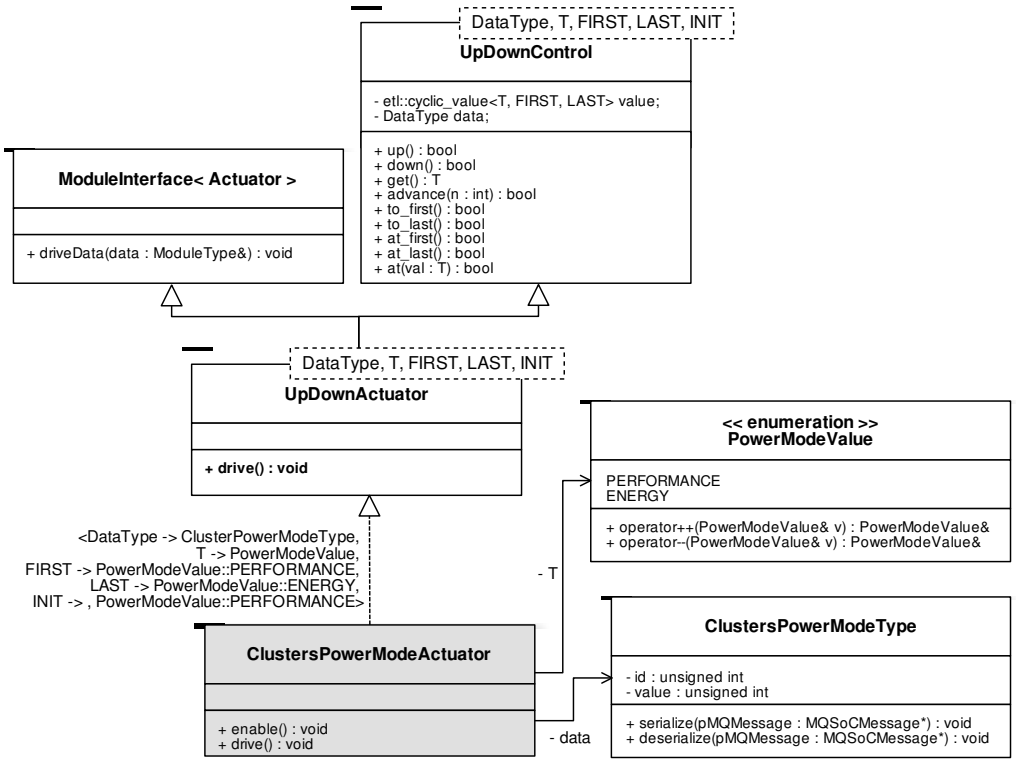


Figure 6.13: ClusterPowerModeActuator Class Diagram.

ClusterPowerModeEffector class is the pair of the *ClusterPowerModeActuator* class. It receives the actuation value from the *ClusterPowerModeActuator* object instantiated in the GM and applies this actuation value in the LM kernel by using an *HSAL* primitive. Figure 6.14 shows the code snippet of the *doit* method of the *ClusterPowerModeEffector* class responsible for calling the *deserialize* method of the *ClusterPowerModeType* class and applying the actuation value represented by the variable named "value". The *Modules* class instances one *ClusterPowerModeEffector* object for each LM.

```
void ClusterPowerModeEffector::doit(MQSoCMessage* pMQMessage)
{
...
  this->data.deserialize(pMQMessage);
  HAL_OS_EFFECTORS_setClusterPowerMode(this->data.id, this->data.value);
...
}
```

Figure 6.14: Code snippet of the *doit* method of the *ClusterPowerModeEffector* class.

6.3. Evaluation

This section presents the evaluation performed on MORM-C and MORM-MQSoC self-adaptive systems. Both self-adaptive systems run on the HEMPS platform detailed in Section 6.1. To include the middleware in the HEMPS platform, it was necessary to implement HSAL directives which represent the interface from the middleware to the kernel. We also included the middleware initialization in the kernel boot process. The middleware initialization includes the initialization of the base middleware structures (Chapter 4) and modules extension (Chapter 5).

Refactoring the MORM-C self-adaptive system originally presented at [MdSR+19] allows us to compare and contrast the advantages and disadvantages of using middleware support for the development of self-adaptive systems. While more modular code seems more understandable and easier to fix and expand, we need to assess this perception through objective evaluation metrics that can be extracted from both approaches. We have summarized performance/energy and software quality metrics extracted from MORM-C and MORM-MQSoC implementations as follows: i) Performance/energy metrics: Execution Time, Energy, Power and CPU Utilization; ii) Software quality metrics: Cyclomatic Complexity, Interface Complexity, Function Complexity, Lines of Code, Effective Lines of Code, Logical Lines of Code and Parameters. Sections 6.3.1 and 6.3.2 detail the performance/energy metrics and present the results obtained upon these metrics. Section 6.3.3 and 6.3.4 detail the software quality metrics and discusses the results obtained upon these metrics.

6.3.1. Performance/Energy Metrics

The hypothesis to be verified in this evaluation is the following:

Hypothesis 6.3.1. *The overhead implied by the middleware-based design approach for the development of a self-adaptive system has low impact on the applications performance and the energy spent by the system.*

To test the Hypothesis 6.3.1, we compared the two approaches by stimulating the system with an application benchmark. Since the MORM self-adaptive system aims to adapt the system in order to comply with a power cap, we have to stimulate the platform and observe the behavior of the system according to some performance and energy metrics. The application benchmark consists of the following applications: DTW (6 tasks), AES (5 tasks), MPEG (5 tasks), and Synthetic (communication-bound application, 6 tasks). The metrics extracted are as follows, depending on the performed experiment: application execution time, workload execution time, energy, power and CPU utilization. The metrics are extracted from a clock-cycle accurate RTL SystemC description of the HeMPS system.

In the experiments detailed in Section 6.3.2, we inserted the applications in the system in a certain order and entry time to cause a behavior in the system against a typical workload. Both MORM-C and MORM-MQSoC self-adaptive systems follow the same assumptions, which are: i) sensor's data samples in the SPs are generated and sent every 250 Kticks[9]; ii) when an LM receives an SP sample, it generates an LM's sensor sample that is immediately sent to the GM; iii) an LM performs its decision-making process whenever it receives a sensor sample from an SP; iv) the GM performs its decision-making process whenever an application request mapping or an application finishes its execution; v) the order and time of entry of applications into the system are the same; vi) the power cap is set to 180mW; vii) the SPs running no tasks are considered off (their sensors do not generate data while off).

6.3.2. Performance/Energy Results

This section discusses the results obtained by MORM-C and MORM-MQSoC self-adaptive systems upon the performance/energy metrics detailed in Section 6.3.1. Figure 6.15 shows the average power results for MORM-C and MORM-MQSoC running a typical workload. We stand out

[9] 1 Ktick corresponds to 1000 clock cycles.

in Figure 6.15 instants of time where the applications are inserted in the system (red vertical bars), snapshots taken for energy evaluation (blue vertical bars) and the time instant where the applications ended (green vertical bars). The applications enter in the system representing a dynamic load of applications in three instants of time in the following order: i) two long applications enter at 1000 Kticks; ii) two short applications enter at 5200 Kticks; iii) other two short application enters at 9300 Kticks. Figure 6.16 shows the average system's slack time for MORM-C and MORM-MQSoC at the same experiment. Slack time is the time (in percentage) where the SPs are idle.

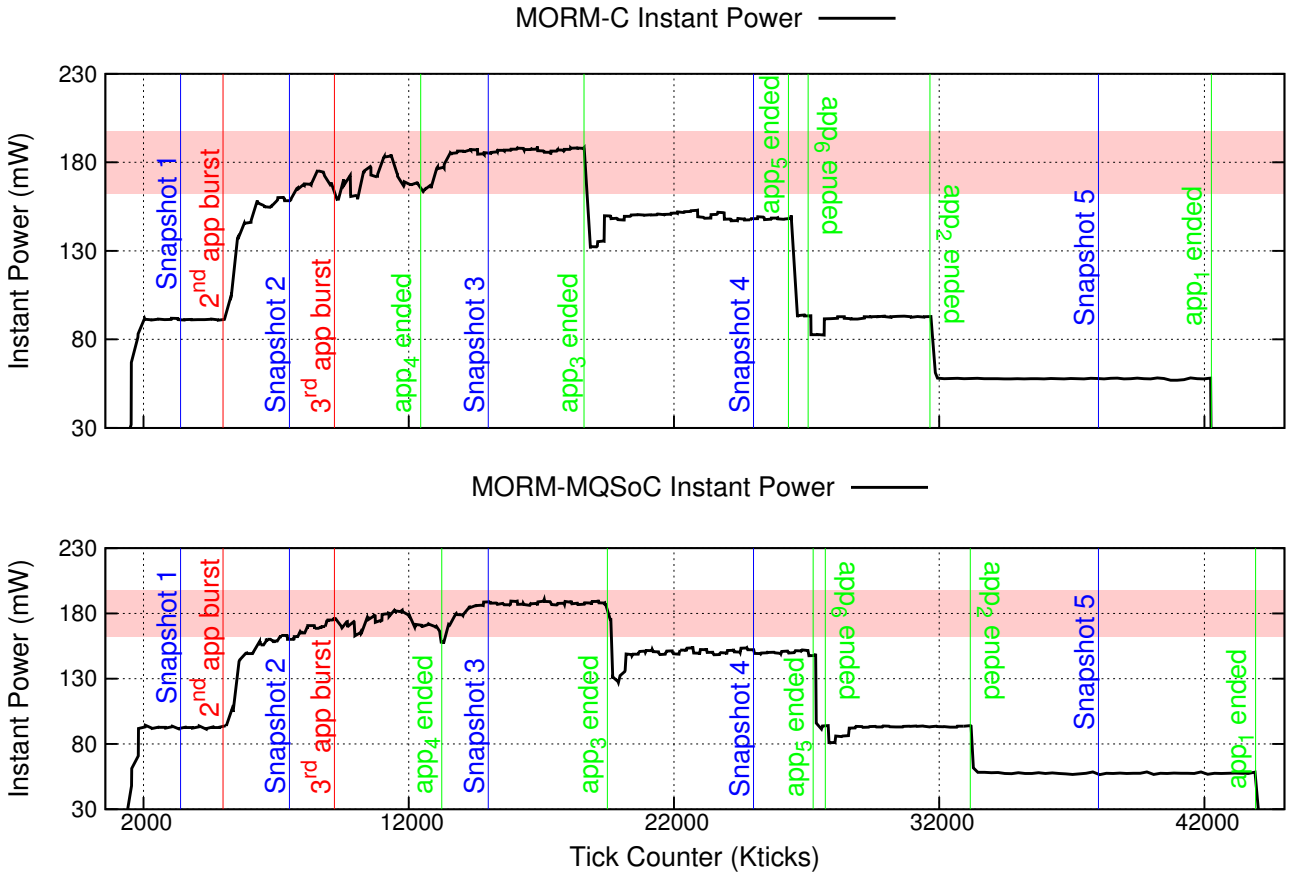


Figure 6.15: Average power results for MORM-C and MORM-MQSoC running typical workload.

Table 6.5 presents the performance and energy results for both implementations extracted in each snapshot: (i) number of executed iterations (app_i *Iter.*), which corresponds to the performance of the applications (higher value correspond to better performance); (ii) total energy consumed by the system (*Energy*); (iii) average utilization of the SPs (*CPU Util.*) including those turned off. We extracted the values of the sample immediately prior to the respective snapshot. The values between parentheses is the overhead in percentage based on the samples, considering the following: negative value for app_i *Iter* means that the number of iterations performed for the application decreased by that percentage; positive overhead for *Energy* means that the energy consumption increased by that percentage.

Firstly, we analyze the adaptation logic of both MORM-C and MORM-MQSoC self-adaptive systems. We can observe that after the first burst of applications (Figure 6.15), the instant power is around 90mW. After the second burst, the power enters the region of the power cap and even after the third burst of applications, the power remains below the cap for both adaptive

[10] MORM-C
[11] MORM-MQSoC
[12] MORM-C
[13] MORM-MQSoC

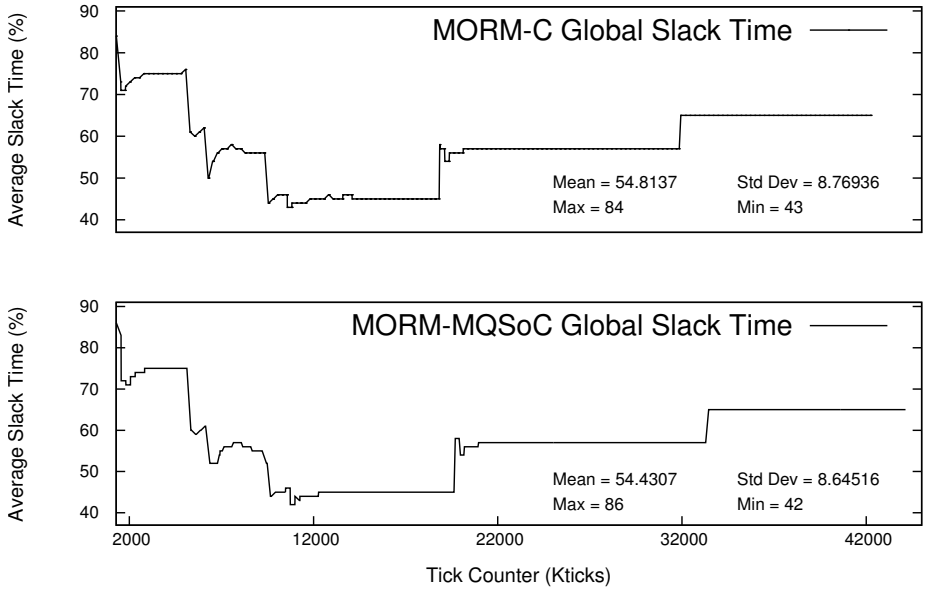


Figure 6.16: Average Slack Time for MORM-C and MORM-MQSoC running typical workload.

Table 6.4: Applications Execution Time.

Application	**Execution Time (Kticks)**		**MQSoC Overhead**
	BASE[10]	*MQSoC*[11]	
$app1_{long}$	42256	44124	4.4%
$app2_{long}$	31846	33272	4.5%
$app3_{short}$	18808	19587	4.1%
$app4_{short}$	12653	13146	3.9%
$app5_{short}$	26473	27346	3.3%
$app6_{short}$	27058	27604	2.0%
Total Energy (mJ)	19606.4	20766.1	5.9%

services. The adaptation logic switches the operating mode of the Cluster 0 from performance to energy mode and changes the vf-pair of the SPs to a lower energy consumption value, as can be observed in Figure 6.17, Snapshot 2. With the third burst of applications, the adaptation logic switches the operating mode of the Cluster 3 to energy, as Snapshot 3. The app5 is already ended in this instant. We can observe that the adaptation logic has switched the Cluster 0 to performance mode because the logic measures that the power increment would not exceed the power cap. Most of all, the system execution does not exceed the power cap at any time for both MORM-C and MORM-MQSoC scenarios.

