Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Aerospace Engineering (35^{th}cycle)

Reference Architecture for an Integrated Ground and Space Software for CubeSats

By

Lorenzo Maria Gagliardini

******

Supervisor(s):
Prof. Sabrina Corpino
Dr. Daniel Fischer, Co-supervisor

Doctoral Examination Committee:
Dr. Paolo Marzioli, Referee, University of Roma La Sapienza
Prof. Dario Modenini, Referee, University of Bologna

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Lorenzo Maria Gagliardini
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Abstract

Ever since the first CubeSat mission was launched, the concept and complexity of CubeSat missions has evolved at a pace that current operational system/doctrine cannot match. In an increasingly dynamic space economy, where small businesses have become the norm, innovative solutions that abstract away complexity of bigger missions and increase autonomy of simpler systems are fundamental to reduce operational costs.

It is within this frame that the current study is presented, aiming at developing a software architecture with data handling capabilities for small- and nano-satellites platforms, reducing the resources required for its adoption.

We first investigate the solutions currently available and established in the market, e.g., CCSDS MO Services [1] and SAVOIR OSRA [2]. We assess the European small satellite market needs through a survey with key players in the sector, highlighting which features of these technologies to keep, and which to elaborate and improve. From this survey, we derive the high-level requirements of a software architecture capable of satisfying the reduced complexity expressed by market needs while maintaining the critical data handling functionalities and providing the packets traceability and persistency needed for system monitoring.

The adopted design methodology makes wide use of a Model-Based System Engineering language [3] and common tool, as well as a fast prototyping, for passing from the model to the implementation, in an iterative optimisation process.

Finally, we report the testing of a communication pipeline between components relying on the implemented framework. The results are twofold: the strict data domain segregation adopted in the design enables a high level of modularity for data distribution and processing, making it possible to re-use portions of code when the system scales up; In addition, the file-based configuration logic used for the

framework API makes the plug-in process much lighter for the developer when introducing new applications.

The results show that starting from the existing technologies, it is possible to reduce time and effort required by their adoption. Still, further development is needed in order to asses tested system in an operational scenario, while supporting the arising systems' complexity.

Contents

List of Figures

List of Tables

Introduction

Research Rationale

The future of CubeSats is headed well beyond the initial *CubeSat* standardisation definition, and nowadays CubeSats have much more in common with normal-sized spacecrafts than they used to have at the beginning of their journey. This inevitably involves the world of the *Operations* as well as the systems for their support. In order to keep up with the unprecedented pase at which small-satellites operations are evolving, ESA and industry have developed and are developing ground- and space-based standards, technologies and applications that improve the way of conducting operations. This phenomenon show the high level of competitiveness that is dominating the CubeSat market today, together with the limits that this competitiveness carries around, being hardly committed by the companies themselves.

The market is populated by a wide variety of different companies, in terms of dimensions, resources, and in terms of working areas. Ranging from private launch companies, to small satellite operators, to bus manufacturers, such entities have become the main driver of the space economy. Among those that were more recently born, some have successfully established themselves in the market. The bigger ones often relying on venture capitals, that made them capable of developing their own proprietary software solutions.

On the other hand, there are many reasons that can be attributed to the lack of market success or to the delays that some other companies experience while getting their product to the market. We can point out to the lack of a standardized, open source software architectural reference for mission operations as one of the main obstacles. Without access to proprietary software, these companies are forced to develop in-house knowledge and build and design their own software stacks, a lengthy

and costly process. Moreover, this ad-hoc development further disincentivizes the development of a common architecture; if a company allocates resources to gain a market edge, it is not likely it is interested in losing it by making its software available to competitors.

The architecture that intends to solve such a variety of issues can potentially play several roles and target many purposes. Their identification cannot completely be fixed from the first stages of the design process, and it shall be evaluated in an iterative manner.

Therefore, the project scope is double, and the relative achievements to be gained concurrently: the architecture is born with the objective of becoming a reference i.e., spreading within the market for unifying the software tooling applicable in the space mission scope. This is a multi-step process scoped in the long period, whose first achievement targets the CubeSat environment as the most convenient test-bench and most relevant use-case.

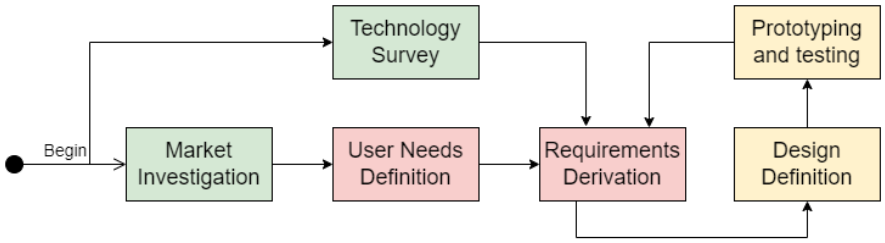

Fig. 1 Iterative process.

In parallel to this, the design and implementation preceding the architecture delivery, follows an iterative process of design and improvement which instead primarily focuses on the software architectural aspects, rather on the strategic role such technology will play. These development stages seek a different approach, played both in the technical and socio-economical fields, whose background logic shall be aligned consistently. The architecture aims at playing the role of a de-facto standard architecture, concurrently maintained, and developed by industry and public agency, namely ESA, in order to:

- Increasing the competitiveness of industry by seeking the low-cost/fast-delivery paradigm.

- Supporting the interoperability and integration in the system of units coming from diverse supply chain.
- Delivering more services with little impact on costs.
- Widening the range of applications involving new stakeholders and business models.

In this context, thanks to its role, ESA is interested in addressing existing limits, and commits itself for finding solutions impacting the CubeSat market actors.

Thesis Layout

The present Section provides the reader with the logic behind the thesis structure. The thesis is composed of five chapters that will guide the reader through the project, prom the very initial stages to the conclusions and next steps.

Chapter 1: Operations

Role of Chapter 1 is to introduce the reader to the research topic by providing a literature review that covers the key aspects of the design process leading to software architecture. The chapter touches the topic background by providing motivation to the proposed work and examples, by emphasizing on the use of Model Based System Engineering methodologies that we will find in further chapters.

Chapter 2: Market Investigation

Chapter 2 reports the market investigation conducted in 2019 in order to get familiar with the CubeSat market environment. The chapter presents the Research Questions and Objectives formalised during the first steps of the project and aims at collecting the perspectives of the entities populating the market (via a proper questionnaire) with respect to the Reference Architecture project, as well as presenting a survey of those existing technologies that could play a role in the new REFA architecture definition. After the outcome of the questionnaire is analysed and the technological survey is concluded, a set of User-Needs is formalised, that presents the working condition from which the design shall start.

Chapter 3: Architecture Design

Chapter 3 is the core chapter of the thesis, and covers all the aspects related to the design of the architecture, as well as all the elements involved in the design. As a continuation of Chapter 2, from the set of User Needs a list of high-level requirements is derived, that lead the design process. A detailed description of the derivation process from user-need to requirement is provided. As everything is set for starting the proper design, a description of the adopted design methodology is given. Afterwards, the two main layers of the architecture are explained in details.The chapter provides a complete description of the core elements of the architecture and of the interaction layer linking them, and here is where the *Service* introduction of Chapter 1 gets useful.

