D1.2 "Reference architecture"

WP1.

Lead Beneficiary DEU

Delivery date 2021/06/30

Dissemination level: Public

Version V1.0

Deusto
Universidad de Deusto
University of Deusto

The ACROBA project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017284.

Approval Status

	Name and Surname	**Role in the project**	**Partner**
Author(s)	Alberto Tellaeche	WP1 leader	DEUSTO
Reviewed by	Iñaki Vázquez Aly Magassouba	WP1 leader WP2 leader	DEUSTO SIGMA
Approved by	Norman U. Baier	Project Coordinator	BFH

History of Changes

Version	**Date**	**Description of Changes**	**By**
0.1	2021.06.15	First version for review	DEUSTO
0.2	2021.06.21	Minor changes	SIGMA

DISCLAIMER

Index

List of Figures

1. EXECUTIVE SUMMARY

This deliverable reports the study conducted for the design of ACROBA system architecture.

Analyzing the deliverables (e.g., D4.1: "Medical Device Pilot Line Specifications", D4.2: "Plastic Pilot Line Specifications", D5.1: "Electronic Components Pilot Line Specifications" and D5.2: "Electric Motor Pilot Line Specifications"), we decide to apply Domain-Driven Design methodology as a general approach to develop the system architecture.

After a brief introduction in Section 2, basic principles of Domain-Driven Design are reported in Section 3. The application of such guides to ACROBA domain is explained in Section 4. The system architecture is then presented in Section 5 as outcome of this study. Section 6 explains the relation of this document with the annex prepared for use cases, and finally, section 7 presents the conclusions obtained from this task and future work to be done in next steps of the project.

This document will provide an open vision of the developed architecture. Low-level drivers and primitives will be defined in the deliverable D2.1, "Robot modules architecture".

2. INTRODUCTION

ACROBA seeks to deliver a robust and modern robotics architecture to meet the demands of today's factories for robotics development in an effective manner. Due to the complexities of robotic development, we propose a generic architecture in this deliverable that may be used in a variety of circumstances.

This architecture poses several difficulties. First, it must deal with a variety of hardware resources, as well as factory information, real-time communication among various actors, and robotics advanced cognitive capabilities based on AI, providing a list of modules and capabilities ready to be used in the most complex use cases in future factories.

When designing the software architecture, a solution that incorporates all these features is complex, avoiding significant coupling (e.g., the degree of reliance between software pieces). Indeed, as an inadequate architecture could result in a waste of time and effort. The concepts around which the ACROBA architecture should be constructed are modularity and flexibility: it must be able to adapt to a variety of changes that may occur over time, such as the availability of new modules, the unavailability of others, the addition of new features, and so on.

A Domain-Driven Design methodology was chosen to facilitate system development since the robotic domains covered by ACROBA are complex. Multiple decoupled sub-domains are modelled as the basis for the realization of a modular architecture, in which each component is focused on one sub-domain, after the study of use cases and system requirements. Furthermore, ACROBA can increase the reusability of software components because of its modular architecture. Components can be implemented in parallel and interact through interfaces, reducing development time by eliminating the requirement for one component to understand how the others work internally.

Domain-Driven Design subdomains are easily transformed into a microservices architecture. Microservices architecture is made up of several loosely linked components that work together to provide a more precise functionality. Therefore, a microservice can be developed independently of the others, using different methodologies and technologies. Nonetheless a microservice needs to provide interfaces to expose its services, which will be executed by ACROBA system. Internal implementations can then be modified and upgraded in real-time without affecting the execution of other microservices in a visible manner to the system.

Microservices are built in ACROBA using several processes that require the execution of various modules. Such modules will be developed in following tasks of the project, in WP2 and also in WP related to precise use cases. As a result, the final architecture must incorporate an orchestration layer to simplify the integration of different modules, to organize their execution, and even to enable effective communication.

Modules that the orchestration layer integrates must also expose public interfaces in order to achieve these features (analogously to microservices development).