We review some possible behaviors that we may find in the analysis of the performance and energy results that we will discuss below. MORM-C uses low-level kernel primitives to send messages between the elements that comprise the adaptive service. MORM-MQSoC does that through middleware's publish-subscribe primitives. Thus, we should find some performance overhead since the middleware performs additional processing to handle features such as serialization, protocol stack and topic verification, as detailed in Sections 4 and 5. We could see the performance overhead in the results as extended time to finish the applications, which should result in additional power consumption. In addition, although MORM-C and MORM-MQSoC have been designed with the

Table 6.5: Performance and energy results.

Metric	Snapshot 1		Snapshot 2		Snapshot 3		Snapshot 4		Snapshot 5	
	$BASE^{12}$	$MQSoC^{13}$	$BASE^{12}$	$MQSoC^{13}$	$BASE^{12}$	$MQSoC^{13}$	$BASE^{12}$	$MQSoC^{13}$	$BASE^{12}$	$MQSoC^{13}$
$app1_{long}$ Iter.	263	253(3.8%)	720	705 (2.1%)	1317	1287 (2.3%)	2067	2001 (3.2%)	3504	3324 (5.1%)
$app2_{long}$ Iter.	45	43 (4.7%)	139	133 (4.3%)	311	299 (3.8%)	541	516 (4.6%)	-	-
$app3_{short}$ Iter.	-	-	292	281 (3.9%)	1285	1211 (5.8%)	-	-	-	-
$app4_{short}$ Iter.	-	-	106	115 (8.1%)	-	-	-	-	-	-
$app5_{short}$ Iter.	-	-	-	-	460	400 (13.0%)	1628	1529 (6.1%)	-	-
$app6_{short}$ Iter.	-	-	-	-	82	79 (3.7%)	276	266 (3.6%)	-	-
Energy (mJ)	713.6	720.6 (1.0%)	2657.9	2794.9 (5.1%)	7891.2	8149.1 (3.3%)	14969.0	15251.8 (1.9%)	18625.3	19349.8 (3.9%)
CPU Util. (%)	25	25	43	44	55	55	43	43	35	35

same set of sensors, decision logic and actuators, a particular behavior observed may be different due to a possible time deviation in a given sample of the system's state.

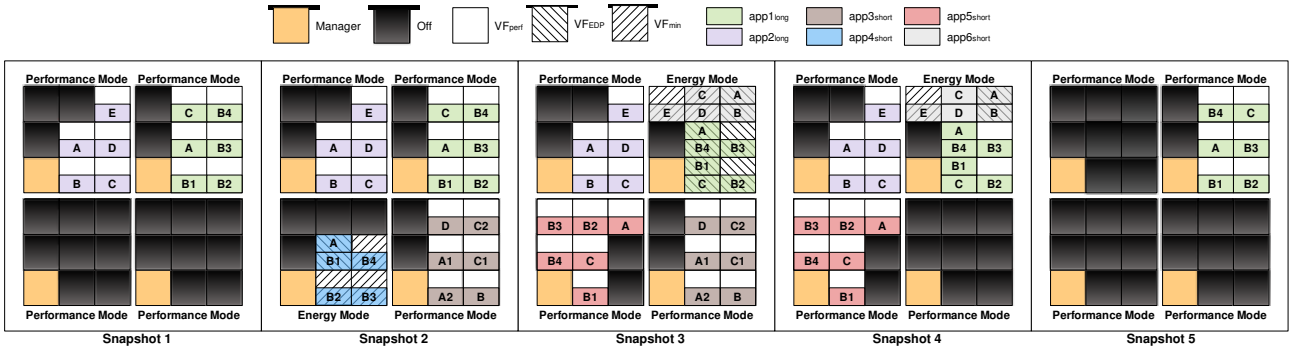


Figure 6.17: Task mapping showing the cluster mode and vf-pair at each snapshot (Same behavior for both MORM-C and MORM-MQSoC).

Analyzing the performance and energy results, we can observe that the system execution begins with a start-up time required for the MORM-MQSoC to advertise and subscribe the topics that identify the set of sensor/monitor pairs and actuators/effectors pairs. This start-up time is the first additional execution time performed by MORM-MQSoC. The middleware start-up finishes at instant 4937 Kticks, including the MORM-MQSoC start-up. The sensor data begins to be sampled after the first application mapping, which happens at instant 1000 Kticks. The performance overhead induced by MORM-MQSoC on the number of iterations of the applications ranges from 2.1% to 13%, depending on the snapshot. The execution time of the whole execution scenario for MORM-MQSoC is 44202 Kticks, which represents an overhead of 4.5% compared to MORM-C. The difference in CPU utilization (*CPU Util.* in Table 6.5) reached a maximum of 1% in Snapshot 2. As viewed in Figure 6.16, the behavior of the system's slack time is similar for both. The total energy spent by the MORM-MQSoC is 20766.1 mJ, which represents an overload of 5.9% compared to MORM-C.

Memory Footprint Analysis

Tables 6.6 and 6.7 show the memory footprint required to store the software for MORM-C and MORM-MQSoC. We analyze the memory footprint size separately for each software component of the adaptive service. MORM-C consists of the base kernel and the MORM-C adaptive logic. MORM-MQSoC is composed of the same base kernel added to the base middleware, the modules extension, and the MORM-MQSoC adaptive logic. Analyzing the footprint size only for MORM software, the overhead of MORM-MQSoC over MORM-C is 27.5% for master software and 128% for the slave. The base software required to incorporate the MORM-MQSoC, consisting of the base middleware and the modules extension, presents an overhead of 52.9% for the master software and 69.7% for the slave. The total overhead of the MORM-MQSoC considering all software components (*Total Size* row) is 44.5% for the master software and 78% for the slave.

The memory footprint overhead is caused in part by the use of OOP features, such as virtual functions and polymorphism. Virtual functions are resolved at runtime. When using virtual functions, the compiler adds additional code to maintain the structures needed to handle pointers and call methods of derived classes. The remaining overhead is caused by other features designed to make the programming of self-adaptive systems easier and more transparent to the user, such as payload serialization (Section 4.4), use of strings and wildcards to represent the publish-subscribe topic (Section 5.3), hardware/software abstraction layer to facilitate the portability of the middleware to other platforms (Section 4.3.4), and protocol stack to facilitate the addition of new layers in the protocol stack handled by the middleware (Section 4.3.4). It's the cost of a code that is more reusable and easier to maintain.

Table 6.6: MORM-C Memory Footprint.

Software Component	**Footprint (KB)**	
	Master	*Slave*
Base Kernel	24.2	15.2
MORM-C	12	2.5
Total Size	36.2	17.7

Table 6.7: MORM-MQSoC Memory Footprint.

Software Component	**Footprint (KB)**	
	Master	*Slave*
Base Kernel	24.2	15.2
Base Middleware	10.5	8.3
Modules Middleware Extension	2.3	2.3
MORM-MQSoC	15.3	5.7
Total Size	52.3	31.5

Internal timing profile

We know that by adding a new level of processing to the software stack, in this case, the middleware, we would cause a performance overhead compared to the MORM-C approach. However, can we quantify the processes that most demand time within middleware processing? To do this, we perform a timing profile of the internal middleware processes and compare the results with MORM-C. In this experiment, we measure the number of clock cycles (ticks) spent by each component of both MORM-C and MORM-MQSoC. We perform the same analysis for the inter-cluster and intra-cluster adaptive service. Figures 6.18 and 6.19 show the components of the intra-cluster and inter-cluster adaptive services, where we place between parentheses the time spent by their respective methods/functions. In these figures, we show in (a) the results for MORM-MQSoC and in (b) the results for MORM-C.

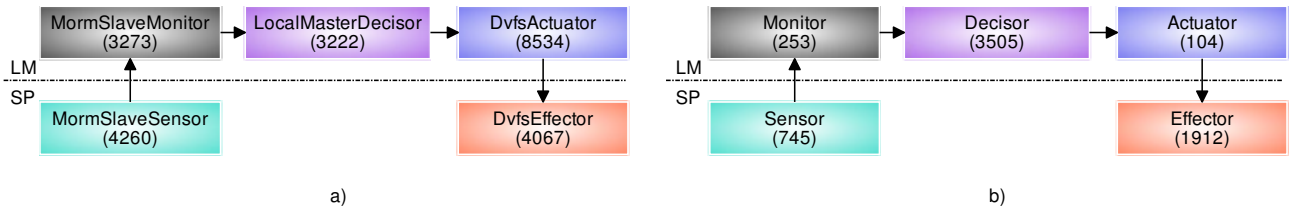


Figure 6.18: Time spent by each component of the intra-cluster adaptive service for: a) MORM-MQSoC b) MORM-C.

As we have shown in the previous experiment, MORM-MQSoC presents an execution time overhead of 4.5% in the simulated scenario. Since the application tasks run on the SPs (and not on the LMs and GMs), the overhead is mainly due to the middleware processing performed on the SPs. The process of generation of the sensor data in SPs and deliver to the DMNI to be sent by NoC is identified in Figure 6.18 by the MormSlaveSensor component. As defined at design-time, the sensor data in SP is generated intermittently every 250 Kticks. This means that every 250 Kticks, the

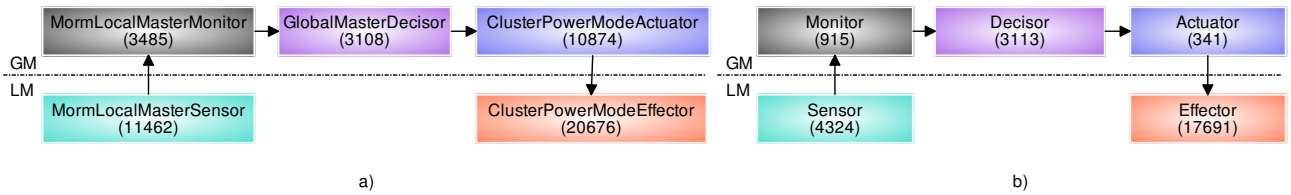


Figure 6.19: Time spent by each component of the inter-cluster adaptive service for: a) MORM-MQSoC b) MORM-C.

middleware uses the processor for additional 3515 ticks than the same process in MORM-C. Thus, the application tasks that are running at the same SP do not use the processor in this time period, causing a delay in their execution time. This delay corresponds to an overhead 1.6%. The remaining overhead is caused by network traffic, as detailed in the network traffic analysis as follows.

Network Traffic Analysis

This section evaluates the network traffic for the same typical workload. The variation in network traffic is mainly due to the difference in message size between MORM-MQSoC and MORM-C. MORM-MQSoC uses the additional packet header for handling the publish-subscribe layer of the protocol stack (as detailed in Section 4.3.4). We analyze the following metrics: a) *router injection*, that is the rate of utilization of the buffer in the local port measured in percentage; b) *router congestion*, that is the rate of utilization of the input buffer in the non-local ports (north, south, west, and east); c) *Dif Time*, that is the time for transmitting a message from a source PE to a target PE.

For both *router injection* and *router congestion* metrics, the routers report the percentage of cycles that the buffer stays with high utilization ($>= 75\%$) in the epoch. For example, a rate of 25% means that in 25% of the clock cycles the buffer usage has been occupied by 75% or more of its capacity.

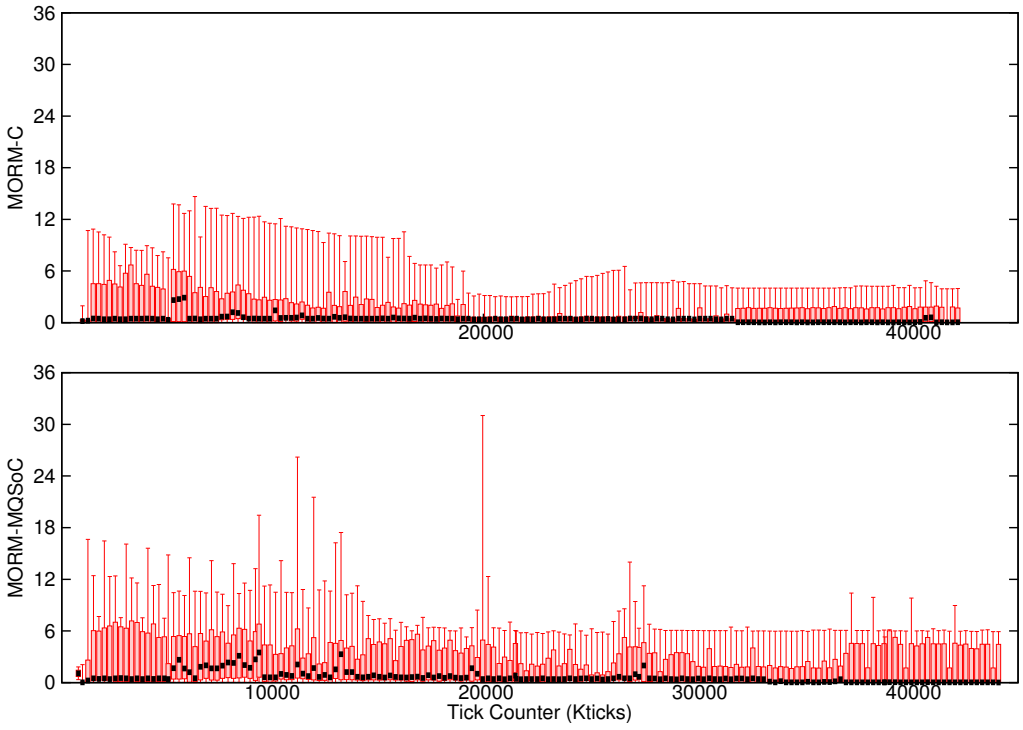


Figure 6.20: Router Injection (%), presented in the Y-axis.

Figure 6.20 shows a box plot graph to the router injection measured for MORM-C and MORM-MQSoC during the execution time of the typical workload. The graph shows a box plot bar for each epoch along the execution time. Each box plot bar presents the minimum, first quartile, median, third quartile, and maximum values calculated on the samples of all SPs at the epoch. The median value (black points in the graphs) is lower than 4% for all samples. However, MORM-MQSoC presents a higher variance in the samples what can be viewed in the maximum values in the graphs. For the worst-case, the injection rate is 31.0% for MORM-MQSoC and 14.6% for MORM-C. Considering the average mean values, MORM-MQSoC router injection rate is 50.9% higher than MORM-C rate (0.63% and 0.42% for MORM-MQSoC and MORM-C, respectively). The higher injection rate in MORM-MQSoC contributes to the execution time overhead of 4.5% in the simulated scenario.

Figure 6.21 shows the box plot graph related to the router congestion rate. The median router congestion is lower than 2% for all cases. MORM-MQSoC reaches 16.2% at worst-case, while MORM-C reaches 3.1%. Considering the average mean values, the router congestion rate is 0.182% for MORM-MQSoC and 0.175% for MORM-C.