Chapter 4: Proof of Concept

Chapter 4 is devoted to the proof of concept following the design process. Objective of the Chapter is to translate the theoretical design into a working environment. This is done in three consecutive steps. First the implementation criteria are explained. Such criteria are not to be as stand-alone choices, as they follow the same modularity and re-usability principles adopted in the design process. As second, a focus on the software deployment is given, so to understand the structure of the code, as reflecting the implementation criteria. Last but not least, the validation of a set of components is provided, by showing their working behaviour observed in test campaigns.

Chapter 5: Conclusions

Chapter 5 discusses on the results of the project, on its limitations and on its current potentialities, for further developments.

Chapter 1

Literature Review

1.1 Cubesats

Traditionally, satellite production follows a limited model, often producing only one or a few units. In such cases, rigorous testing is essential to ensure the functionality of the system when deployed. Both physical and digital testing, including modeling and simulation, are conducted extensively. When there's only one opportunity to achieve success, it becomes crucial to ensure its practical reliability. The expenses associated with these programs can be excessively prohibitive, particularly when considering the cumulative costs of various segments such as hardware, software, and launch. The testing required to validate the design can significantly escalate the overall price, particularly as the complexity of the system increases. In contrast, CubeSats offer a more affordable and easily reproducible alternative.

The stringency of testing can be balanced with the reduced consequences of failure. According to Robert Twiggs, co-creator of the CubeSat Design Specification [12], in the book "Space Mission Engineering: The New SMAD," [13] some degree of failure is acceptable in the pursuit of technological advancement. Although initial failures can be costly, they ultimately contribute to lowering costs by providing valuable information.

1.2 Ground and Space Segments

Today the Ground Data Systems (in short Ground) and on-board software (in short OBSW) of a space mission have separate development lifecycles. These systems are developed independently as two distinct software architectures with a well-defined interface, which is captured in the so-called space to ground (S2G) ICD. For ESA missions this interface is typically based on the ECSS Packet Utilisation Services (PUS) standard [7].

While the successful history of ESA missions is a practical evidence that the current paradigm works, it shall not befog the missing opportunities for achieving a higher level of reuse across the Ground and Space segments, which would lead to a much more optimised and integrated development lifecycle, higher degree of automation and more advanced concepts of space mission operations. ESA is investigating in the context of a TRP study [14] a new harmonised architecture for the Ground and Space Segments for future missions, which takes a different view to the Ground and Space software systems, conceiving them as two elements of one distributed system by merging the advantages of these optimized tools.

This harmonised architecture combines the underlying concepts of three relevant architectures, the service-oriented architecture of the CCSDS MO and CCSDS SOIS, and the component-based architecture of European SAVOIR initiative (On-board Software Reference Architecture – OSRA). The long-term perspective of this harmonised architecture is motivated by the strong logical dependencies between the Space and the Ground systems and the fact that today many artefacts with similar purposes are redeveloped independently for the two segments. Examples hereof are application logic (source code), automation procedures, validation artefacts and simulation models.

One of the high-level concepts of the harmonised architecture is to expose the native interfaces of selected OBSW components (OSRA components) as MO Services, which can then be consumed by other on-board components or by Ground components, using different communication protocols (bindings). The independence from the implementation and communication technology is in the core of the CCSDS MO architecture. The harmonised architecture and its related tooling, shall allow:

- Rapid development of space missions through auto-generation of large portions of OBSW and Ground components;

- Specification of unambiguous, machine readable formal interface specifications for the space to ground interface;
- Integrated validation of space and ground systems at different stages of the development lifecycle (today performed only during System Validation Tests, SVTs;)
- Higher level of reuse of simulation models, automation procedures, validation artefacts and even source code between space and the ground

Notwithstanding its potential, this harmonised architecture needs to be assessed carefully and validated before implementation on a large-scale. CubeSat projects might represent valuable use cases for development and application of this architecture. On the other hand, the emerging (European) CubeSat context would benefit of an integrated SW Reference Architecture (REFA) for Ground and OBSW in terms of:

- Increasing competitiveness of industry (low-cost/fast delivery paradigm;
- supporting interoperability and integration in the system of units from diverse supply chains;
- delivering more services with little impact on costs;
- widening the range of applications involving new stakeholders and business models

In the context of the development and V&V of ground and on-board software for CubeSats, especially oriented to mission operations, some key points are identified which might deserve the attention of developers and operators, in the areas of:

- Standardization. Today, any developer has its own customized standard and software implementation, although open source SW and tools are used. In the software testing process, no common rules are specified and agreed;
- Automation. At mission operations level: human intervention is often required, especially for managing contingencies. Command and data handling effort increases with the quantity of data generated by the mission, and the complexity

of mission architecture (e.g. operations of mega-constellations). At code generation level: often most part of the SW is written by a developer that implements functions and algorithms as specified in documents

The poor standardization and little automation of today systems leads to not-optimised mission operations management and software development and V&V.

1.3 Industry and Academia

The number of CubeSats in orbit has increased in the last twenty years, when back in 2003 six of them were deployed in low Earth orbit (LEO) in the first multiple CubeSat launch [15]. CubeSats are nano-satellites that share a common interface and equipment standardization. The basic CubeSat is a cube-shaped platform 10 cm on a side whose mass is less than 1.33 kg, but multiple unit CubeSats exist and are becoming popular [12]. The CubeSat standard was born within the academia with a pure educational purpose [16].

Low-cost and fast-delivery features were key parameters playing a crucial role for the standard definition. These factors led to a high number of CubeSats developed by universities in the first decade, mainly with education as primary objective; technology demonstration, scientific experiments and/or Earth observation have been secondary objectives for most of them [17]. However, in the last years, governments, space agencies and private companies recognised CubeSats and nano-satellites as attractive space platforms to pursue a broad set of mission goals, including science, technology demonstration, communications, and Earth observation, with a significant cost reduction and a relatively faster development time, from design to operations, compared with traditional larger-satellite missions [18].

CubeSats allow building brand new architectures, which would be unattainable with bigger satellites. Constellations of nano-satellites in LEO are becoming a reality [19][20], while the CubeSat community is exploring the possible applications of nano-satellites for interplanetary missions [21][22], and two units for technology demonstration of future missions to Mars have already been launched [23]. This new paradigm requires the development of adequate technology to enhance CubeSat performance and increase mission success, while keeping the cost of the mission within an acceptable cap.

Whilst ongoing technology miniaturisation, radiation-hardened COTS electronics and tight system integration associated with CubeSats will enable a significant reduction in the space and launch segment costs, the operations costs do not scale down with spacecraft size/mass unfortunately. This will be critical especially for high-end CubeSats, such as for interplanetary scientific and exploration missions, and for LEO applications involving multi-unit architecture (e.g. EO and/or telecom mega-constellations, orbiting vehicles inspection missions). The future CubeSat missions will then require advanced concepts of mission operations and a high degree of reliability. It must be said that educationally-driven CubeSat missions have often failed. The failure of CubeSats is dominated by infant mortality, which can be traced back to design weakness and/or ineffective Assembly-Integration-Verification (AIV) planning and execution. An effective and exhaustive Verification and Validation (V&V) process may help to increase the reliability of small-scale satellites [24].