Microservices architecture necessitates the use of a testing infrastructure. A testing infrastructure decreases development time by automating tests of microservices' correct execution and integration into the final system. When a microservice undergoes internal modifications, the testing infrastructure can determine whether the microservice's integration is successful or not. To that end, the ACROBA architecture will be built using a test-driven development approach to associate test functions to each expanded functionality.

Importantly, the system architecture was created with privacy and ethical issues in mind from WP3 work, with the goal of ensuring that the produced system respects human rights, especially the right to privacy and data protection.

3. INTRODUCTION TO DOMAIN-DRIVEN DESIGN APPROACH

In order to establish an architecture that covers all essential features without losing emphasis on the ACROBA system's applicability domain, the system architecture was created using the Domain-Driven Design (DDD) technique, which considered the many situations of applicability. An overview of DDD concepts is presented in the following sections.

3.1. Overview

System design must consider the domain in which the system will operate to make it beneficial to its end users. Domain-Driven Design (DDD) is a software architecture design process that involves creating the domain-based logic and models that the final system will rely on.

Domain logic and models are defined by technical and domain specialists who collaborate in an ongoing refinement process to blend their distinct viewpoints and solve domain-based problems.

DDD defines the domain's core elements to be the main entities that interact in the final system, implying that the domain and its representation model are inextricably linked. Each domain element has its own set of behaviors, which are represented in the functions that its counterpart in the final system is required to provide. In complex domains, elements can interact to provide distinct operation contexts, which can be viewed as various services from a system perspective.

The key challenge with the DDD is defining the correct domain representation: a domain model that does not accurately reflect the operational viewpoint of the system may result in a system that does not meet user needs. As a result, constant coordination between technologists and domain specialists is essential. To eliminate ambiguities or misunderstandings, and to preserve consistency within the model, everybody involved in the design definition should adopt a common established language (*ubiquitous language*) that unambiguously describes the elements of the domain model.

The problem in DDD can be approached in two ways: **strategic** patterns, which focus on analyzing and modelling the domain from the perspective of different areas and their communication, and **tactical** patterns, which aim to define domain elements that will perform functionalities of the different areas identified in the strategic design and the technological approach.

3.2. Strategic DDD

It is possible to break the model into bounded contexts (BC) when the domain consists of numerous weakly connected sub-domains. Each BC can be thought of as a separate group of services that all deal with the same data. A BC can communicate with another BC via messages or APIs; in this instance, domain logic must allow the former BC's concepts to be translated into the latter BC's concepts and vice versa. DDD analyzes many forms of BC relationships for this purpose.

Because BCs can have a variety of relationships, a context map that indicates how BCs are connected and with what kind of relationships is useful in this method.

Each domain identified along a software design process should be treated in isolation from the others. Then, each isolated domain can be structured according to a three-layered approach:

- The **application layer** describes the work that the software is supposed to do; this layer delegates the work to the underlying layers.
- **Domain layer** represents the domain model's business entities; it is considered as the architecture's heart and must be implemented as independently as possible due to the project's high modularity**.**
- **Infrastructure layer** consists of the technological tools and implementations realized to support the previous layers, such as database persistence, message sent, etc.

BCs can, for example, have their own UI, application, domain, and infrastructure layers, or they can have one or more layers under their control while sharing the others. The domain layer is the only layer that cannot be shared among BCs in any situation. Indeed, the domain layer is the representation of a domain model and given that a BC refers to a specific domain model, each BC must have its own domain layer.

3.3. Tactical DDD

The following defines different components and concepts related to tactical DDD:

- ***Domain Elements patterns*** – Patterns for defining the data model and its associated functionality (the names of these elements usually constitute part of the ubiquitous language that technical and domain experts share to ease communication)
    - The main part of the model, the **entity**, represents a well-defined notion with its own identity.

- A **value object** is an element that has a specific value, is immutable, and has no identity.
- **Service** represents a domain logic operation that does not conceptually belong to any domain entity.
- **Modules** are useful for grouping services and domain model elements so that developers may quickly see which parts of the domain the module handles.