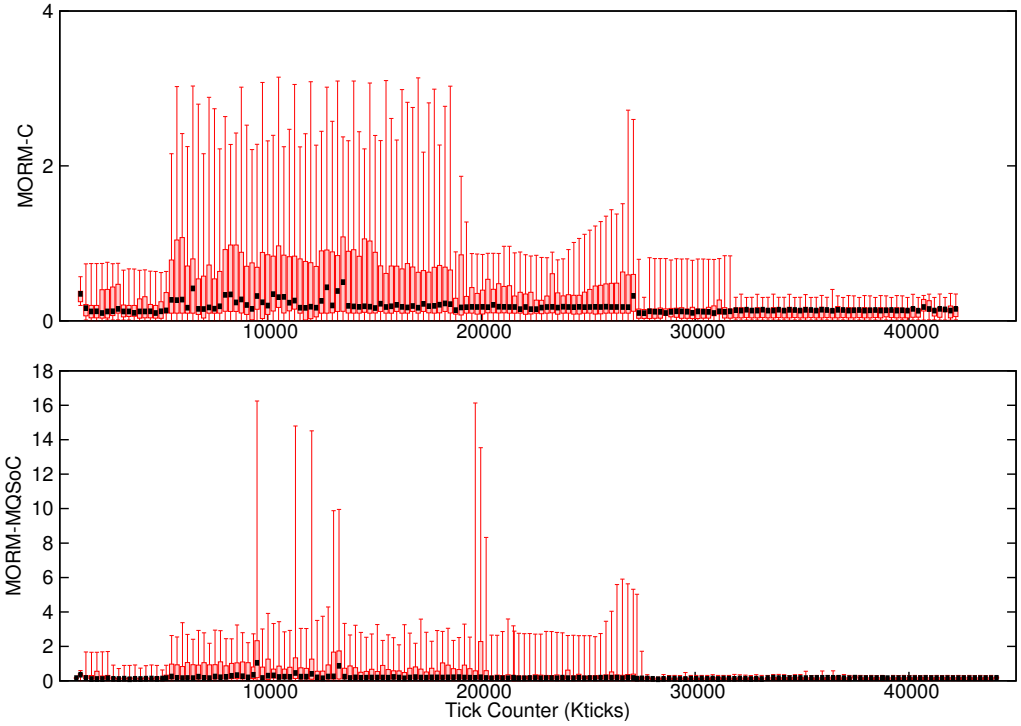


Figure 6.21: Router Congestion (%), presented in the Y-axis (note that the scale of the Y-axis is not the same for better viewing purpose).

The next experiment evaluates the delay in the delivery of messages generated in the self-adaptive system. We analyze the delay in the delivery of messages from the SPs to their respective LMs. The metric used is *Dif Time*, that represents the time for transmitting a message from a source PE to a target PE, considering only the hardware elements (DMNI, router buffer and NoC). Figure 6.22 shows the *Dif Time* for the sensor data messages generated by SPs. A line in the graph corresponds the average of all *Dif Time* samples in an epoch for the respective cluster. A point in the graph is the *Dif Time* of a given SP in that cluster. Figure 6.23 shows the summarized *Dif Time* for all messages of the SP's sensor data.

The *Dif Time* median is 6494 Kticks and 7927 Kticks for MORM-MQSoC and MORM-C, respectively. Although MORM-MQSoC presents a lower *Dif Time* median, it presents a higher delay to process the received message in the middleware and also when a DVFS actuation process is

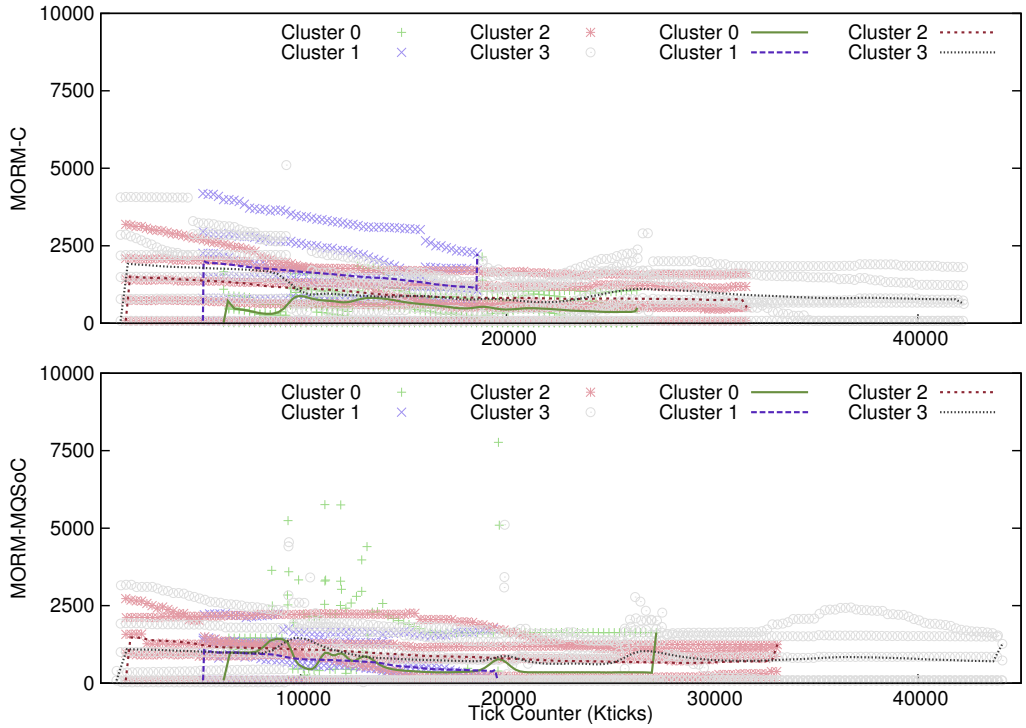


Figure 6.22: *Dif Time* (Kticks), presented in the Y-axis, showing the arrival delay of the sensor data messages from SPs to LMs.

triggered by the decisor. As showed in Figure 6.18, MORM-MQSoC (MormSlaveMonitor) spends 3273 ticks to process the received message, while MORM-C (Monitor) spend 253 ticks. This processing is related to the updating of SP statistics in the LM. In the case of a DVFS actuation, the LM spends 8534 ticks to generate the actuation message, while MORM-C spend 104 ticks. One consequence of this delay is that another received message, stored in the buffer router, is not consumed until the entire process described here is finished. This can have a cascading impact, since other PEs cannot send messages through this router if it becomes full. The same reasoning we can apply to the LM of the Cluster 0 which also has GM functions. In addition to the delay caused by the reception of a SP sensor message, the LM (which is also GM) may also be slow to consume messages in periods of time in which it is processing a reception of a LM sensor data or when is triggered an actuation in a given cluster's operating mode. For these reasons, the *Dif Time* worst case achieves 77786 ticks for MORM-MQSoC and 51031 ticks for MORM-C.

6.3.3. Software Quality Metrics

This section presents the metrics used to test the following hypothesis:

Hypothesis 6.3.2. *The use of middleware-based design approach improves the quality of the self-adaptive system software.*

Measuring the quality of software can be done by *control* or *prediction* metrics. *Control* metrics are associated with software development processes from its design until a considerable period of time in production. Examples of control metrics are the average effort and time required to fix reported bugs. Control metrics can only be extracted after the software is developed by reviewing both bugs fixing and feature extending processes. *Prediction* metrics help predict the characteristics of the software. Examples of prediction metrics are cyclomatic complexity and interface complexity.

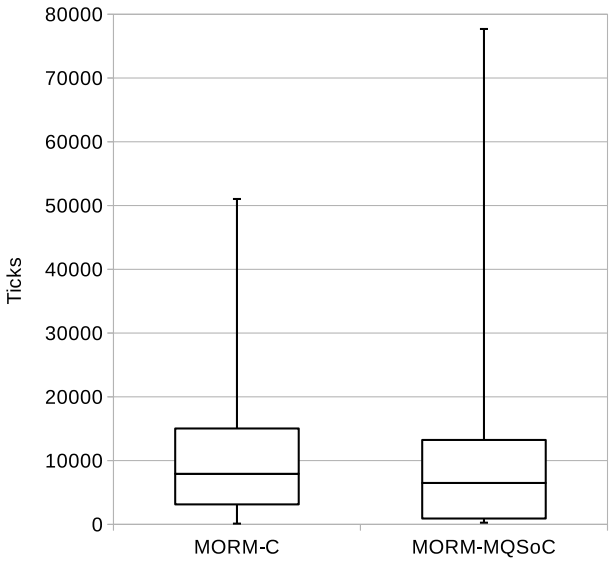


Figure 6.23: Box Plot of the summarized *Dif Time* for transmission of the SP's sensor data.

Software metrics can influence the designer to choose which design approach to use to deploy a given software. We do not have a complete software development cycle guaranteed by control metrics available to promote the use of a modular and self-contained approach such as that used in MORM-MQSoC. Thus, we evaluate the software through predictor metrics of quality.

We select predictor metrics that could be extracted from both approaches used in MORM-C and MORM-MQSoC. We extract the metrics using the M-Squared Technologies' Resource Standard Metrics (RSM) tool. For each source file of MORM-C and MORM-MQSoC implementations, we use RSM to extract the following metrics: i) *Cyclomatic Complexity*; ii) *Interface Complexity*; iii) *Function Complexity*; iv) *Lines of Code*; v) *Effective Lines of Code*; vi) *Logical Lines of Code*; vii) *Parameters*.

Cyclomatic Complexity (CC) metric is the number of linearly independent code paths through the source code of a software [McC76]. A software with complex control flow requires more testing to achieve good code coverage and is less easy to maintain. This metric can also indicate the number of test cases required in the unit testing to fully validate a function/method. CC can be applied to functions, methods, classes or for projects as a whole. A small value for CC means less logical complexity, implying less risk in its modification and easier to understand. The RSM tool calculates the CC value according to McCabe [McC76]. The calculation is based on a representation of the function control flow. The control flow represents a function such as a graph consisting of nodes and edges. CC is defined as Equation 6.12, where P is the number of predicted nodes (nodes that contain conditions) in the control flow graph.

$$CC = P + 1 \tag{6.12}$$

Interface Complexity (IC) is a software quality metric that sums the number of input parameters for a function and the number of return states for that function. This metric can be useful when analyzing the complexity of a function's interface. IC can be applied to functions, methods, classes, or for projects as a whole. A small value for IC means less interface complexity. IC is defined as Equation 6.13, where *Par* is the number of input parameters for the function and *Ret* is the number of return states for the function.

$$IC = Par + Ret \tag{6.13}$$

Function Complexity (FC) is the sum of the Cyclomatic Complexity (CC) and the Interface Complexity (IC) of a function. This metric can be useful when analyzing the complexity of a function. A small value for FC means less function complexity. FC is defined as Equation 6.14.

$$FC = CC + IC \tag{6.14}$$

Lines of Code (LoC) is a software quality metric that represents the number of lines of code for a function. This metric considers all the lines that compose the code of a function, including comments, blacks, and parentheses. LoC can be applied to functions, methods, classes or to projects as a whole.

Effective Lines of Code (eLoC) is a software quality metric that represents the number of effective lines of code for a function. This metric excludes comments, blanks, or stand-alone keys or parentheses. eLoC can be applied to functions, methods, classes or to projects as a whole.

Logical Lines of Code (1LoC) is a software quality metric that represents the number of code's logical lines for a function. This metric considers only those lines of code with statements ending with a semicolon. 1LoC can be applied to functions, methods, classes or to projects as a whole.

Parameters is a software quality metric that represents the number of parameters of a function. *Parameters* can be applied to functions, methods, classes or to projects as a whole.

6.3.4. Software Quality Results

This section discusses the results obtained by MORM-C and MORM-MQSoC self-adaptive systems following the software quality metrics detailed in Section 6.3.3. We extract the software quality metrics using the RSM tool [MSq18] having as object of evaluation the source code of the MORM-C and MORM-MQSoC self-adaptive systems. The objective is to test the Hypothesis 6.3.2. To do this, we have to differentiate the MORM source code from the rest of the kernel and middleware. The MORM-MQSoC's source code is simple to identify since all its code is composed of the files of its classes, as demonstrated in Sections 6.2.2 and 6.2.1. However, the differentiation of the MORM-C's source code from the rest of the kernel software is not trivial. To differentiate it, we identify the methods used by MORM-C in kernel code and surround them with preprocessing macros. In addition, the platform contains two kernel versions corresponding to the master and slave kernels. The master kernel is the software used in GM and LM. The slave kernel is used in SP.

Equations 6.15 and 6.16 define the computation of a given software quality metric (QM) for the MORM-C source code, where: i) $QM(KERNEL_C^{Master}+MORM_C^{Master})$ and $QM(KERNEL_C^{Slave}+MORM_C^{Slave})$ are the respective metric extracted from the whole code formed by kernel and MORM-C source code for the master and slave kernels, respectively; ii) $QM(KERNEL_C^{Master})$ and $QM(KERNEL_C^{Slave})$ are the respective metric extracted from the kernel source code excluding MORM-C source code through the preprocessing macro for the master and slave kernels, respectively.

$$QM(MORM_C^{Master}) = QM(KERNEL_C^{Master} + MORM_C^{Master}) - QM(KERNEL_C^{Master}) \tag{6.15}$$

$$QM(MORM_C^{Slave}) = QM(KERNEL_C^{Slave} + MORM_C^{Slave}) - QM(KERNEL_C^{Slave}) \tag{6.16}$$

Since MORM works with sensor, making-decision and actuator components located on both master and slave kernels, we have consolidated the quality metrics of both kernels, as Equation 6.17.

$$QM(MORM_C^{Total}) = QM(MORM_C^{Master}) + QM(MORM_C^{Slave}) \quad (6.17)$$

The metrics for the MORM-MQSoC's source code are extracted directly from the files of its classes. Equation 6.18 defines the computation of a given quality metric (QM) for the MORM-MQSoC's source code, where the value extracted from the MORM-MQSoC classes used in the master kernel ($QM(KERNEL_{MQSoC}^{Master})$) is added to those extracted for the slave kernel ($QM(KERNEL_{MQSoC}^{Slave})$) in order to calculate the total value ($QM(MORM_{MQSoC}^{Total})$).

$$QM(MORM_{MQSoC}^{Total}) = QM(MORM_{MQSoC}^{Master}) + QM(MORM_{MQSoC}^{Slave}) \quad (6.18)$$

Table 6.8 shows the results measured for each software quality metric, comparing the source code MORM-C and MORM-MQSoC. The last line shows the percentage reduction achieved by MORM-MQSoC. From the results achieved by MORM-MQSoC, we can state that the Hypothesis 6.3.2 is true, considering the evaluated metrics. That is, the use of middleware-based design approach improves the quality of the self-adaptive system software, with gains from 33% to 47.8%, depending on the metric.

Table 6.8: Software Quality Results.

Implementation	Quality Metric (QM)						
	CC	**IC**	**FC**	**LOC**	**eLOC**	**1LOC**	**Parameters**
$MORM_C^{Total}$	3.45	3.22	6.67	16.31	13.67	10.82	1.94
$MORM_{MQSoC}^{Total}$	2.03	2.16	4.19	9.35	7.18	5.64	1.06
Reduction	41.1%	33.0%	37.2%	42.7%	47.5%	47.8%	45.3%

The performed experiments allow us to assert that the MORM-MQSoC software has better quality than the MORM-C software, presenting the same functionality with an additional cost of 5.9% of the energy spent and 4.5% of the execution time in a typical workload.

To summarize, we show in a normalized way in Figure 6.24 the set of quality metrics along with the performance/energy results presented in Section 6.3.2. While values greater than 1 represent a MORM-MQSoC overhead against MORM-C, values less than 1 represent a gain. For example, 1.05 for Energy represents a MORM-MQSoC overhead against MORM-C of 5%, while 0.59 for CC represents a MORM-MQSoC gain against MORM-C of 59%.