1.4 Reference Architecture

Creating a new model for each project, considering the various needs, stakeholders, goals, hardware, and facilities, can be inefficient. Instead, it may be more effective for organizations with common elements to establish a common baseline. This baseline would incorporate relevant aspects of the organization and provide engineers with a more advanced starting point. This is where a reference architecture comes into play. A reference architecture serves as a repository of knowledge, offering guidance and rules for structuring, classifying, and organizing a model. It essentially encapsulates the collective architectural knowledge accumulated over many years of work [25]. According to Maier and Rechtin [26], systems architecting occurs when the problem is not yet fully understood. Architecture forms the foundation for a structured problem-solving approach, ensuring the consistent application of principles necessary for project success.

A reference architecture is most effective when there are significant similarities between projects. Within an organization, a reference architecture can serve as a guide for employees, outlining expectations for product development and business practices. In a specific domain, a domain-specific reference architecture ensures consistency by establishing common foundations for all products being developed

within that domain. The reference architecture should encompass relevant standards, legislation, domain constraints, and mandatory frameworks.

There are also external factors that drive the need for a reference architecture, including increased interoperability, adaptability, and shorter time to market for individual products. Having a reference architecture facilitates greater compatibility and integration among various components and systems, allowing for quicker adaptation to changing market demands.

The purpose of creating a reference architecture is not solely to ensure the success of one project. It is designed to capture the best practices from prior designs and projects, promoting continuous growth and evolution. The knowledge gained from modeling is reinvested back into the architecture, enabling it to incorporate existing architectural efforts while also focusing on future innovation and the development of new products.

Chapter 2

Market Investigation

2.1 Research Questions and Objectives

Operations do not necessarily scale down with the size of the spacecraft [27]. CubeSat systems and missions can increase in complexity and this brings an increasing amount of management and engineering capabilities [28]. The goal of the current project starts from this assumption and targets a software architecture devoted to CubeSat capable of reducing the effort required by CubeSat mission's operations. The system architecture design can heavily impact the time and cost at operations and therefore the complexity and variety of aspects reported in the previous chapter shall be handled at design-time.

The present project started by stating a list of questions and objectives that were given at the very early stages and have been hereafter formalised. They focus on both technical and strategic aspects, mostly related to the CubeSat market environment and the opportunities the market itself can offer.

2.1.1 Research Questions

The short list of questions, address both the role such project can play in the next years' market development, as well as the constraints leading the design.

Time and Cost-saving Architecture

"Is it possible to define a time- and cost-efficient architecture for future space and ground segment developments?"

The first research question focused on the difficulties that small-sized companies experience when bringing their products to the market. A solutions like a reference architecture (REFA in short), which aims at spreading over the market shall align with time and cost requirements imposed by the market dynamics. For such a REFA to be effective, it shall be able of imposing itself because of its capabilities to save both the time of development and implementation and the relative costs.

Market Needs

"Is it possible to model an architecture on current market needs?"

Achieving time and cost efficiency in a market populated by a wide variety of different companies may not be trivial. We cannot exclude the contrasting needs from different stakeholders. And at the same time we shall not befog the opportunity to shape our architecture on a common solution, a compromise which includes any party in the game, where possible, and avoids selective choices.

Re-inventing the Wheel

"Can we exploit existing market technologies, without reinventing the wheel?"

One of the main risks when introducing a new standard, is the limited added value i.e., limited amount of solved problems when compared to the overhead that the new architecture carries around. In other words, what is supposed to be *the final* solution, threatens ending up for being just an additional one. Aligning to the marked needs is a good starting point, but at what cost does it come?

2.1.2 Research Objectives

The objectives of the the project target the definition of the architecture that best answers the research questions, by driving the design choices. These have been formalised in two sentences that summarise the purposes of the project.

Harmonised Solution

To develop an on-board and ground software architecture which includes the best features of the existing technologies in the space domain, and which provides a harmonized implementation of a core set of resources and the framework for accessing them.

The architecture shall attempt to define mechanisms which allow the combination of existing technologies whilst maximising the benefits for the end-users. In the present work, we shall specify an architecture which owns the requirements needed for a wide adoption, starting from the given state of the constituent technologies. This research work shall propose the characteristics of the architecture together with potential ways in which they can be achieved.

Targeting the User-Needs

To align the architecture development to the market investigation results, in order to target the needs coming directly from the companies' developers.

The present research objective underlines the importance of taking into consideration the market perspective along the work, as the final target of the current project. The architecture sophistication, as one the principal design aspects, highlighted in the first research objective, shall not befog the importance and associate risks of shaping the architectural solution in accordance to socio-economical aspects.

2.2 Software Architectures and Standards Survey

The first step of the investigation starts from the technologies populating the market, and it is important mentioning here certain key aspects that might help us better defining the context that the REFA design will target. The discussed points may seem quite technical, especially at this stage of the REFA design, nevertheless it is important bearing in mind that such discussion will help us critically evaluating the potential solutions during the architecture survey as well as during the analysis on the market investigation results and during the proper design of the architecture.

2.2.1 Survey Scope

Far from providing a complete list of all the possible aspects that are implied in the REFA definition and design, we provide this list of examples to give the reader an idea of the topics, questions and perspectives that shall be assessed, answered and evaluated during all the stages of this project. These are hereafter reported.

- One of the main points opened on the architecture design from the very beginning, concerns the domain of applicability. The REFA interest focuses on whether such architecture should target the actual on-board software domain or rather a higher, application-level environment. This aspect implies considerations on the governance model leading the system's development and on the implications the same architectural choices induce on the management logic. This brings us to the next point.
- A second key aspect concerns the development management. It is important to define whether such development can be left to the final user e.g., as it can be an open-source project, or rather a delivery granted by an official development entity should be foreseen. Depending on the previous point e.g., in the case this software is running in a user-space i.e., in a safe environment, any new application can be plugged into the framework with almost no hazard. On the other hand, in the case this is the primary system handling the spacecraft activities, the verification and validation needed for avoiding conflicts and inconsistencies, directly affect and potentially threaten the system's health, becoming therefore mandatory. And since the scope of the project is not only technical, we can ask:

- Which implications would such choice have out of the technical scope of the current project? May the REFA become concurrent to other on-board software rather than running on top of it/being controlled by it? And if this is the case, from a development point of view:
- Does it make sense to start prototyping either such two different systems with the same approach or rather a first version of the software shall be located closer to the user space? And therefore;
- Which functionalities of the software do we want to address? Which of them can reside in the user-space and which shall be part of the on-board software? Are there technical solutions that can be adopted only at certain levels?

Having familiarised with the breadth of the project's scope, we have one more instruments that will help us answering these and more questions through the survey on the existing architecture solutions and during the next design phases.

In the present chapter we examine four technologies, both spread within and outside of the CubeSat industry. These are:

- core Flight System, [4]
- SAVOIR On-board Software Reference Architecture, [2]
- Mission Operations Services [1]
- Consolidated Architecture from the MOSS study [29]

2.2.2 core Flight System

The core Flight System (cFS) [30, 31] is a widely adopted flight software (FSW in short) in many NASA missions. It is developed and maintained at the Johnson Space Center and its purpose is to reduce the costly and time-consuming process of developing software for spaceflight missions. Its flexible and layered architecture creates a development environment where system integrators can assemble a significant portion of a software system for new missions, test platforms, and technology prototypes, resulting in reduced technical, schedule, and cost risks.