- ***Lifecycle patterns*** – Patterns that demonstrate how items can be organized, produced, and saved.
  - **Aggregate** refers to the ability to treat a group of items as if they were one; one of these items will serve as the aggregate root, with whom other external items will communicate; the items within an aggregate cannot be accessed by other external items.
  - **Factory** generates new domain objects, particularly aggregates; the presence of a Factory is necessary to permit the integration of various object creation methods.
  - **Repository** stores and maintains the of domain objects; like the Factory, the inclusion of a Repository is required to support the integration of various object storage technologies.

- ***Communication patterns*** – Patterns for managing communication between business domains, particularly when a change in one area causes changes in others.
  - **Domain Event**: consists of events relevant to domain logic and broadcast by a domain object; such events can be subscribed to by other objects, which can

begin new actions as soon as a new event is received; this technique offers a richer understanding of domain object communication flows.

- In contrast to the possibility of storing only the current state of an entity in the database, **Event Sourcing** proposes to store the entity's initial state and then a series of events on that entity; in this way, the entity's current status is the sum of its initial state and all related events; this approach is useful for restoring a previous state or detecting the state of an entity.

During the cyclic process of creating the system architecture, such patterns will be applied. They will be utilized to show how technological implementation can be leveraged to realize relationships between BCs presented in the Context Map and related communications. The final solution should adhere to decoupling principles to the greatest extent possible, allowing for the focus on each sub-domain at a time without losing sight of the whole system.

4. APPLICATION OF DDD TO ACROBA DOMAIN

4.1 Ubiquitous language

This section aims to define and unify words used throughout the deliverable. It gives a quick definition of the robotics words that will be used throughout the rest of this paper, as well as their relationship to the concepts described in the DDD architecture. This term list can be found on the following list:

- **Use Case:** Each of the industrial processes to be optimized using the ACROBA architecture and that have been defined in the project's proposal.
- **Task:** Equivalent term to Use Case.
- **Skill:** Self-contained smaller modules inside a Task, the joint of different Skills solves a Task. These skills can either be

generic, that solve complex operations in a generic way (Learning, perception, …) and that can be used in different Tasks when needed, or more specific related to particular needs in a use case.

- **Drivers, primitives:** Lowest level modules with direct access to hardware or that perform real time and small operations (e.g., CAD matching with point cloud data). A group of these primitives working together perform a skill.
- **High level communication layer:** Layer to transfer information between a certain task and the factory level, using FIWARE and the Orion Context Broker
- **Low level communication layer:** Low latency communication layer used in a distributed architecture (ROS / ROS 2).
- **Task Planner:** Special module, at the same level of Skills, specific for each task, in charge of sequencing the execution of different skills.

4.2 Data and communication model

ACROBA follows a distributed data architecture. This type of data architecture is established in advance by the two frameworks that have been used as building blocks of the project architecture, ROS and FIWARE frameworks.

In the case of communication at the platform level, of robotic elements, ROS is used, which in turn presents a distributed architecture based on the publisher/subscriber model.

In this model message passing is carried out transparently following the ROSTCP or DDS protocols, depending on the used version of ROS.

The messages sent are the standard ones provided by ROS or the ones defined in specific tasks, created using the tools provided by ROS.

In the case of the high-level connection layer, FIWARE is used, which follows a similar message passing model. In this case, the translation of low-level ROS messages into FIWARE messages is performed by using the FIROS tool.

4.3 Bounded Contexts

Bounded contexts can be explicitly related with each use case in the ACROBA project. That is, each use case is a bounded context, resulting in a self-contained domain for each of the situations. Under the Strategic DDD perspective, this initial definition corresponds to the application layer.

The different skills that compose a certain use case, would correspond to the *domain layer,* and the *infrastructure layer* would have its correspondence with the low-level communication layer, previously defined.

5. ACROBA SYSTEM ARCHITECTURE

The system architecture has been defined starting from the requirements listed in deliverable 1.1 and following the results obtained through applying DDD principles to the ACROBA architecture. The result consists in a definition of skills and tasks, to gather, by one side, the needed requirements of the different use cases, and, by another, the transversal functionalities that the ACROBA architecture offers, namely "Robot Cognitive Capabilities", and structured as modules.