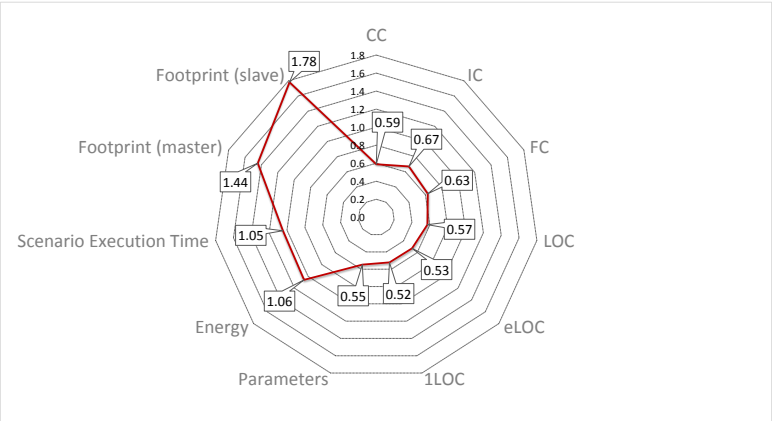


Figure 6.24: Result of the evaluation considering performance, energy and software quality metrics for the MORM-MQSoC self-adaptive system compared to MORM-C.

7. CONCLUSION

This Thesis argued the need for more systematic and standardized software development approaches in the domain of MPSoC platforms, especially for the development of self-adaptive systems in these platforms. We argued that current approaches for software development are mostly hard-coded to the underlying system components such as kernel software or ad-hoc communication protocols. This characteristic makes the maintenance and extension of the developed software a difficult task.

The fundamental problems addressed in the Thesis was how to aggregate quality properties to the self-adaptive system software, related to maintainability and software reuse, without generating an excessive performance and power consumption overhead along with memory usage required by the generated software. In the same way, we investigated current programming models regarding the coupling between the elements that make up the system. Among them, we highlighted the publish-subscribe model that enables complete decoupling in the communication between the elements of the system and the underlying hardware/software architecture.

Considering the fundamental problems addressed in this Thesis, the introduction of this Thesis stated the following hypothesis:

"Faced with current programming models and the need for self-adaptation of MPSoC systems that meet application requirements while complying system constraints, the use of a publish-subscribe model along with a middleware-based software development approach can improve software quality of self-adaptive systems while minimizing the middleware's impact on system performance, memory usage and energy spend."

To test the Thesis hypothesis, we followed an evolutionary middleware design approach starting with aspects of middleware communication on a NoC-based platform, detailed in Chapter 4, succeeded by the deployment of a middleware extension that supports the development of self-adaptive systems, detailed in Chapter 5. Regarding middleware communication aspects, we presented in Section 4.2 both the phases of the proposed publish-subscribe protocol and the communication API that are the foundation of the programming model incorporated to the middleware design. As an evaluation, we compared the results achieved with an MPI-based programming model on an MPSoC platform simulated in an instruction accurate abstraction level. The proposed publish-subscribe model improved the performance of the evaluated application by up to 29.9%, while it presented an overhead of 43.7% in the footprint size.

From this initial design of the middleware implemented in C programming language without any object-oriented technique, we derived the publish-subscribe protocol phases and redesigned the middleware incorporating best-practices of object-oriented programming and design patterns to introduce code reuse aiming the support to the development of self-adaptive systems. Section 4.3 described the new middleware structure and API, introducing an interface for middleware extensions along with other design improvements such as protocol stack and hardware/software abstraction layer. We evaluated the new middleware structure by comparing the results with the previous middleware implementation. The middleware presented an improvement in performance of up to 19.5% in the simulated applications, while having an 8.7% footprint overhead compared to the previous middleware.

In order to establish a standard for how middleware encapsulates messages in the self-adaptive system, we identified the need to include a data serialization feature into the middleware. We detailed in Section 4.4 a benchmark evaluation that we have performed on available serialization libraries for embedded devices with small memory. The goal was to identify the library with the

best performance results and that requires the lowest footprint size. From the achieved results, we highlighted the Msgpuck library to perform the serialization into the middleware.

The middleware structure designed following OOP programming allowed us to develop a middleware extension that abstracts communication details to the designer. This middleware extension, called *Modules*, has the function of wrapping the logic of the elements of the self-adaptive system in C++ classes: Sensor, Monitor, Decisor, Actuator and Effector. Thus, the designer of the self-adaptive system does not need to know details of the middleware communication primitives. We detailed the *Modules* extension in Chapter 5, including a demonstration of how to implement an adaptive service using the *Modules* extension in Section 5.4.

In order to demonstrate that the provided middleware-based approach generates software of better quality while minimizing the middleware's impact on the system execution and memory requirements, we developed a case study comparing our proposal with a baseline self-adaptive system developed at [MM18]. The adopted self-adaptive system is a hierarchical adaptive Multi-Objective Resource Management (MORM) for MPSoCs under a power cap, considering dynamic application workloads. MORM can dynamically shift the management goals prioritizing energy or performance according to the workload behavior. MORM employs the Observe-Decide-Act (ODA) paradigm comprising elements of observation, decision-making and actuation.

We redesigned the whole baseline implementation of MORM using our approach, as detailed in Section 6.2. The logic of the elements that make up the self-adaptive system - sensors, decision makers and actuators - has been wrapped in the *Modules* extension classes with the communication between the elements performed by the proposed publish-subscribe protocol. Our approach stands out from the baseline development approach regarding software quality metrics. The results showed that the MORM software designed following our approach presented improvements of 41.1% in code complexity and 33% in interface complexity, in addition to similar gains in other software quality metrics, as detailed in Section 6.3.4. As expected due to the addition of a new layer in the software stack, our approach presented an overhead at execution time of 4.5% and at energy spent of 5.9%. The software generated in our approach showed an overhead of 52.7% and 69.7% respectively for the master and slave kernel software. The overall evaluation was detailed in Section 6.3.2.

The proposed middleware allowed us to assert the hypothesis stated in this Thesis, showing that the proposed publish-subscribe programming model, along with the middleware-based approach improved the quality of the software of the evaluated MORM self-adaptive system. Regarding the middleware's impact on the system, the achieved results showed a reduced overhead concerning execution time and energy spent. Besides, the requirements for the middleware and its extensions are suitable for MPSoC platforms with a minimum of 128KB of memory.

7.1. Future Works

To guide future works that can be extended from the current state of the research presented in this Thesis, we identify the following possible improvements:

7.1.1. Kernel services on the middleware

Kernel services ported to work as applications or extensions over the middleware. So the kernel services communicate using topics and does not need to know where the services are running.

The operating system becomes cleaner and the development of new services becomes more modular. Examples of kernel services that can be ported are the task mapping and task migration.

7.1.2. Topic Name Dictionary

We have identified that the topic-name scheme as described in Section 5.3 present a high overhead in its processing in addition to requiring a higher memory requirement by the middleware. This happens because we use the string format to define the topic type. As an alternative to this solution without losing the ease of identifying topics through strings is the definition of a dictionary of topic names. In this solution, a protocol between the clients and the broker would be responsible for translate topic names from strings to integers and thus reduce the processing overhead on middleware.

7.1.3. Security Publish-Subscribe Operations

Current middleware operations trust in the clients that are accessing the publish-subscribe system. Adding a security feature to the operations of the publish-subscribe system, particularly to the advertise, subscribe and publish operations, could to prevent malicious clients from causing any improper action on the publish-subscribe system.

7.1.4. Broker Fault-tolerance Protocol

Expand the fault tolerance protocol that we have proposed at [DHA18]. This protocol aims to assure the availability of the publish-subscribe system in case of failures in the broker. We didn't include this work in this Thesis because it is in an initial stage.

7.1.5. High-level modeling for decision-making logic

In the current approach, the decision making logic can be modeled using an algorithm. A high-level modeling approach could abstract the system adaptation rules by relating system states provided by the monitors to the respective actions in the actuating elements of the self-adaptive system, assigning priorities to these rules. In addition, the priorities could be adapted according to changes in the system requirements at runtime. Another related open research is the high-level modeling that handles making of decisions in multi-objective systems, where the objectives could be conflicting.

7.1.6. Issues regarding distributed decision-making

The self-adaptive system model and middleware provided in this Thesis does not include mechanisms to address issues concerning distributed decision-making. Instead, the designer must deal with scenarios where there is more than one decision maker for the same set of configurable

hardware/software elements, which could generate undesired behavior. An extension to the model and middleware could be provided for this purpose, aiming to abstract this design complexity through mechanisms already employed in multi-agent systems [Mor05].

7.1.7. Support to real-time applications and services

This Thesis does not address applications or services with real-time constraints. The support to this type of applications implies in improving the middleware architecture and models with QoS mechanisms, such as: publish-subscribe protocol with message delivery reliability; message prioritization of given topics in queues present in middleware or even in routing protocols; and customization of QoS options through API primitives parameters.

REFERENCES

[AJMH13] Aguiar, A.; Johann, S.; Magalhaes, F.; Hessel, F. "Customizable rtos to support communication infrastructures and to improve design space exploration in mpsocs". In: RSP, 2013, pp. 130–135.

[AMM+17] Amorim, T.; Martin, H.; Ma, Z.; Schmittner, C.; Schneider, D.; Macher, G.; Winkler, B.; Krammer, M.; Kreiner, C. "Systematic pattern approach for safety and security co-engineering in the automotive domain". In: SAFECOMP, 2017, pp. 329–342.

[AMR+16] Abich, G.; Mandelli, M. G.; Rosa, F. R.; Moraes, F.; Ost, L.; Reis, R. "Extending freertos to support dynamic and distributed mapping in multiprocessor systems". In: ICECS, 2016, pp. 712–715.

[AS15] Abuseta, Y.; Swesi, K. "Design patterns for self adaptive systems engineering", *Journal of Software Engineering and Applications*, vol. 6–4, 2015, pp. 11–28.

[ASB+09] Almeida, G. M.; Sassatelli, G.; Benoit, P.; Saint-Jean, N.; Varyani, S.; Torres, L.; Robert, M. "An adaptive message passing mpsoc framework", *Journal of Reconfigurable Computing*, vol. 2009–1, 2009, pp. 1–20.

[BBS15] Berkane, M. L.; Boufaida, M.; Seinturier, L. "A modular approach dedicated to self-adaptive system", *Lecture Notes on Software Engineering*, vol. 3–3, 2015, pp. 183–188.

[BCR14] Bellavista, P.; Corradi, A.; Reale, A. "Quality of service in wide scale publish—subscribe systems", *IEEE Communications Surveys & Tutorials*, vol. 16–3, 2014, pp. 1591–1616.

[Bel58] Bellman, R. "On a routing problem", *Quarterly of applied mathematics*, vol. 16–1, 1958, pp. 87–90.

[BJR11] Becker, J.; Johann, M. D. O.; Reis, R. "VLSI-SoC: Technologies for Systems Integration: 17th IFIP WG 10.5/IEEE International Conference on Very Large Scale Integration". Springer, 2011, 202p.

[BM06] Bjerregaard, T.; Mahadevan, S. "A survey of research and practices of network-on-chip", *ACM Computing Surveys*, vol. 38–1, 2006, pp. 1–51.

[BS10] Boonma, P.; Suzuki, J. "TinyDDS: An Interoperable and Configurable Publish/Subscribe Middleware for Wireless Sensor Networks". IGI Global, 2010, pp. 206 – 231.

[But97] Butenhof, D. R. "Programming with POSIX threads". Addison-Wesley Professional, 1997, 400p.

[CCJ+14] Cassano, L.; Cozzi, D.; Jungewelter, D.; Korf, S.; Hagemeyer, J.; Porrmann, M.; Bernardeschi, C. "An inter-processor communication interface for data-flow centric heterogeneous embedded multiprocessor systems". In: DTIS, 2014, pp. 1–6.

[CCK07] Choi, Y.; Chang, N.; Kim, T. "Dc–dc converter-aware power management for low-power embedded systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26–8, 2007, pp. 1367–1381.

[CCM14] Carara, E. A.; Calazans, N. L. V.; Moraes, F. G. "Differentiated communication services for noc-based mpsocs", *IEEE Transactions on Computers*, vol. 63–3, 2014, pp. 595–608.

[CCS+08] Ceng, J.; Castrillon, J.; Sheng, W.; Scharwachter, H.; Leupers, R.; Ascheid, G.; Meyr, H.; Isshiki, T.; Kunieda, H. "Maps: An integrated framework for mpsoc application parallelization". In: DAC, 2008, pp. 754–759.

[CdOCM09] Carara, E. A.; de Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. "Hemps - a framework for noc-based mpsoc generation". In: ISCAS, 2009, pp. 1345–1348.

[CDRT13] Chen, J.; Díaz, M.; Rubio, B.; Troya, J. M. "Ps-quasar: A publish/subscribe qos aware middleware for wireless sensor and actor networks", *Journal of Systems and Software*, vol. 86–6, 2013, pp. 1650–1662.

[Cha01] Chandra, R. "Parallel programming in OpenMP". Morgan Kaufmann, 2001, 231p.

[CM12] Chaqfeh, M. A.; Mohamed, N. "Challenges in middleware solutions for the internet of things". In: CTS, 2012, pp. 21–26.

[CNG10] Carter, N. P.; Naeimi, H.; Gardner, D. S. "Design techniques for cross-layer resilience". In: DATE, 2010, pp. 1023–1028.

[CPC10] Che, W.; Panda, A.; Chatha, K. S. "Compilation of stream programs for multicore processors that incorporate scratchpad memories". In: DATE, 2010, pp. 1118–1123.

[CPS14] Chitic, S.-G.; Ponge, J.; Simonin, O. "Are middlewares ready for multi-robots systems?" In: SIMPAR, 2014, pp. 279–290.

[DCT+13] Derin, O.; Cannella, E.; Tuveri, G.; Meloni, P.; Stefanov, T.; Fiorin, L.; Raffo, L.; Sami, M. "A system-level approach to adaptivity and fault-tolerance in noc-based mpsocs: The madness project", *Microprocessors and Microsystems*, vol. 37–6, 2013, pp. 515–529.

[DDF+06] Dobson, S.; Denazis, S.; Fernández, A.; Gaïti, D.; Gelenbe, E.; Massacci, F.; Nixon, P.; Saffre, F.; Schmidt, N.; Zambonelli, F. "A survey of autonomic communications", *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1–2, 2006, pp. 223–259.

[DHA18] Domingues, A.; Hamerski, J. C.; Amory, A. "Broker fault recovery for a multiprocessor system-on-chip middleware". In: SBCCI, 2018, pp. 1–6.

[DJS15] Dutt, N.; Jantsch, A.; Sarma, S. "Self-aware cyber-physical systems-on-chip". In: ICCAD, 2015, pp. 46–50.

[DJS16] Dutt, N.; Jantsch, A.; Sarma, S. "Toward smart embedded systems: A self-aware system-on-chip (soc) perspective", *ACM Transactions on Embedded Computing Systems*, vol. 15–2, 2016, pp. 22:1–22:27.

[Dou10] Douglass, B. P. "Design patterns for embedded systems in C: an embedded software engineering toolkit". Elsevier, 2010, 472p.

[DRA10] Deshpande, S.; Ravale, P.; Apte, S. "Cache coherence in centralized shared memory and distributed shared memory architectures", *Journal on Computer Science and Engineering*, vol. 2010–1, 2010, pp. 39–44.