The idea behind the design involves the reuse of cFS elements that not only include the software, but the artifacts such as requirements, design, test procedures and results, and documentation, saving projects and missions the cost and effort of reinventing these pieces over and over again as was done in the past. The end result is an architecture that can evolve and is flexible.

From the objectives point of view, the cFS is quite close to what the REFA project aims at reaching i.e., a modular, flexible and, in the end, reusable set of artifacts that carry around time and cost saving implications. Let's look at it closer and from a structural point of view.

cFS architecture

The cFS is composed of five main elements [32], covering from the hardware adaptation elements necessary for the system to be run on different physical architectures, to the application standard functionalities remaining fixed between missions. Such elements are hereafter reported:

- Platform Support Package (PSP), is the set of packages needed for making the architecture runnable on different machines, while maintaining performance constraints. Each mission is expected to customize a Platform Support Package.
- Operative System Abstraction Layer (OSAL), is a small software library that isolates the Flight Software itself from the Real Time Operating System. With the OS Abstraction Layer, flight software such as the Core Flight Executive can run on several operating systems without modification.

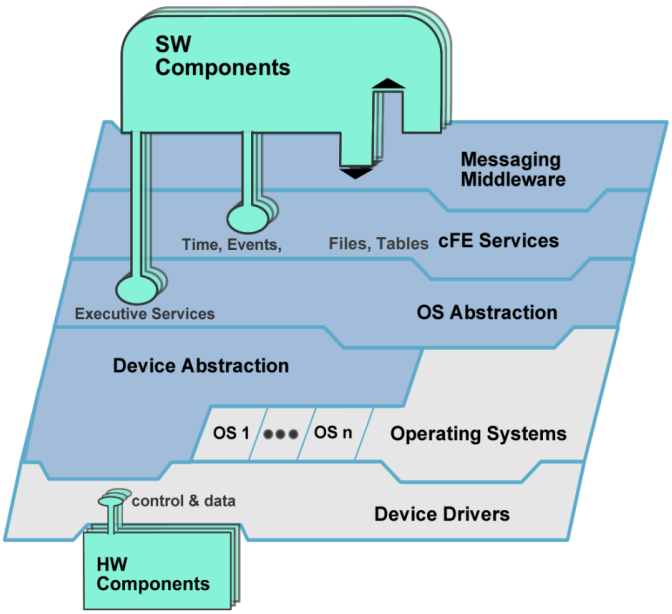

Fig. 2.1 core Flight System architecture [4].

- core Flight Executive (cFE), is a set of mission independent, re-usable, core flight software services [33], applications, and operating environment. It provides standardized Application Programmer Interfaces (API) and supports software development for on-board FSW, desktop FSW development and simulators. The core Flight Executive includes five core services:
  - Executive Services, among the others it provides ability to start, restart and delete cFS Applications and manages data preservation.
  - Event Services, provides an API for sending asynchronous debug, informational, or error message telemetry to ground and for filtering event messages.
  - Software Bus provides a portable inter-application message service, routes messages to all applications that have subscribed to the message and reports errors detected during the transferring of messages.
  - Table Services, manages all cFS table images.
  - Time Services, provides a user interface for correlation of spacecraft time to the ground reference time.

- core Flight Applications [34]. Running on top of the core flight executive, the applications are developed for mission-specific purposes. Such applications use the core services for basic functionalities exploitation and communication.
- cFS Libraries.

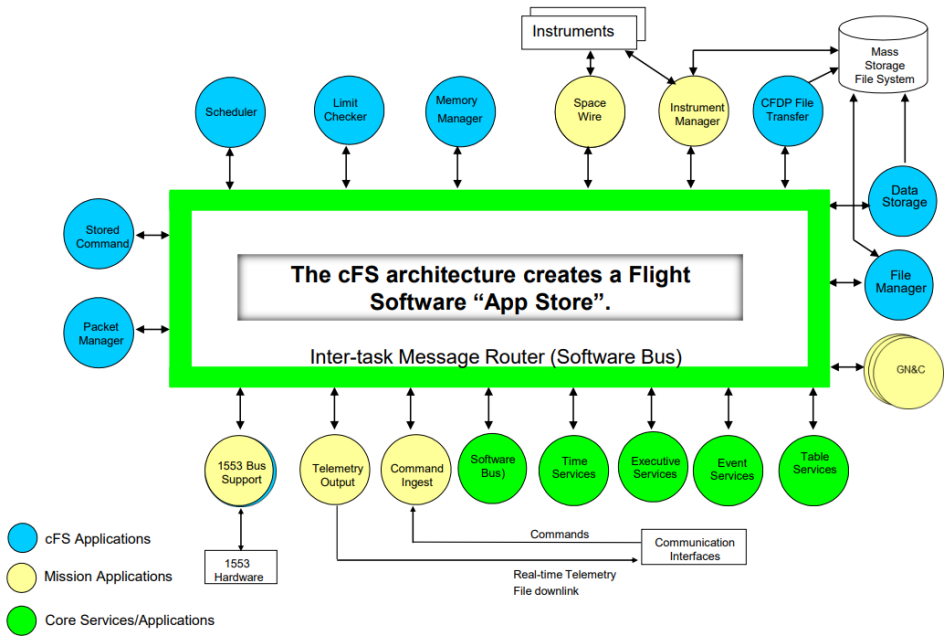

Fig. 2.2 core Flight System logic [4].

The cFE/cFS provides a layered assembly of components, so that every component manages its component-specific set of operations that are then offered to other applications as an API.

Despite its smart approach to the problem, we highlight here two major problematics that we think the described architecture maintains. These are hereafter reported and will be assessed during the REFA development, in the next sections of the present document.

- The first issue is mostly technical. The cFS correctly keeps services segregated from components, nevertheless every component i.e., application, develops its own internal logic and operations i.e., data models, then offered to other components in terms of APIs i.e., there is no native common data-model for distributed-component applications.

- The second issue is related to management and logistic. Despite its smart approach and efficient architecture, the amount of work required for its adoption does not scale down to the size of small missions (and companies e.g., hiring 10-20 people who have to cope with the whole spacecraft project).

The example provided by the cFS, together with the other existing technologies, had a strong impact in the direction the REFA would take. Notwithstanding the mentioned issues, it shows relevant characteristics from which the REFA can benefit, and despite a re-invention of the framework, it is considered convenient to adopt some of its basic principles and to build of top of these. In the next chapter the REFA design are discussed more in details so to make such commonalities clearer for the reader.

2.2.3 SAVOIR OSRA

The SAVOIR-FAIRE working group was set up in 2009 to determine and specify a reference architecture for onboard software (OBSW) [35], together with necessary interfaces, to smooth the way that OBSW is developed and procured in Europe. The group issued an initial document describing the User Needs and High-Level Requirements determined by industry representatives and capturing the key aspects of the proposed reference architecture.

It was therefore suggested that a reference architecture is needed to actively assists in the management of this complexity making software development, verification and validation, and maintenance easier and more manageable. This was approached through:

- Through strict encapsulation of various elements of the system, using standard, or regular, interfaces;
- Through the separation of concerns within the software system, such as the separation of functional and non-functional aspects of the design.