5.1 The microservice architecture in DDD

The microservices architecture pattern is built on the concept of individually deployed pieces, where each microservice can be developed and deployed independently of the others, allowing the overall distributed system to have a high degree of decoupling, monitoring, fault tolerance, and scalability. These are the task skills in our scenario.

In the ACROBA architecture (Figure 1), factory requests are handled by an intermediate layer (namely, High Level Communication Layer) that is responsible for the redirection of the request

to the correct skill or task planner. The major advantage of using an intermediate layer is the easy replacement of a certain skill with another one. Moreover, mechanisms for load balancing and error tracking can also be included in this layer. A skill represents a service component, which can vary in dimension and number of modules that it involves depending on its nature.

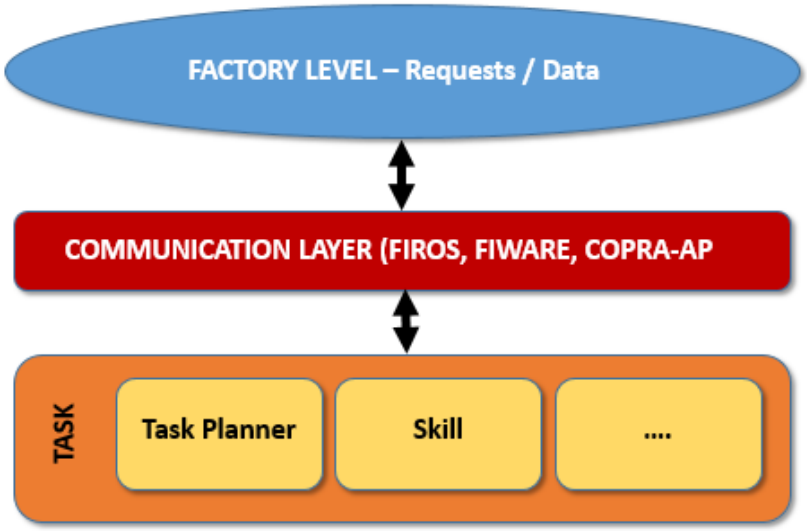

Figure 1. GENERIC ACROBA ARCHITECTURE

Skills architecture has been identified as a more rapid and effective translation of bounded applications present in use cases into technical implementation with the use of DDD for ACROBA. A skill will have its own architecture (which may differ from other skills), will use primitives that are appropriate for its goals, and will oversee its internal organization.

In addition to this, relationships among skills can exist. In ACROBA, skills need to be orchestrated, basically in two ways:

- Some skills (and their internal primitives) need to be executed in a specific order (sequential execution).

- Other skills are triggered by some specific domain event messages (event-based execution).

For this reason, a low-level communication layer has been included in the system architecture: such a layer will be characterized by a hybrid implementation, in order to enable both sequential and event-based execution of skills and modules.

Finally, the architecture will involve also the factory communication layer, decoupled from the lower robotics architecture, by integrating a Gateway (FIROS / COPRA-AP) between both architecture levels, that enables the routing of needed factory information to the correct skill. In order to assist the reader in the comprehension of the overall system, the architectural skeleton represented in Figure 1 is further explained in following sections, starting with the general adoption of the microservice (skill) approach to ACROBA and following with the specific tailoring of the approach to the different use cases proposed in the project.

5.2. Skill based architecture in the ACROBA platform

As described in D1.1, ACROBA architecture can be understood as a set of reusable skills and modules among the different tasks in the use cases. Along with the skills, a group of more transversal modules will be grouped as "Cognitive Capabilities" to be used whenever demanded.

Finally, two communication layers will be needed to create a flexible robotics platform:

- **Low level communication layer (ROSTCP / DDS)**: For real-time communication among skills and contained primitives. The protocol used depends on the ROS version of the node implementations.
- **High level communication layer (FIROS / COPRA-AP)**: To present the data at factory level.