[Ecl16] Eclipse. "An open source mqtt v3.1/v3.1.1 broker". Source: https://mosquitto.org/, Last access on: 2016-12-13.

[EFGK03] Eugster, P. T.; Felber, P. A.; Guerraoui, R.; Kermarrec, A.-M. "The many faces of publish/subscribe", *ACM Computing Surveys*, vol. 35–2, 2003, pp. 114–131.

[EHL+09] Eloranta, V.-P.; Hartikainen, V.-M.; Leppänen, M.; Reijonen, V.; Haikala, I.; Koskimies, K.; Mikkonen, T. "Patterns for distributed embedded control system software architecture", Technical Report, Tampere University of Technology, 2009, 17p.

[ES12] Elkady, A.; Sobh, T. "Robotics middleware: A comprehensive literature survey and attribute-based bibliography", *Journal of Robotics*, vol. 2012–1, 2012, pp. 1–15.

[FDLP11] Fattah, M.; Daneshtalab, M.; Liljeberg, P.; Plosila, J. "Exploration of mpsoc monitoring and management systems". In: ReCoSoC, 2011, pp. 1–3.

[Fer15] Fersi, G. "Middleware for internet of things: A study". In: DCOSS, 2015, pp. 230–235.

[FF62] Ford, L.; Fulkerson, D. "Flows in networks", *Princeton University Press*, vol. 412–1, 1962, pp. 527–532.

[GB10] Göhringer, D.; Becker, J. "High performance reconfigurable multi-processor-based computing on fpgas". In: IPDPSW, 2010, pp. 1–4.

[GBO+16] Garibotti, R.; Butko, A.; Ost, L.; Gamatie, A.; Sassatelli, G.; Adeniyi-Jones, C. "Efficient embedded software migration towards clusterized distributed-memory architectures", *IEEE Transactions on Computers*, vol. 65–8, 2016, pp. 2645–2651.

[GHHDB10] Göhringer, D.; Hübner, M.; Hugot-Derville, L.; Becker, J. "Message passing interface support for the runtime adaptive multi-processor system-on-chip rampsoc". In: SAMOS, 2010, pp. 357–364.

[GLH+12] Gillen, M.; Loyall, J.; Haigh, K. Z.; Walsh, R.; Partridge, C.; Lauer, G.; Strayer, T. "Information dissemination in disadvantaged wireless communications using a data dissemination service and content data network". In: SPIE, 2012, pp. 1–12.

[GOB+13] Garibotti, R.; Ost, L.; Busseuil, R.; Kourouma, M.; Adeniyi-Jones, C.; Sassatelli, G.; Robert, M. "Simultaneous multithreading support in embedded distributed memory mpsocs". In: DAC, 2013, pp. 1–7.

[Gun16] Guntheroth, K. "Optimized C++: Proven Techniques for Heightened Performance". O'Reilly Media Inc., 2016, 388p.

[GWHB11] Göhringer, D.; Werner, S.; Hubner, M.; Becker, J. "Rampsocvm: Runtime support and hardware virtualization for a runtime adaptive mpsoc". In: FPL, 2011, pp. 181–184.

[HAR+17] Hamerski, J. C.; Abich, G.; Reis, R.; Ost, L.; Amory, A. "Publish-subscribe programming for a NoC-based multiprocessor system-on-chip". In: ISCAS, 2017, pp. 1–4.

[HAR+18] Hamerski, J. C.; Abich, G.; Reis, R.; Ost, L.; Amory, A. "A design patterns-based middleware for multiprocessor systems-on-chip". In: SBCCI, 2018, pp. 1–6.

[HDFGM18] Hamerski, J. C.; Domingues, A.; F. G. Moraes, A. A. "Evaluating serialization for a publish-subscribe based middleware for mpsocs". In: ICECS, 2018, pp. 1–4.

[HLLL08] Hsieh, K.-Y.; Liu, Y.-C.; Lai, C.-H.; Lee, J. K. "The support of software design patterns for streaming rpc on embedded multicore processors". In: SIPS, 2008, pp. 263–268.

[Hof13] Hoffman, H. "Seec: A framework for self-aware management of goals and constraints in computing systems", Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, USA, 2013, 172p.

[HTSC08] Hunkeler, U.; Truong, H. L.; Stanford-Clark, A. "Mqtt-s - a publish/subscribe protocol for wireless sensor networks". In: COMSWARE, 2008, pp. 791–798.

[HZZ+14] Heisswolf, J.; Zaib, A.; Zwinkau, A.; Kobbe, S.; Weichslgartner, A.; Teich, J.; Henkel, J.; Snelting, G.; Herkersdorf, A.; Becker, J. "Cap: Communication aware programming". In: DAC, 2014, pp. 1–6.

[ICG07] Issarny, V.; Caporuscio, M.; Georgantas, N. "A perspective on the future of middleware-based software engineering". In: FOSE, 2007, pp. 244–258.

[JBA+13] Joven, J.; Bagdia, A.; Angiolini, F.; Strid, P.; Castells-Rufas, D.; Fernandez-Alonso, E.; Carrabina, J.; Micheli, G. D. "Qos-driven reconfigurable parallel computing for noc-based clustered mpsocs", *IEEE Transactions on Industrial Informatics*, vol. 9–3, 2013, pp. 1613–1624.

[JSHP14] Javaid, H.; Shafique, M.; Henkel, J.; Parameswaran, S. "Energy-efficient adaptive pipelined mpsocs for multimedia applications", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33–5, 2014, pp. 663–676.

[JW04] Jerraya, A.; Wolf, W. "Multiprocessor systems-on-chips". Elsevier, 2004, 608p.

[KKKH16] Kim, T.; Kang, J.; Kim, S.; Ha, S. "Sophy+: Programming model and software platform for hybrid resource management of many-core accelerators", *Microprocessors and Microsystems*, vol. 43–1, 2016, pp. 47–58.

[KMKL16] Khemaissia, I.; Mosbahi, O.; Khalgui, M.; Li, Z. "Crmpsoc: New solution for feasible reconfigurable mpsoc". In: ICSOFT, 2016, pp. 175–198.

[KMZS08] Kasim, H.; March, V.; Zhang, R.; See, S. "Survey on parallel programming model". In: NPC, 2008, pp. 266–275.

[Kor18] Kormanyos, C. "Real-time C++: efficient object-oriented and template microcontroller programming". Springer, 2018, 426p.

[Lab10] Labiod, H. "Wireless ad hoc and Sensor Networks". John Wiley & Sons, 2010, 352p.

[LBP15] Lingaraj, K.; Biradar, R. V.; Patil, V. C. "A survey on middleware challenges and approaches for wireless sensor networks". In: CICN, 2015, pp. 56–60.

[LCA+11] Li, S.; Chen, K.; Ahn, J. H.; Brockman, J. B.; Jouppi, N. P. "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques". In: ICCAD, 2011, pp. 694–701.

[LF15] Lakhani, F.; Faisal, N. "Design patterns-from architecture to embedded software development", *Journal of Computer Science Issues*, vol. 12–1, 2015, pp. 146–152.

[LM14] Li, X.; Moh, S. "Middleware systems for wireless sensor networks: A comparative survey", *Contemporary Engineering Sciences*, vol. 7–13-16, 2014, pp. 649–660.

[Mae11] Maeda, K. "Comparative survey of object serialization techniques and the programming supports", *Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 5–12, 2011, pp. 1488–1493.

[Mae12] Maeda, K. "Performance evaluation of object serialization libraries in xml, json and binary formats". In: DICTAP, 2012, pp. 177–182.

[MAJJ08] Mohamed, N.; Al-Jaroodi, J.; Jawhar, I. "Middleware for robotics: A survey". In: RAM, 2008, pp. 736–742.

[MCA12] MCAPI. "Multicore communication api (mcapi), version 2015". Source: http://www.multicore-association.org/pdf/MCAPI_Reference_Card.pdf, Last access on: 2016-12-16.

[McC76] McCabe, T. J. "A complexity measure", *IEEE Transactions on Software Engineering*, vol. SE-2–4, 1976, pp. 308–320.

[MCM+04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: An infrastructure for low area overhead packet-switching networks on chip", *IEEE Transactions on Very Large Scale Integration Systems*, vol. 38–1, 2004, pp. 69–93.

[MCS+15] Mandelli, M.; Castilhos, G.; Sassatelli, G.; Ost, L.; Moraes, F. G. "A distributed energy-aware task mapping to achieve thermal balancing and improve reliability of many-core systems". In: SBCCI, 2015, pp. 1–7.

[MdSR+19] Martins, A. L. D. M.; da Silva, A. H. L.; Rahmani, A. M.; Dutt, N.; Moraes, F. G. "Hierarchical adaptive multi-objective resource management for many-core systems", *Journal of Systems Architecture*, vol. 2019–1, 2019, pp. 1–16.

[MFRC15] Munk, P.; Freier, M.; Richling, J.; Chen, J. J. "Dynamic guaranteed service communication on best-effort networks-on-chip". In: PDP, 2015, pp. 353–360.

[Mia15] Miasnikov, A. "C++ for embedded systems". Kindle Edition, 2015, 271p.

[MKC11] Motakis, A.; Kornaros, G.; Coppola, M. "Dynamic resource management in modern multicore socs by exposing noc services". In: ReCoSoC, 2011, pp. 1–7.

[MLIB08] Mahr, P.; Lörchner, C.; Ishebabi, H.; Bobda, C. "Soc-mpi: A flexible message passing library for multiprocessor systems-on-chips". In: ReConFig, 2008, pp. 187–192.

[MM18] Martins, A. L. D. M.; Moraes, F. G. "Multi-objective resource management for many-core systems", Ph.D. Thesis, PUCRS, Porto Alegre, Brasil, 2018, 149p.

[Moo59] Moore, E. F. "The shortest path through a maze". In: ITST, 1959, pp. 285–292.

[Mor05] Moreau, L. "Stability of multiagent systems with time-dependent communication links", *IEEE Transactions on Automatic Control*, vol. 50–2, 2005, pp. 169–182.

[MOS09] Minhass, W. H.; Öberg, J.; Sander, I. "Design and implementation of a plesiochronous multi-core 4x4 network-on-chip fpga platform with mpi hal support". In: FPGAworld, 2009, pp. 52–57.

[MPI15] MPI. "Mpi: A message-passing interface standard, v3.1". Source: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, Last access on: 2016-12-16.

[MQT99] MQTT. "Mq telemetry transport (mqtt)". Source: http://mqtt.org, Last access on: 2016-12-16.

[MSC+14] Martins, A. L.; Silva, D. R.; Castilhos, G. M.; Monteiro, T. M.; Moraes, F. G. "A method for noc-based mpsoc energy consumption estimation". In: ICECS, 2014, pp. 427–430.

[MSHH11] Matilainen, L.; Salminen, E.; Hämäläinen, T. D.; Hännikäinen, M. "Multicore communications api (mcapi) implementation on an fpga multiprocessor". In: SAMOS, 2011, pp. 286–293.

[MSK15] Magyar, G.; Sincak, P.; Krizsán, Z. "Comparison study of robotic middleware for robotic applications", *Advances in Intelligent Systems and Computing*, vol. 316–1, 2015, pp. 121–128.

[MSq18] MSquared. "Resource standard metrics". Source: http://msquaredtechnologies.com/, Last access on: 2018-10-03.

[NVC10] Nollet, V.; Verkest, D.; Corporaal, H. "A safari through the mpsoc run-time management jungle", *Journal of Signal Processing Systems*, vol. 60–2, 2010, pp. 251–268.

[OMG05] OMG. "Realtime corba specification, v1.2". Source: http://www.omg.org/spec/RT/1.2/, Last access on: 2016-12-16.

[OMG11] OMG. "Corba core specification, v3.2". Source: http://www.omg.org/spec/CORBA/3.2/, Last access on: 2016-12-16.

[Ope14] OpenCores. "Plasma - most mips i(tm) opcodes". Source: https://opencores.org/projects/plasma, Last access on: 2019-01-10.

[Par03] Pardo-Castellote, G. "Omg data-distribution service: Architectural overview". In: ICDCS, 2003, pp. 200–206.

[PBYP17] Petersen, B.; Bindner, H.; You, S.; Poulsen, B. "Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things". In: COMPUTING, 2017, pp. 1339–1346.

[PG14] Pérez, H.; Gutiérrez, J. J. "A survey on standards for real-time distribution middleware", *ACM Computing Surveys*, vol. 46–4, 2014, pp. 49.

[Pie04] Pietzuch, P. R. "Hermes: A scalable event-based middleware", Ph.D. Thesis, University of Cambridge, Cambridge, England, 2004, 180p.

[PRJW10] Popovici, K.; Rousseau, F.; Jerraya, A. A.; Wolf, M. "Embedded software design and programming of multiprocessor system-on-chip". Springer, 2010, 290p.

[QCG+09] Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A. Y. "Ros: an open-source robot operating system". In: ICRA, 2009, pp. 1–6.

[RC10] Ramirez, A. J.; Cheng, B. H. "Design patterns for developing dynamically adaptive systems". In: ICSE, 2010, pp. 49–58.

[RCM] Ruaro, M.; Carara, E. A.; Moraes, F. G. "Runtime Adaptive Circuit Switching and Flow Priority in NoC-Based MPSoCs", *IEEE Transactions on VLSI Systems*, vol. 23–6, pp. 1077–1088.

[RCM14] Ruaro, M.; Carara, E. A.; Moraes, F. G. "Runtime qos support for mpsoc: A processor centric approach". In: SBCCI, 2014, pp. 1–7.

[RLC15] Rosa, T. R.; Lemaire, R.; Clermidy, F. "A co-design approach for hardware optimizations in multicore architectures using mcapi". In: INA-OCMC, 2015, pp. 17–20.

[RLMM16] Ruaro, M.; Lazzarotto, F. B.; Marcon, C. A.; Moraes, F. G. "Dmni: A specialized network interface for noc-based mpsocs". In: ISCAS, 2016, pp. 1202–1205.

[RM18] Ruaro, M.; Moraes, F. G. "Self-adaptive qos at communication and computation levels for many-core system-on-chip", Ph.D. Thesis, PUCRS, Porto Alegre, Brasil, 2018, 150p.

[RMJPC16] Razzaque, M. A.; Milojevic-Jevric, M.; Palade, A.; Clarke, S. "Middleware for internet of things: A survey", *IEEE Internet of Things Journal*, vol. 3–1, 2016, pp. 70–95.

[ROR+14] Rosa, F.; Ost, L.; Raupp, T.; Moraes, F.; Reis, R. "Fast energy evaluation of embedded applications for many-core systems". In: PATMOS, 2014, pp. 1–6.

[ROS16a] ROS. "Names - ros wiki". Source: http://wiki.ros.org/Names, Last access on: 2016-12-13.

[ROS16b] ROS. "Parameter server - ros wiki". Source: http://wiki.ros.org, Last access on: 2016-12-13.

[RR08] Ryll, M.; Ratchev, S. "Application of the data distribution service for flexible manufacturing automation", *World Academy of Science, Engineering and Technology*, vol. 41–1, 2008, pp. 178–185.