These two aspects were satisfied by the application of a component-based software engineering (CBSE) paradigm which permitted the construction of software from multiple components each with well-defined interfaces and which cleanly separate functional and non-functional aspects.

The SAVOIR presents an overview of a consolidated avionics functional reference architecture derived from the SAVOIR avionics reference architecture objectives and functions. It gives through an example a possible physical implementation of the reference architecture. It does not only limit the definition to the software architecture definition but spans over the avionics definition and interfaces.

Among all such model, the On-board Softwware Reference Architecture (OSRA) is the part of the model focusing on the reference architecture for spacecraft onboard (flight) software developed as part of the SAVOIR initiative. The OSRA is intended to address the major needs for software development.

The OSRA adds the model-based approach to the component-based design, to allow the development of onboard software in a more efficient and flexible way than traditional methods, without sacrificing robustness.

Let's investigate point by point the most relevant aspects of such architecture and comment them with respect to the REFA design described in the next chapter.

OSRA Architecture

The OSRA [36] is based around a three-layer architecture. The application functionality is defined by components, which exist in the Component Layer, along with the specification of the desired non-functional properties. The containers which envelope components, and the connectors which bind component interfaces, exist within the Interaction Layer. Both of these layers are dependent on the application function and rely on the underlying Execution Platform for application-independent services.

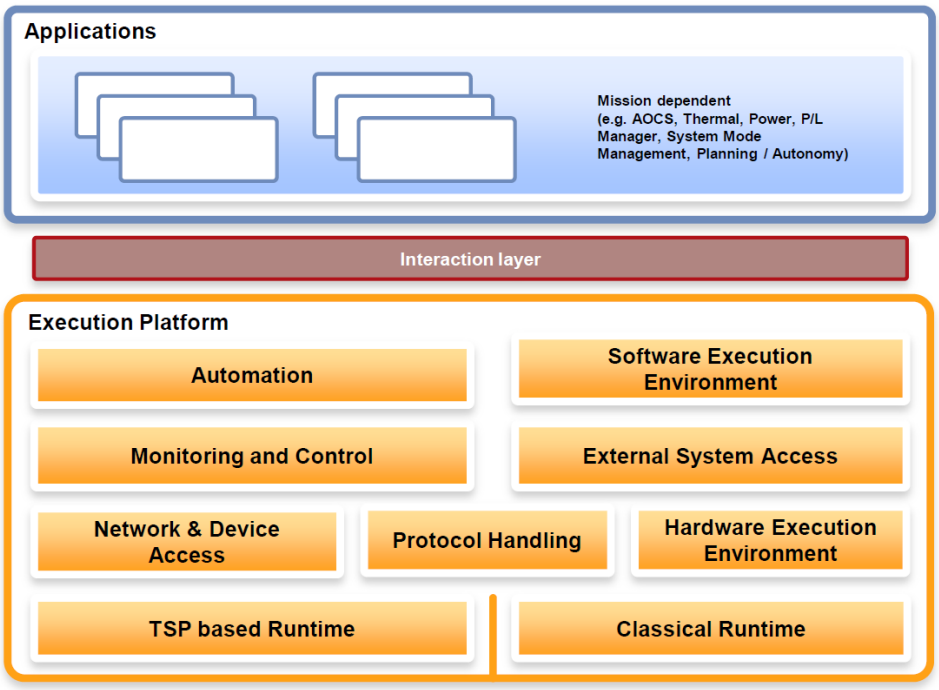

Fig. 2.3 OSRA layered architecture [2].

The Component Layer is where the application software is designed and implemented. The functional aspects of the application are captured as components whilst the non-functional concerns are captured declaratively as non-functional properties. A component in the OSRA is intended as composed of the following properties:

- Has a well-defined interface.
- Has explicitly captured dependencies, which form part of its interface.
- Is purely functional, containing only sequential behaviour with no timing or synchronisation.

Such layaring [37] represents the state of the art in term of flight software. Such three design hinges will remain valid in the REFA design, unless for realization constraints, as we will see in the next chapters, mainly impacting (and requiring) the strict segregation between interface (as general as services) and functions (dictated by the applications' data models).

In the OSRA a component is defined by its component type which provides and requires specific interfaces, each of which is defined by an interface type. To use a component, the component type is implemented and then instantiated, creating a component instance for each use.

The idea of modular elements remains in the REFA, since every interface, as either required or provided is part of the component's dependencies (inheritances) and therefore instantiated at run-time depending on the type i.e., service & sub-service specified.

The Interaction Layer acts as "middle-ware", implementing containers and connectors and matching the services provided by the Execution Platform to those required by the containers and connectors. The Interaction Layer is tailored for a given assembly of components and is specific for a given Execution Platform. In this respect, it is the "glue" which allows a platform-independent component model to be executed on a platform-specific Execution Platform.

Within the OSRA a component in use is completely enveloped by a container. A container is specific to a component instance and is responsible for the realisation of non-functional properties associated with tasking, synchronisation and timing, using the mechanisms offered by a given execution platform. For example, if a non-functional property specifies periodic execution, the container would be responsible for executing the component operation periodically, using a task provided by the execution platform.

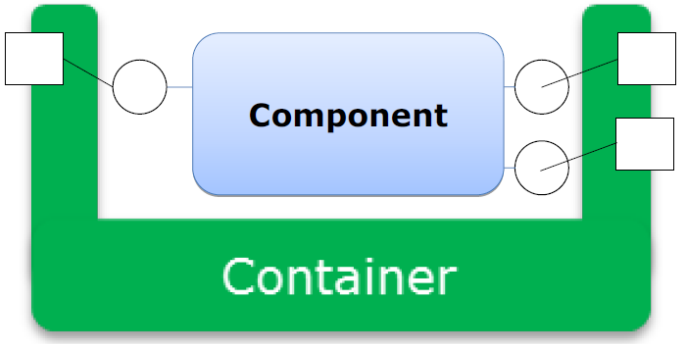

Fig. 2.4 OSRA component and container [2].

For the REFA the concept is lost, since there is no self-standing isolation element mediating the interaction between interfaces at the application level (it is not true

anymore at the level 1-2 of the OSI stack). It is rather true that the component is responsible for making use of a set of standard interfaces depending on the component's configuration i.e. specified before the start-up.

The Execution Platform [38] provides all the underlying services [39] necessary to support the execution of components. The Execution Platform provides application-independent resources to the Interaction Layer and is dependent only on the underlying hardware, not on the application or mission.

As discussed later, the REFA project targets the functional properties of the architecture and their interaction, by providing a structured layerization, the physical platform on which the architecture is run and relative efficiency considerations are not included in the study.

The REFA project inherited a lot from the SAVOIR-OSRA, not from the engineering solutions but rather from the methodology followed during its development and the criteria adopted at system high-level design.

More in particular:

- It was decided to follow the same methodology of market investigation, by first analysing the outputs of a questionnaire submitted to the entities present in the CubeSat market, in order to collect a first list of User Needs, and from these extrapolating a list of requirements to be applied to the high-level design of the architecture; moreover
- The methodology inherited from the OSRA did not cover the only procedure leading to the investigation but also the design, namely the Component-Based Software Engineering (CBSE) which requires:
  - Strict segregation of concerns; and
  - Loosely coupled set of components.