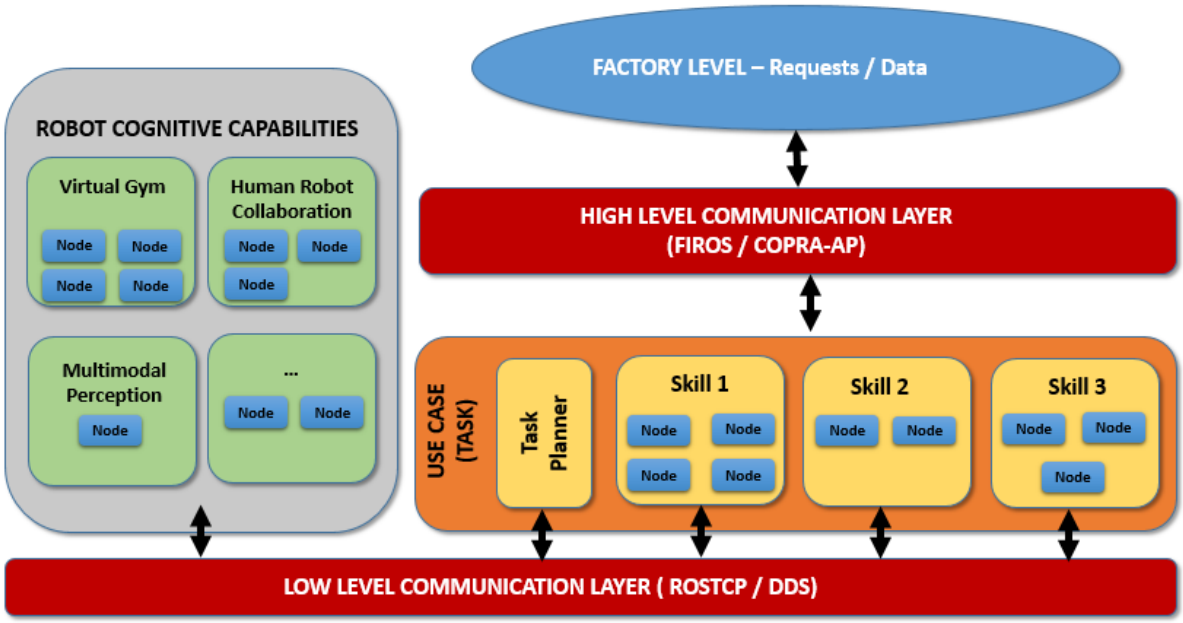

Figure 2. ACROBA ARCHITECTURE FOR A GIVEN USE CASE.

As can be seen in the diagram of Figure 2, each use case is structured in different skills with the task planner (based on behavior trees, state machine, …).

Each of the skills correspond to subtasks to be performed within the same use case. Each skill, in turn, is composed of several primitives or drivers (ROS nodes), depending on the work carried out in it.

These last entities, the primitives or ROS nodes, are the lowest level elements in the proposed architecture, and provide, among other functionalities, direct access to different HW elements or execution of necessary self-contained algorithms (3D matching, for example).

The SW definition language SysML will be used in deliverables D2.1 and D2.2 to define these entities. Therefore, the overall ACROBA architecture uses a DDD approach whereas skills are used for the upper layers, and SysML for the lowest SW layers.

In parallel to the SW and HW definition we specify a series of transversal skills common to all use cases. This group of generic skills have been defined in the project as "Robot Cognitive Capabilities" and will be developed in WP2. They include aspects related to perception and control models, human-robot collaboration, reinforcement learning, artificial intelligence or a simulator environment for the different tasks.

6. ANNEX TO THE DOCUMENT

Due to different visibility scopes established in the grant agreement of the project, an annex document to this deliverable is needed to gather all the required information of skills in the different use cases proposed in the project.

While the created general robotic architecture is at a public dissemination level, proposed use cases of different companies are private.

To have all this information formalized in documents, this deliverable contains the information related to the general structure, and a parallel annex contains the definition of use cases at a skill-level detail

7. CONCLUSIONS

This report presents the general architecture of ACROBA platform. Using Domain Driving Design approach, we have developed a flexible architecture that can be adapted not only for the project uses cases, but also for a wide range of scenarios. We have also defined the skill based on use cases.

The skills needed for each use case addressed in this project are presented in an annex document related to this deliverable. Due to the confidentiality conditions established in the grant agreement, this document is not disclosed to the public.