[RRPS16] Ross, J. A.; Richie, D. A.; Park, S. J.; Shires, D. R. "Parallel programming model for the epiphany many-core coprocessor using threaded mpi", *Microprocessors and Microsystems*, vol. 43–1, 2016, pp. 95–103.

[SARM16] Sheltami, T. R.; Al-Roubaiey, A. A.; Mahmoud, A. S. "A survey on developing publish/subscribe middleware over wireless sensor/actuator networks", *Wireless Networks*, vol. 22–6, 2016, pp. 2049–2070.

[SB03] Schmidt, D. C.; Buschmann, F. "Patterns, frameworks, and middleware: their synergistic relationships". In: ICSE, 2003, pp. 694–704.

[SBP+09] Saraswat, V.; Bloom, B.; Peshansky, I.; Tardieu, O.; Grove, D. "X10 language specification, v2.3", Technical Report, IBM, 2009, 17p.

[SC08] Schmidt, D.; Corsaro, A. "Addressing the challenges of tactical information management in net-centric systems with dds", *CrossTalk*, vol. 21–1, 2008, pp. 1–6.

[SD14] Sarma, S.; Dutt, N. "Minimal sparse observability of complex networks: Application to mpsoc sensor placement and run-time thermal estimation amp; tracking". In: DATE, 2014, pp. 1–6.

[SDG+15] Sarma, S.; Dutt, N.; Gupta, P.; Venkatasubramanian, N.; Nicolau, A. "Cyberphysical-system-on-chip (cpsoc): A self-aware mpsoc paradigm with cross-layer virtual sensing and actuation". In: DATE, 2015, pp. 625–628.

[SDP+14] Silva, J. R.; Delicato, F. C.; Pirmez, L.; Pires, P. F.; Portocarrero, J. M.; Rodrigues, T. C.; Batista, T. V. "Prisma: A publish-subscribe and resource-oriented middleware for wireless sensor networks". In: AICT, 2014, pp. 87–97.

[SDV+13] Sarma, S.; Dutt, N.; Venkatasubramanian, N.; Nicolau, A.; Gupta, P. "Cyberphysical-system-on-chip (cpsoc): Sensor-actuator rich self-aware computational platform", Technical Report, University of California Irvine, 2013, 26p.

[SF09] Schneider, S.; Farabaugh, B. "Is dds for you?", Technical Report, Real-Time Innovations, 2009, 5p.

[SK00] Schmidt, D.; Kuhns, F. "An overview of the real-time corba specification", *Computer*, vol. 33–6, 2000, pp. 56–63.

[SKK+14] Said, M. B.; Kacem, Y. H.; Kerboeuf, M.; Amor, N. B.; Abid, M. "Design patterns for self-adaptive rte systems specification", *Journal of Reconfigurable Computing*, vol. 2014–1, 2014, pp. 1–21.

[SKL11] Sain, M.; Kumar, P.; Lee, H. "A survey of middleware and security approaches for wireless sensor networks". In: ICCIT, 2011, pp. 64–69.

[SM12] Sumaray, A.; Makki, S. K. "A comparison of data serialization formats for optimal efficiency on a mobile platform". In: ICUIMC, 2012, pp. 48:1–48:6.

[SR14] Saghian, M.; Ravanmehr, R. "A survey on middleware approaches for distributed real-time systems", *Journal of Mobile, Embedded and Distributed Systems*, vol. 6–4, 2014, pp. 147–158.

[Tar12] Tarkoma, S. "Publish/subscribe systems: design and principles". John Wiley & Sons, 2012, 360p.

[TN12] Tong, X.; Ngai, E. C. "A ubiquitous publish/subscribe platform for wireless sensor networks with mobile mules". In: DCOSS, 2012, pp. 99–108.

[TR13] Tuveri, G.; Raffo, L. "Integrated support for adaptivity and fault-tolerance in mpsocs", Ph.D. Thesis, Universita'degli Studi di Cagliari, Cagliari, Italy, 2013, 93p.

[Whi11] White, E. "Making Embedded Systems: Design Patterns for Great Software". O'Reilly Media Inc., 2011, 330p.

APPENDIX A – List of base primitives of the HSAL

Table APPENDIX A.1: Primitives of the Hardware/Software Abstraction Layer

Primitive	Category	Description
HAL_OS_TASK_GetCurrentTask()	OS Task	Retrieves the currently running task
HAL_OS_TASK_GetCurrentApp(task)	OS Task	Retrieves the application id of a given task
HAL_OS_COMM_send(netID, msg, length)	OS Communication	Sends a message (msg, pointer to integer) with a given length (length - number of integers) to a destination PE (netID - network address) through network interface.
HAL_OS_DEBUG_getTickCount()	OS Debug	Retrieves the number of ticks that have elapsed since the system was started
HAL_OS_DEBUG_PRINT(str)	OS Debug	Prints a given string (str)
HAL_OS_DEBUG_PRINT_1ARG(str,a)	OS Debug	Prints a given string (str) and integer (a)
HAL_OS_DEBUG_PRINT_2ARG(str,a,b)	OS Debug	Prints a given string (str), and two integers (a,b)
HAL_CONFIG_getnPes()	System Config	Retrieves the number of PEs of the cluster
HAL_CONFIG_getPeType(pe_id)	System Config	Retrieves the type (master or slave) of a given PE (pe_id))
HAL_CONFIG_getNumberOfCpusX()	System Config	Retrieves the number of PEs of the whole MPSoC located in the horizontal axis
HAL_CONFIG_getNumberOfCpusY()	System Config	Retrieves the number of PEs of the whole MPSoC located in the vertical axis
HAL_CONFIG_getClusterX()	System Config	Retrieves the number of PEs of the cluster located in the horizontal axis
HAL_CONFIG_getClusterY()	System Config	Retrieves the number of PEs of the cluster located in the vertical axis
HAL_CONFIG_GetUID()	System Config	Retrieves the network address of the current PE
HAL_CONFIG_GetClusterUID()	System Config	Retrieves the cluster identification of the current PE
HAL_CONFIG_CLUSTER_NUMBER()	System Config	Retrieves the number of clusters of the MPSoC
HAL_CONFIG_MAX_LOCAL_TASKS()	System Config	Retrieves the number of maximum tasks running on the current PE
HAL_OS_SENSORS_getSample_routerInjection()	OS Sensors	Retrieves a router injection sample
HAL_OS_SENSORS_getSample_routerCongestion()	OS Sensors	Retrieves a sample of the router congestion sensor
HAL_OS_SENSORS_getSample_routerEnergyLeak()	OS Sensors	Retrieves a sample of the leakage energy sensor of the router
HAL_OS_SENSORS_getSample_routerEnergyDyn()	OS Sensors	Retrieves a sample of the dynamic energy sensor of the router
HAL_OS_SENSORS_getSample_memoryEnergyLeak()	OS Sensors	Retrieves a sample of the leakage energy sensor of the memory
HAL_OS_SENSORS_getSample_memoryEnergyDyn()	OS Sensors	Retrieves a sample of the dynamic energy sensor of the memory
HAL_OS_SENSORS_getSample_processorEnergyLeak()	OS Sensors	Retrieves a sample of the leakage energy sensor of the processor
HAL_OS_SENSORS_getSample_energy_leakage()	OS Sensors	Retrieves a sample of the leakage energy of the PE, considering all components
HAL_OS_SENSORS_getSample_energy_total()	OS Sensors	Retrieves a sample of the PE energy, considering both leakage and dynamic energy
HAL_OS_SENSORS_getSample_processorSlacktime()	OS Sensors	Retrieves a sample of the slack time sensor of the processor
HAL_OS_EFFECTORS_setDVFS(vf)	OS Effectors	Sets a vf-pair value (vf) on the Voltage/Frequency Scaling actuation method (Section 6.1.3)
HAL_OS_EFFECTORS_setClusterPowerMode(clusterID, powerMode)	OS Effectors	Sets a power mode value on the cluster operation mode actuation method (Section 6.1.3)

APPENDIX B – Diagram class of the Modules extension classes from Section 5.2

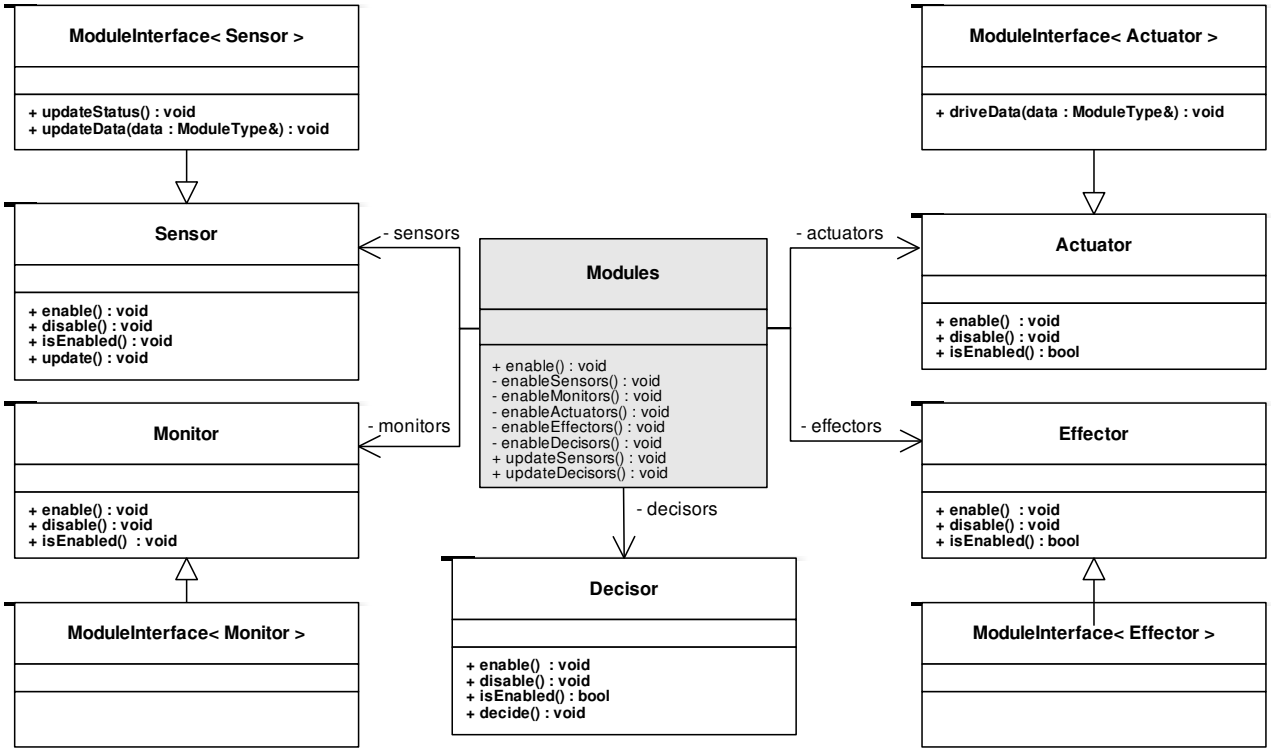


Figure APPENDIX B.1: *Modules* Class Diagram.

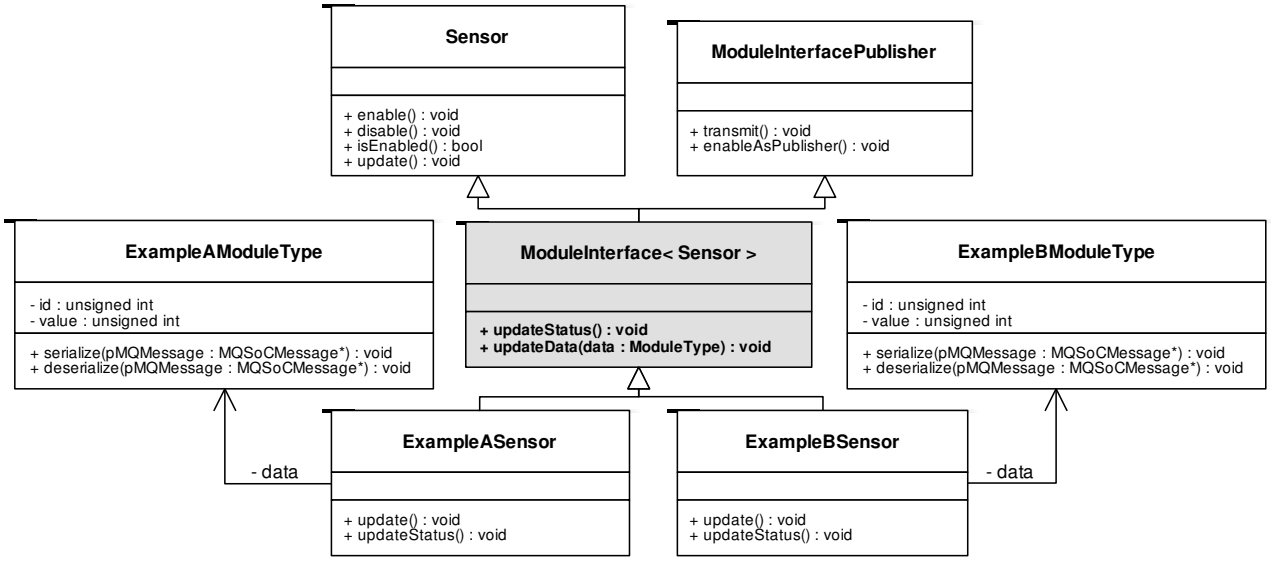


Figure APPENDIX B.2: *ModuleInterface<Sensor>* Class Diagram.

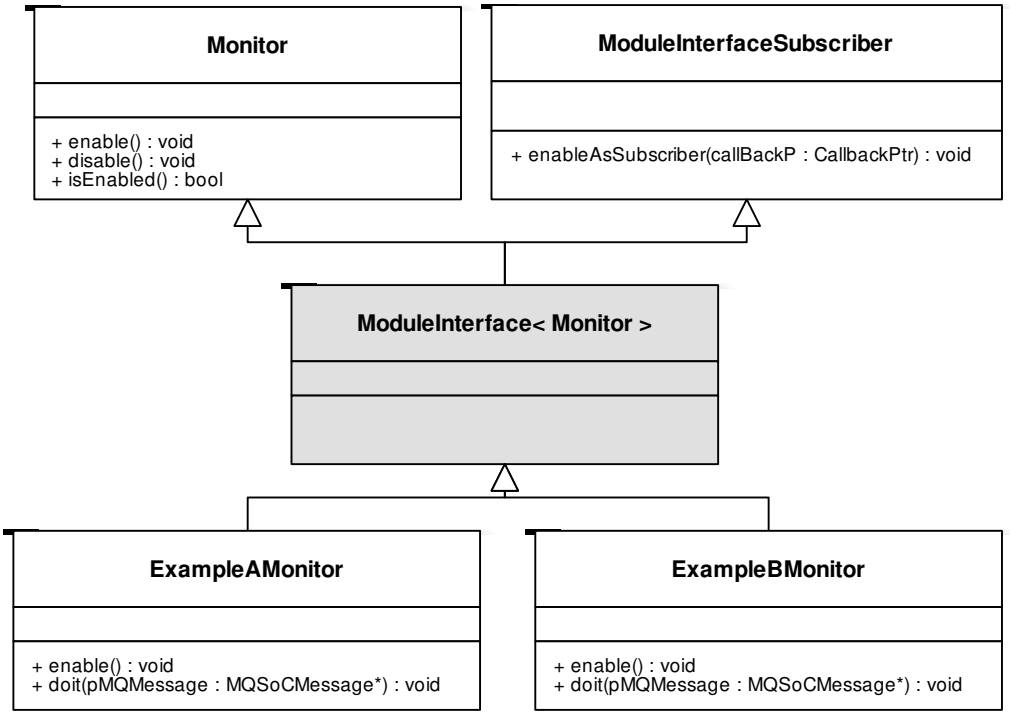


Figure APPENDIX B.3: *ModuleInterface<Monitor>* Class Diagram.