  We will find such features in the REFA design, impacted both by the adoption of a Model-Based language, in Section 3.2.1, and by the introduction of the so-called Data Models [40], discussed in Section 3.4.

2.2.4 CCSDS Mission Operations

Since the year 2003, the Spacecraft Monitor and Control working group of the Consultative Committee for Space Data Systems (CCSDS), participated by members of 10 different space agencies, has been working on the definition of a Service Oriented Architecture (SOA) consisting of a suite of standard end-to-end Mission Operations (MO) services [9, 1] between functions resident on-board a spacecraft or based on the ground. The MO services are intended to cover all the activities needed for operating any possible mission and, despite its more recent introduction and very limited deployment on existing missions when compared to the PUS , the gap between the two standards is being gradually filled. OPS-SAT, launched in December 2019, is the first small-sat (CubeSat 3U) using the CCSDS MO services as the mean for exchanging data among the experimental platform on-board and the experimenters on ground. OPS-SAT runs the NanoSat MO Framework [41] which is built upon the CCSDS Mission Operations Service Framework and thus inherits the same agnosticism towards the transport layers used in the space system. This allows the conceptualization of a "multi-segment" software framework dedicated to nanosatellites that is neither limited to the space segment nor the ground segment, and instead it covers both segments simultaneously. At the time this survey is being written, the latest publication of a new MO area of services, the so called Common Services (CCSDS 522.0-B-1), has been issued by June 2020, in addition to the previous COM services [5] and M&C services [6]. From the architectural point of view, the CCSDS Mission Operations standard introduces a strict and coherent task subdivision, within the same service layer, which threatens aspects going from the abstract interface with the applications to the abstract interface with the communication protocols. In the middle, concepts like the Service Specifications, the Common Object Model (COM) and the Message Abstraction Layer (MAL) are introduced to guarantee an effective independence between the three levels of the system (i.e. application level, service level and transport level). Let's see them in detail.

MO Architecture

From a structural point of view, the the CCSDS Mission Operations [1] service framework can be internally decomposed into several elements, vertically structured so that the upper layers rely on the lower layers for operating.

On the bottom layer, the MO standard provides the mapping between the MAL specification e.g., [42], and transport technology/encoding.

On the upper layer, the framework interacts with the components via an adaptation component, which translates the service API into the components API and vice versa.

In the between, within the framework itself, MO provides the definition for the Message Abstraction Layer (MAL) [43]. The MAL defines an abstract definition for a data types set that are used for further data structuring. These same data types are used by the same MAL definition for designing the abstract MAL message header i.e., a set of information used for addressing packets among the MO framework.

In addition to this, the MAL defines the interaction patterns that any service can adopt for data exchange purposes.

Beside the MAL, the CCSDS MO standard provide a data model definition for handling services related data. The COM both provides a syntax for connecting the MO objects and a set of rules for identifying the objects.

In the present section we will present the main elements of the MO framework.

Message Abstraction Layer

The Message Abstraction Layer [44] represents the core of the Mission Operations standard and provides the abstract definition of aspects like:

- Interaction patterns according to which any service shall be expressed in order to exchange data;
- Set of basic elements composing the service messages, which constitute the bricks from which any further service-specific structure shall be built; and
- Addressing methodology for service messages within a distributed architecture, by gathering the information functional to the messages exchange in an abstract MAL message header.

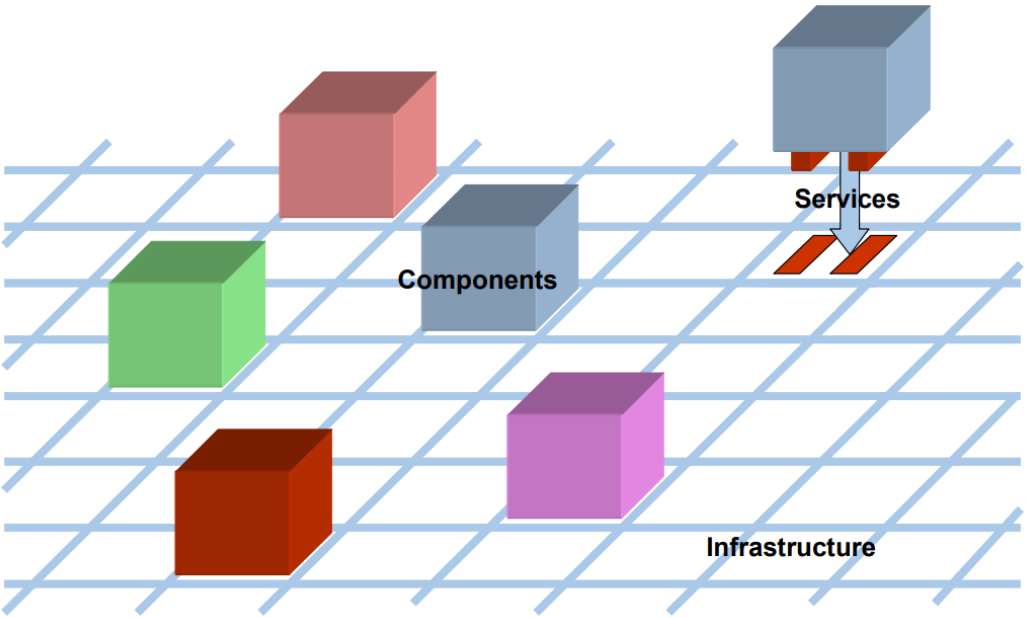

Fig. 2.5 MO Service Architecture [1].

Thanks to these features defined within the MAL, the specification of a set of MAL-consistent services remains transport and encoding-agnostic. In other words, the MAL is meant to give the means for managing any new service compliant with the MAL prescriptions regardless of the implementation language and transport protocol chosen.

The specification of a service, as already mentioned, shall be MAL compliant, and starts from the selection of the interaction patterns defined within the MAL specification, meant to be specified for any service operation. As opposing to the PUS, any service operation must rely on six predefined interaction patterns: SEND, SUBMIT, REQUEST, INVOKE, PROGRESS and PUBLISH-SUBSCRIBE, each being a slight development in complexity of the previous one. The interaction patterns include the generation of acknowledgments, progress, execution and completion reports as well as error reports. Each pattern shall specify the set of statuses that both the provider and consumer can attain during the execution of the pattern exchange. The interaction patterns specifications do not standardise the only space-to-ground interface. The MO services specification does not conceive a marked distinction between application processes located either on-board or on ground. The majority of these interaction patterns are composed by more than a single message instance

exchanged between entities which may be either distributed or located in a same asset. As a consequence, the CCSDS MO covers any possible message exchange between peer entities, regardless of the domain, session and network zone, being all aspects considered in the MAL specification. The second needed element for the specification of a service, as for the PUS, is the definition of the data structures used for the semantic representation of the message content. The service data structures rely on the building blocks defined within the MAL and therefore the MAL implementation will be capable of handling such kinds of messages. Nevertheless, any service specification defines its own data structures different from any other, composed by the fundamental MAL data types. The MAL message header provides an abstract specification for addressing and handling any MAL compliant message. The header contains all the fields needed for identifying the location of the application processes involved in the interaction and the characteristics of the message itself (e.g. time stamp, interaction type, interaction stage). The fields composing the MAL message header are hereafter reported.

Common Object Model

The Common Object Model (COM) is the second fundamental MO specification and provides a standard object model for the MO services. Whilst the MAL provides the building blocks to be used for the operations, the COM provides the object definition upon which any service can rely to manage service-specific data structures. Whilst the COM does not limit what may be considered an object by a service specification, it does define how objects are referenced and how they can reference each other. Each object can link to up to two other objects. The two links have different roles, the related role is expected to be used to link to a related object (for example a parameter value object could link to a parameter definition object), and the source link would be used to link to an unrelated object (for example the operator who requested its value change). It is service specific how these links are to be used. Each linked object can define its links (so a parameter definition object could define a related link to a parameter name object if this was required) and therefore provides the ability to have ‘n’ levels of hierarchy. In addition, the same specification of the COM defines a set of basic services upon which any further service specification can rely. In the Mission Operations framework, any data handled by the framework shall be first modelled. The data representation model offered by the Mission operations services

i.e., the Common Object Model (COM), consists of four main points [5], section 2.2.1:

- An object is defined as a thing which is recognised as being capable of an independent existence and which can be uniquely identified.
- There are no requirements on what an object may be.
- It must be possible to uniquely identify an instance of an object so that it can be referenced.
- Each service that utilises the COM must define the object, or set of objects, that form the data model of the service.

In other words, the framework handles the only objects that have firstly been modeled accordingly to the COM rules and any data handled by an MO Service, shall be formally represented in memory as a self-standing object, by means of an identity, definition, time instance, etc. This applies to any kind of data, from the representation of a parameter, to an activity object. The consequence of such definition implies a duplication. As first, we find the instance (e.g., the parameter value) taking part to the framework activities, exchanged from component to component by means of packets, which then get stored in memory too, and whose existence we can assume being independent from the framework itself, since it is originated by a component. As second, we find the framework-domain representation, compliant with the COM rules for the object identification, whose instance is then exchanged via packets, as just mentioned.

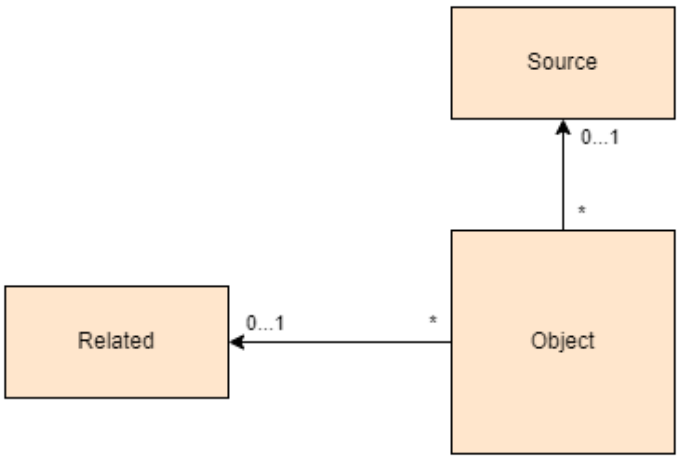

Fig. 2.6 UML *Source* and *Related* links an rules [5].

It is worth noticing that such framework's feature, consisting in the identification and classification in a framework model of data instances originated at component-level, does not serve the interfaces standardisation process, being characterised by the Service Body definitions. Such feature requires instead the components to comply with the service's specific rules for objects identification, for them to exploit the data instance transfer capabilities offered by the services. Only after such kind of "object registration" is finalised, the component can fill the service API with the identified data instance.

In particular, the CCSDS Mission Operations standard provides a central data model (the Common Object Model) that any service specification uses for structuring the service-specific objects. Starting from the definition offered by the Common Object Model, here we highlight the main features characterising the relations between COM-compliant objects:

- Any object relies on two formal links for referencing other objects. These are the *Source Link* and the *Related Link*, reported in Figure 2.6.
- Every service defines its own service objects and the relations between service objects.

Table 2.1 MO submitAction operation

Operation Identifier	submitAction	
Interaction Pattern	SUBMIT	
Pattern Sequence	Message	Body Signature
IN	SUBMIT	actionInstId : (MAL::Long) actionDetails : (ActionInstanceDetails)

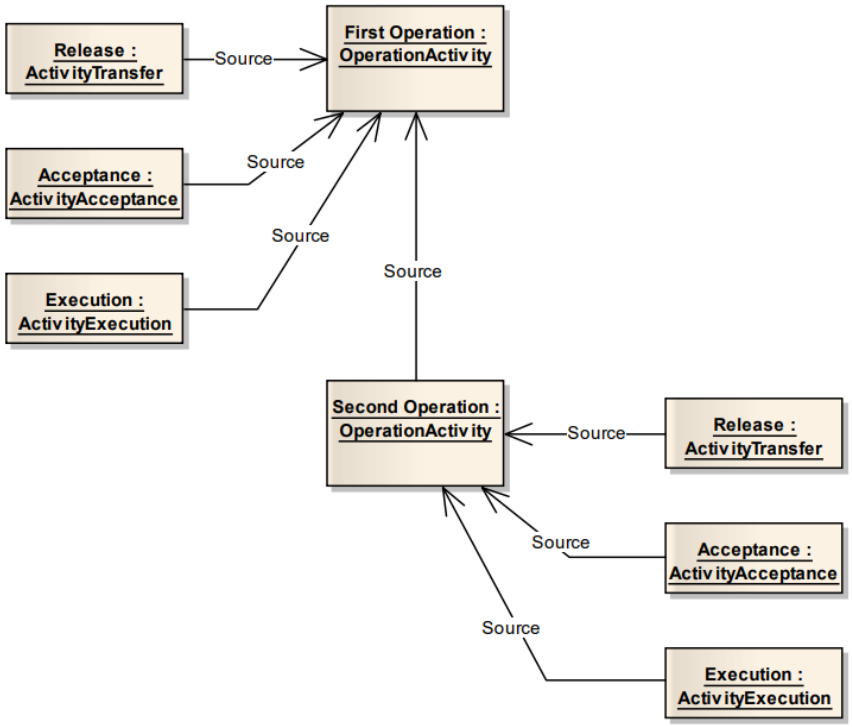

Fig. 2.7 Service object's relations [6].

- The service objects' relations defined by every service are formalised by the solely use of the mentioned Source and Related Links, reported in Figure 2.7.

Such data model, based on two link, is adopted for the description of any service objects' relation. Such data model is also strictly tied to the services data structures, and APIs more in general. Many service data structure indeed host fields devoted to describing such kind of relations, as shown in Table 2.1. It is not possible for any application to get rid of such model and develop a custom one, to be eventually translated into an application-specific service API.

2.2.5 ECSS Packet Utilisation Standard

The ECSS Packet Utilisation Standard (PUS) is now at its third issue[7, 45, 46]. Since its first publication, it has been widely developed and updated in order to meet the more recent operational needs. As a result, it has reached a high level of maturity in terms of requirements' satisfaction and is nowadays adopted by most of the existing ESA (and non-ESA, in Europe) missions. The PUS has standardised 20 tailorable services, from which any adopting mission can choose, which are intended to cover all the missions' specific needs [47]. In case the standard services are not sufficient for satisfying the mission requirements, the good practices for aligning the new ad-hoc services to the PUS standard are provided. The standard services' names and enumerations are reported hereafter:

service type	
name	ID
request verification	1
device access	2
housekeeping	3
parameter statistics reporting	4
event reporting	5
memory management	6
(reserved)	7
function management	8
time management	9
(reserved)	10
time-based scheduling	11
on-board monitoring	12
large packet transfer	13
real-time forwarding control	14
on-board storage and retrieval	15
(reserved)	16
test	17
on-board control procedure	18
event-action	19
parameter management	20
request sequencing	21
position-based scheduling	22
file management	23

Fig. 2.8 Common PUS services [7].