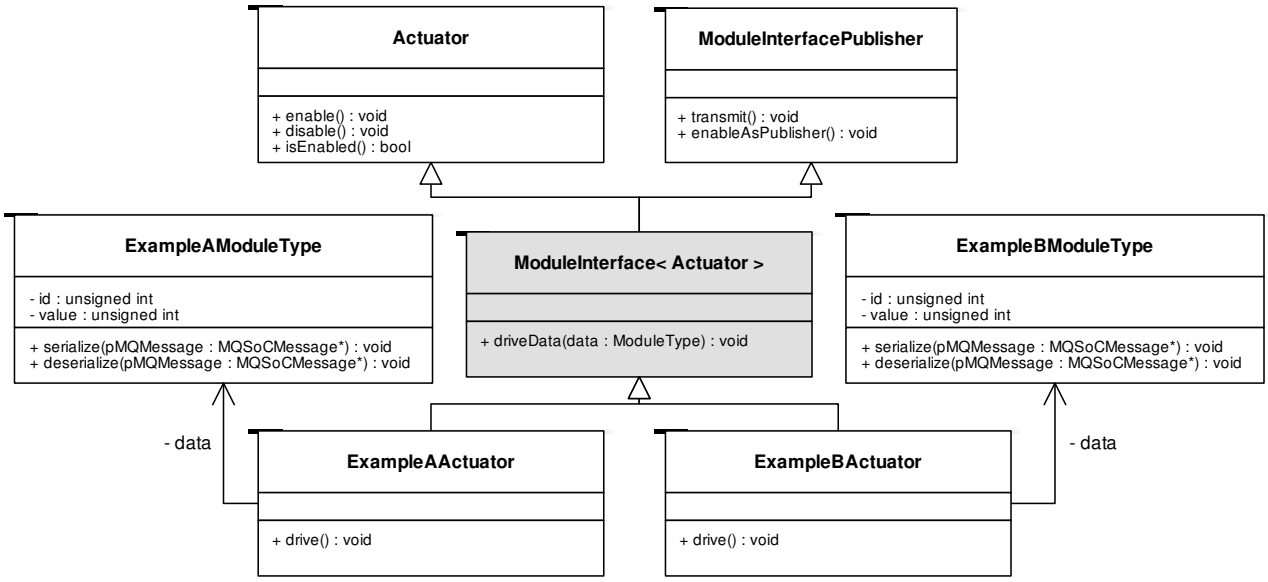


Figure APPENDIX B.4: *ModuleInterface<Actuator>* Class Diagram.

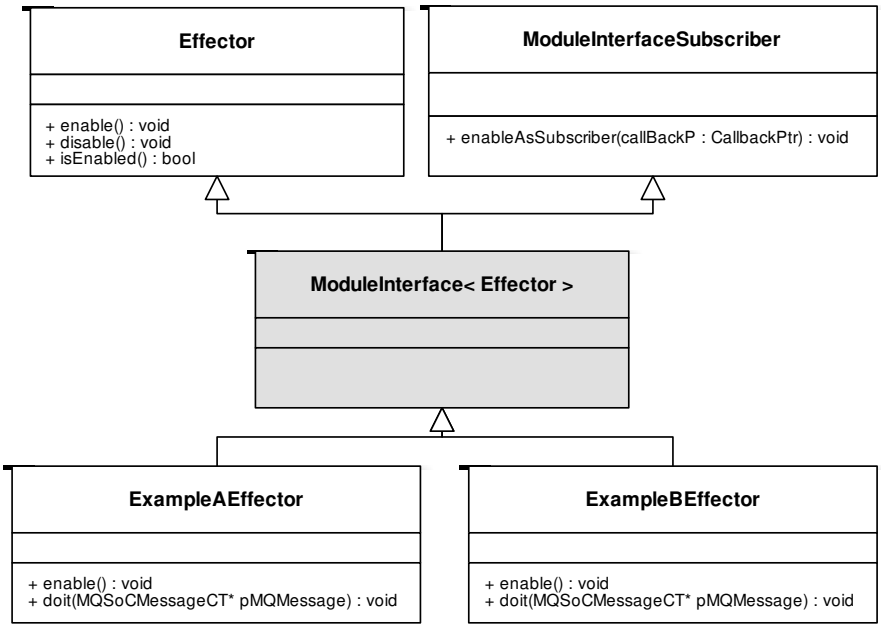


Figure APPENDIX B.5: *ModuleInterface<Effector>* Class Diagram.

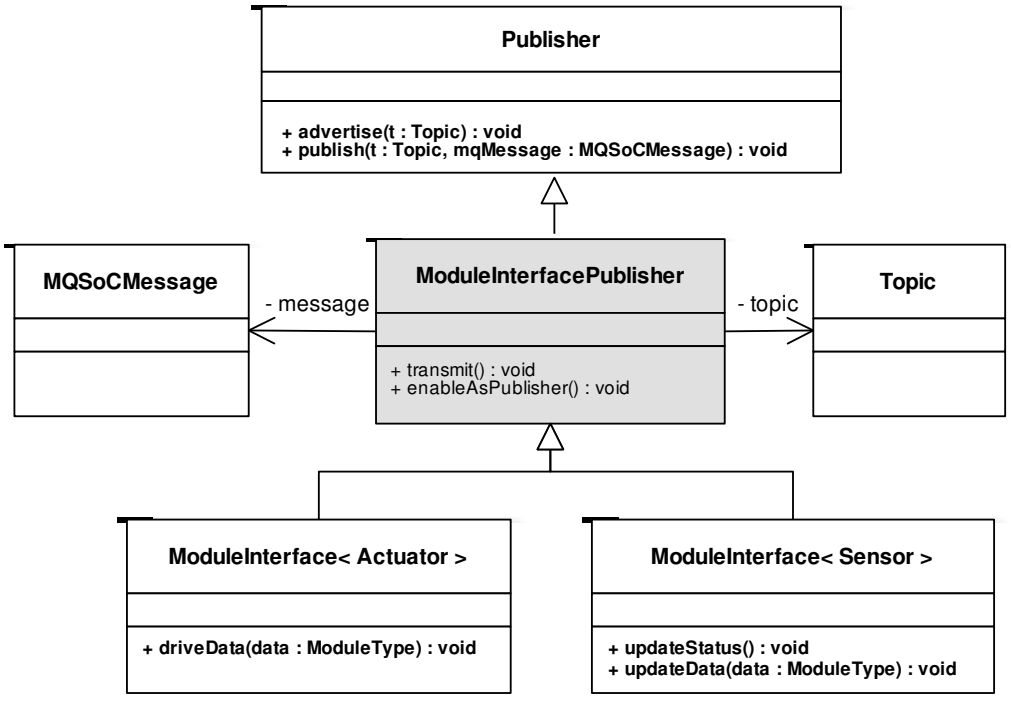


Figure APPENDIX B.6: *ModuleInterfacePublisher* Class Diagram.

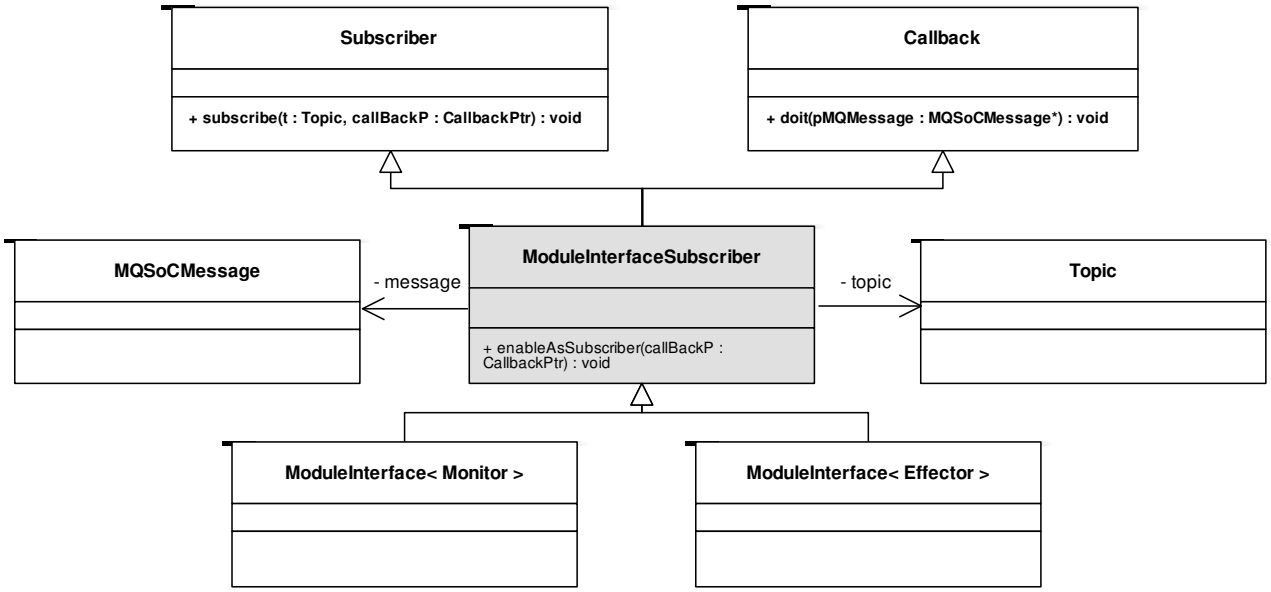


Figure APPENDIX B.7: *ModuleInterfaceSubscriber* Class Diagram.

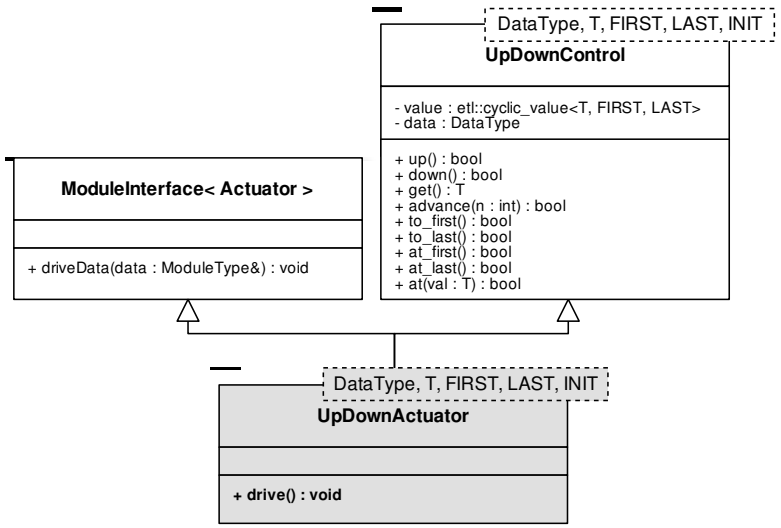


Figure APPENDIX B.8: *UpDownActuator* Class Diagram.

APPENDIX C – Header and source files of the classes detailed in Section 5.4

```
#include <Sensor.h>
#include <STType.h>
#include <ModuleInterface.h>
...
class STSensor : public ModuleInterface<Sensor>
{
  public:
    //*** Class Constructor ***//
    STSensor();
    //*** Data that will be serialized  ***//
    STType data;
    ...
    //*** Methods inherited from the parent classes (ModuleType/Sensor)   ***//
    void update();
    void updateStatus();
    ...
};
```

Figure APPENDIX C.1: Code snippet of the *STSensor* header file.

```
#include <STSensor.h>
#include <hal.h>
...
STSensor::STSensor(){
  this->enable();
  //*** Get the sensor identification using the HAL ***//
  this->data.id = HAL_OS_GetDecUID();
  //*** Topic Name and Domain definition ***//
  this->topic.name.assign("/sensor/pe/");
  this->topic.name.append(HAL_OS_itoa(HAL_OS_GetDecUID()));
  this->topic.name.append("/slacktime");
  this->topic.domain = Topic::SLAVES;
  ...
  this->enableAsPublisher();
}

void STSensor::update (){
  this->updateData(this->data);
}

void STSensor::updateStatus(){
  //*** Get actual value using the specific HAL primitive ***//
  this->data.value =  HAL_OS_SENSORS_getSample_processorSlacktime();
}
```

Figure APPENDIX C.2: Code snippet of the *STSensor* source file.

```
#include <STType.h>
#include <ModuleInterface.h>
...
class STMonitor_T {
public:
  STType sensorData[HAL_OS_SLAVE_NUMBER()];
};

class STMonitor : public ModuleInterface<Monitor>{
  public:
    //*** Local variable data that stores the received sensor data ***//
    STMonitor_T data;
    //*** Methods inherited from the base classes (Effector and Callback) ***//
    void doit(MQSoCMessage* pMQMessage);
    void enable();
    //*** Callbacks for the classes that instances the monitor ***//
    static STMonitor* getObject();
    void addCallbackMonitor(CallbackMonitorPtr cb, Decisor* base) {
      notifies[callbacksInc] = {cb, base};
      callbacksInc++; //todo: tratar quando excede MAX_NOTIFIES
    }
    private:
    //*** Class Constructor ***//
    STMonitor();
    static STMonitor* obj;
    void callNotifies(unsigned int id) {
      for(binder* begin = notifies;begin != notifies+callbacksInc;begin++) {
        begin->call(id, sendTime, receiveTime);
      }
    }
    unsigned int callbacksInc;
    binder notifies[MAX_NOTIFIES];
};
```

Figure APPENDIX C.3: Code snippet of the *STMonitor* header file.

```
#include <STMonitor.h>
#include <hal.h>
...
STMonitor* STMonitor::obj = NULL;
static size_t mem_STMonitor_local[852];
//*** Method that return a pointer to already instanced STMonitor Class object ***//
STMonitor* STMonitor::getObject() {
  if(obj==NULL)
    obj = new (mem_STMonitor_local) STMonitor;
  return obj;
}
//*** Method that is called when the monitor is enabled. ***//
void STMonitor::enable(){
  this->topic.name = "/sensor/pe/+/slave";
  this->topic.domain = Topic::SLAVES;
  this->enableAsSubscriber(this);
}
//*** Method called by middleware (SubscribersManager Class) when a new message is received on the subscribed topic.
//*** This method deserializes a message and writes the received data at variable members.
void STMonitor::doit(MQSoCMessage* pMQMessage)
{
  STType DataRet;
  DataRet.deserialize(pMQMessage);
  this->data.sensorData[DataRet.id-1] = DataRet;
  this->callNotifies(DataRet.id, sendTime, receiveTime);
}
```

Figure APPENDIX C.4: Code snippet of the *STMonitor* source file.

```
#include <ModuleType.h>
...
class STType: public ModuleType {
  public:

    //*** Class Constructor ***//
    STType();

    //*** Variables that are serialized/deserialized. ***//
    unsigned int id;
    unsigned int value;

    //*** Methods inherited from the parent classes (ModuleType) ***//
    void serialize(MQSoCMessage* pMQMessage);
    void deserialize(MQSoCMessage* pMQMessage);
};
```

Figure APPENDIX C.5: Code snippet of the *STType* header file.

```
#include <STType.h>
#include <msgpuck.h>
...
//*** Serialize the variable members to the passing message. ***//
void STType::serialize(MQSoCMessage* pMQMessage)
{
  char buf[PS_MSG_PAYLOAD_SIZE*4];
  char *w = buf;

  //*** Place bellow the code to serialize each variable class member.  ***//
  w = mp_encode_uint(w, this->id);
  w = mp_encode_uint(w, this->value);

  pMQMessage->msg_len = w-buf;
  pMQMessage->msg = buf;
}

//*** Deserialize the passing message writing the variable members. ***//
void STType::deserialize(MQSoCMessage* pMQMessage) {
  const char *r = pMQMessage->msg;

  //*** Place bellow the code to deserialize the message to each variable member of the class. ***//
  this->id = mp_decode_uint(&r);
  this->value = mp_decode_uint(&r);
}
```

Figure APPENDIX C.6: Code snippet of the *STType* source file.

```
#include "Actuator.h"
#include <DvfsType.h>
#include <Topic.h>
#include <DVFSValue.h>
#include <UpDownActuator.h>
...
class DvfsActuator : public UpDownActuator <DvfsType, DVFSValue, DVFSValue::VF_PAIR_3, DVFSValue::VF_PAIR_1,
DVFSValue::VF_PAIR_1>
{
  public:
   //*** Class Constructor ***//
    DvfsActuator();
    //*** Methods inherited from the parent classes (Actuator) ***//
    void enable(unsigned int id);
  private:
    bool enabled;
};
```

Figure APPENDIX C.7: Code snippet of the *DvfsActuator* header file.

```
#include "include/DvfsActuator.h"
...
//*** Method that is called when the Actuator is enabled. ***//
void DvfsActuator::enable(unsigned int id){
  this->enabled = true;
  this->data.id = id;

  this->topic.name.assign("/actuator/pe/");
  this->topic.name.append(HAL_OS_itoa(id));
  this->topic.name.append("/dvfs");
  this->topic.domain = Topic::SLAVES;

  this->enableAsPublisher();
}
```

Figure APPENDIX C.8: Code snippet of the *DvfsActuator* source file.

```
#include <DvfsType.h>
#include <ModuleInterface.h>
...
class DvfsEffector : public ModuleInterface<Effector>
{
  public:
    //*** Class Constructor ***//
    DvfsEffector();
    //*** Methods inherited from the parent classes (Effector and Callback) ***//
    void enable();
    void doit(MQSoCMessage pMQMessage);
    //*** Local variable data that stores the received actuation data ***//
    DvfsType data;
};
```

Figure APPENDIX C.9: Code snippet of the *DvfsEffector* header file.

```
#include <hal.h>
#include <DvfsEffector.h>
...
void DvfsEffector::enable(){
  this->data.id = HAL_OS_GetDecUID();
  this->topic.name.assign("/actuator/pe/");
  this->topic.name.append(HAL_OS_itoa(HAL_OS_GetDecUID()));
  this->topic.name.append("/dvfs");
  this->topic.domain = Topic::SLAVES;
  ...
  this->enableAsSubscriber(this);
}
void DvfsEffector::doit(MQSoCMessage* pMQMessage)
{
  this->data.deserialize(pMQMessage);
  HAL_OS_EFFECTORS_setDVFS(this->data.value);
}
```

Figure APPENDIX C.10: Code snippet of the *DvfsEffector* source file.

```
#include <ModuleType.h>
...
class DvfsType: public ModuleType {
  public:
    //*** Class Constructor ***//
    STType();
    //*** Variables that are serialized/deserialized. ***//
    unsigned int id;
    unsigned int value;
    //*** Methods inherited from the parent classes (ModuleType) ***//
    void serialize(MQSoCMessage* pMQMessage);
    void deserialize(MQSoCMessage* pMQMessage);
};
```

Figure APPENDIX C.11: Code snippet of the *DvfsType* header file.

```
#include <DvfsType.h>
#include <msgpuck.h>
...
//*** Serialize the variable members to the passing message. ***//
void DvfsType::serialize(MQSoCMessage* pMQMessage)
{
  char buf[PS_MSG_PAYLOAD_SIZE*4];
  char *w = buf;
  //*** Place bellow the code to serialize each variable class member.  ***//
  w = mp_encode_uint(w, this->id);
  w = mp_encode_uint(w, this->value);
  pMQMessage->msg_len = w-buf;
  pMQMessage->msg = buf;
}
//*** Deserialize the passing message writing the variable members. ***//
void DvfsType::deserialize(MQSoCMessage* pMQMessage) {
  const char *r = pMQMessage->msg;
  //*** Place bellow the code to deserialize the message to each variable member of the class. ***//
  this->id = mp_decode_uint(&r);
  this->value = mp_decode_uint(&r);
}
```

Figure APPENDIX C.12: Code snippet of the *DvfsType* source file.

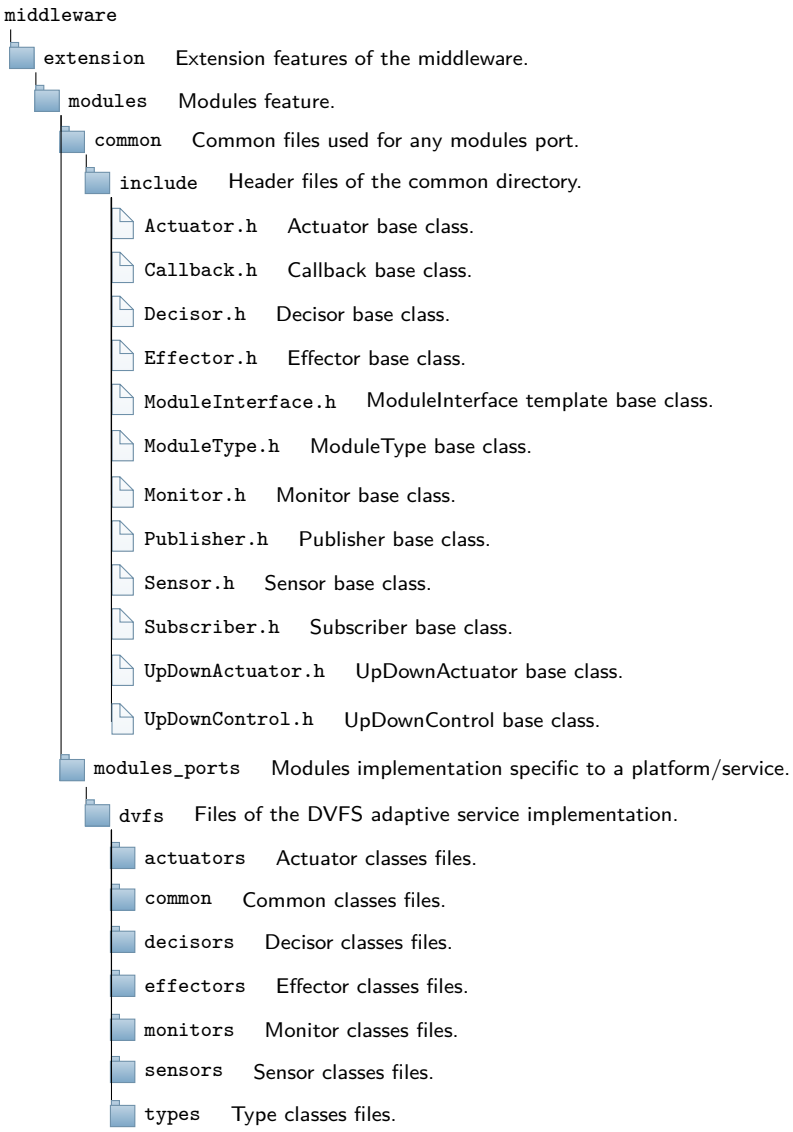
```
#include <Decisor.h>
#include <DvfsActuator.h>
#include <STMonitor.h>
...
class SimpleDecisor : public Decisor {
  public:
    //*** Methods inherited from the base class (Decisor) ***//
    void enable();
    void decide(unsigned int id);
    ...
  private:
    void notifySTMonitor(unsigned int id);
    //*** Monitor Class Instance ***//
    STMonitor* STMonitor;
    //*** Actuator Class Instance ***//
    DvfsActuator DvfsActuator[HAL_OS_SLAVE_NUMBER()];
};
```

Figure APPENDIX C.13: Code snippet of the *SimpleDecisor* header file.

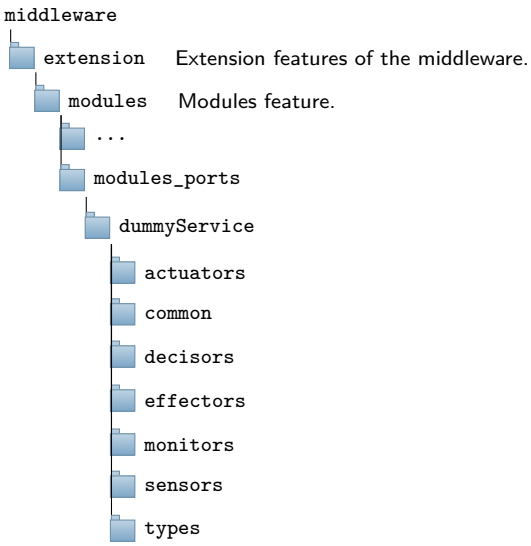
```
#include <SimpleDecisor.h>
...
//*** Method that is called when the decisor is enabled. ***//
void SimpleDecisor::enable(){
  this->STMonitor = STMonitor::getObject();
  this->STMonitor->enable();
  this->STMonitor->addCallbackMonitor((CallbackMonitorPtr) &this->notifySTMonitor,this);
  int i;
  for (i=0;i<HAL_OS_SLAVE_NUMBER();i++)
    DvfsActuator[i].enable(i+1);
}
//*** Method that is called when the the monitor variables are updated. ***//
void SimpleDecisor::notifySTMonitor(unsigned int id){
  decide(id);
}
/*** Method that is called to decide an action. ***//
void SimpleDecisor::decide(unsigned int id){
  if (STMonitor->data.sensorData[id-1].value < 25)
    DvfsActuator[id-1].down();
  else
    if (STMonitor->data.sensorData[id-1].value > 75)
      DvfsActuator[id-1].up();
}
```

Figure APPENDIX C.14: Code snippet of the *SimpleDecisor* source file.

APPENDIX D – Directory tree of Modules extension of the middleware



APPENDIX E – Base Directory tree of a Modules port



APPENDIX F – A tool to automate the creation of objects of an adaptive service

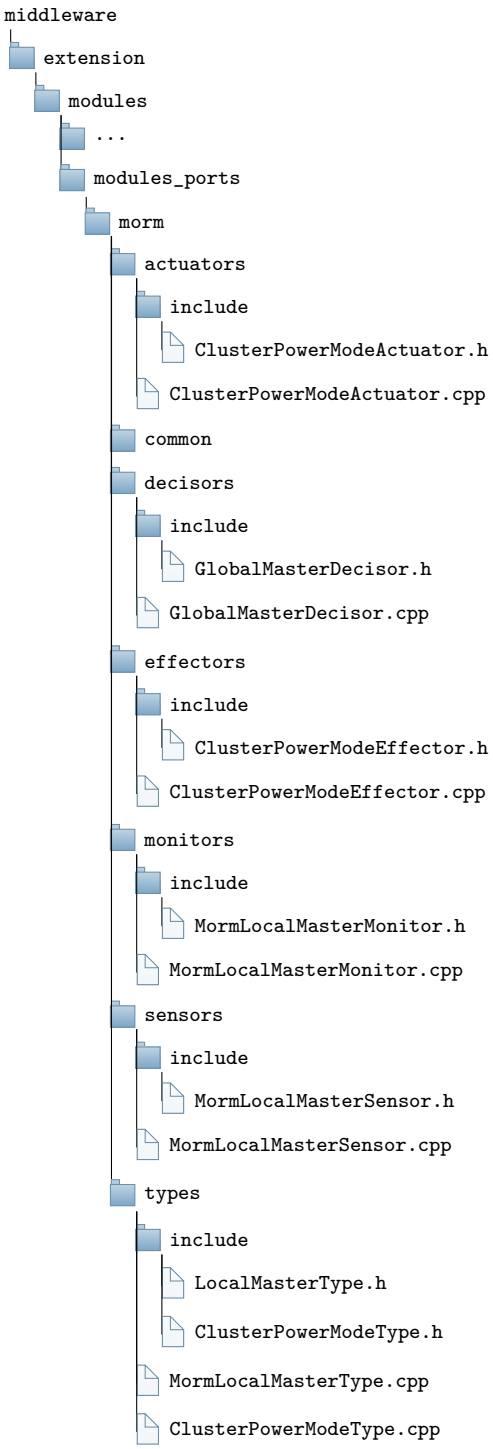
This Appendix describes a tool provided to automatically create the files, directory structures, and makefile of the objects of an adaptive service. This tool aims to facilitate the work of the user that is developing an adaptive service in the middleware.

The tool is a script written in the Python language and can be run in a Linux shell environment with support to Python. The following steps were tested in an Ubuntu Linux environment, version 14.04.1, with Python in the version 2.7.6.

The first step is to create the directory structure of the module port. For this, the user must type in the shell `# python modules.py ports new <port_name>`, where port_name is the name of the adaptive service that the user wants to create. For example, the following command `# python modules.py ports new dummyService` generates the directory structure showed in APPENDIX E. The tool provides options to create the classes:

- Sensor:
  ```
  $ python modules.py ports <port\_name> sensor <sensorName> <topicName> <topicDomain> <typeName>
  ```
- Monitor:
  ```
  $ python modules.py ports <port\_name> monitor <monitorName> <topicName> <topicDomain> <typeName>
  ```
- Decisor:
  ```
  $ python modules.py ports <port_name> decisor <decisorName> <monitorName> <actuatorName>
  ```
- Actuator:
  ```
  $ python modules.py ports <port_name> actuator <actuatorName> <topicName> <topicDomain> <actuatorType>
  <typeName> <actuator_enum_class> <actuator_enum_first_value> <actuator_enum_last_value>
  <actuator_enum_initial_value>
  ```
- Effector:
  ```
  $ python modules.py ports <port_name> effector <effectorName> <topicName> <topicDomain> <typeName>
  ```
- Type:
  ```
  $ python modules.py ports <port_name> type <typeName>
  ```
- 3-part Sensor/Monitor/Type:
  ```
  $ python modules.py ports <port_name> 3-part-sensor/monitor <name> <topicName> <topicDomain>
  ```
- 3-part Actuator/Effector/Type:
  ```
  $ python modules.py ports <port_name> 3-part-actuator/effector <name> <topicName> <topicDomain>
  <actuatorType> <actuator_enum_class> <actuator_enum_first_value>
  <actuator_enum_last_value>
  ```
- Modules:
  ```
  $ python modules.py ports <port_name> Modules <sensorName> <effectorName> <decisorName>
  ```

APPENDIX G – Directory tree of the MORM-MQSoC adaptive service



Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br