Taking into account what already described in the introduction chapter, a more detailed description of the PUS services is here provided. Each service message structure is meant to semantically reflect the service-specific information. This is achieved by designing a specific data field set for each service message instance. Any data field composing the service message relies on a set of common data types shared among all the services specifications. The data types shall be capable of representing any possible kind of data: Boolean, Enumerated, Unsigned Integer, Signed Integer, Real, bit-string, octet-string, character-string, Absolute Time, Relative Time, Deduced and Packet. This means that any data field composing the service message shall be defined and constrained within the defined type set. Any application/device relying on the PUS shall, in the end, comply with the service interface expressed as a specific combination of data fields. This implies that adapting the interface provided by the component to the interface offered by the service is a developer's responsibility, thus left as an implementation concern. It is remarkable that the ECSS-E-ST-70-41C document standardises the only space-to-ground interface. This aspect has a great impact in the distinction between the ground-based processes and the space processes, sunning on-board. The PUS does not specify the interaction patterns involved in the transactions but how to relate the services needed for performing the desired operation (e.g. the generation of a TM[1,3] report in case of Success Start of Execution). This reflects on the conceptual definition of Telecommand (request type, directed from ground to the spacecraft) and Telemetry (report type, directed from the spacecraft to ground) and allows the classification of any service transaction within three different classes:

- The service instance can be a request related transaction type and therefore, as a consequence of a request (TC) message, a response (TM) message is dispatched from the spacecraft.
- It can be an autonomous data reporting, such as a cyclic TM parameters update.
- It can be and Event report, so being generated autonomously on-board in response to a specific change in on-board conditions.

The identification of each end-to-end transaction with a specific TC/TM packet exchange constitutes the base service model for the Packet Utilisation Standard. Nevertheless, some application processes (peer entities on-board) may require the

on-bard mutual exchange of messages. The mechanisms used to exchange such on-board messages are mission-dependent and therefore outside the scope of the PUS Standard.

2.2.6 MOSS

The MOSS study [48, 49] is an initiative funded by ESA an carried out by *OHB*, *RHEA* and *Bright Ascension*, in the context of a research for future spacecraft activity, which analysed three key technologies in the on-board software frame:

- CCSDS Mission Operations Services (MO Services).
- CCSDS Spacecraft Onboard Interface Services (SOIS); and
- the SAVOIR-FAIRE Onboard Software Reference Architecture (OSRA).

with a view to consolidating them into a single harmonised architecture.

The MOSS study is particularly representative in the survey here reported because provides a first attempt to merge existing technologies that were not natively thought to work together. The components definition provided by the OSRA as well as the MO services definition (both mentioned above) are widely pertinent to the scope of the REFA project. On the other hand, definitions like the SOIS services, which address lower functionalities, do not apply to our case of study. Therefore in the present section we will focus on the functionalities offered by the first two technologies and on how they can interact for offering a more sophisticated architecture.

Also in this case the initial phase of the activity studied the three technologies carefully and elicited a set of User Needs and High-Level requirements for each one.

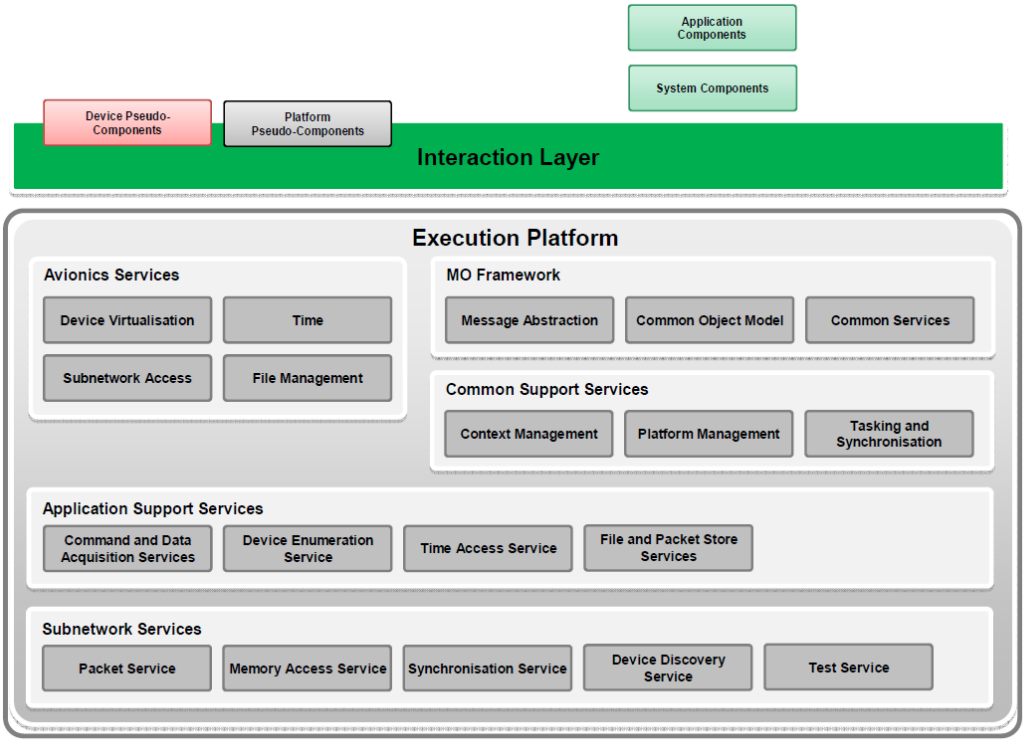

Fig. 2.9 Potential Structure of the Harmonised Architecture.

The primary output of the activity is the consolidated architecture [50], a reference architecture combining MO services, SOIS and the OSRA into a single architecture for application across space and ground segments. The activity provided a concrete specification for this architecture which highlighted a number of issues with each of the technologies which hamper easy integration.

Further steps in this activity defined the high-level characteristics of a harmonised architecture which permits tighter integration of the technologies and full realisation of the consolidated User Needs. This long-term approach requires some adaptation of the individual technologies.

The main aim of the consolidated architecture is to permit the combination of the three constituent technologies in a single system such that the original User Needs and Requirements of each technology are still realised. The High-Level Requirements document [51, 52] captured this as a single list of consolidated requirements. The intention is that the consolidation can be achieved in two steps, as follows.

- In the short term, the aim is to permit the use of the OSRA and SOIS with an MO framework, using MO service definitions to define the interfaces to OSRA components, elements of the OSRA Execution Platform and SOIS devices. This requires defined mappings between MO service specifications and OSRA component interfaces, and the SOIS device model (as captured by SOIS electronic data-sheets) and MO service specifications. Additionally, the short term consolidated architecture can specify expectations on the conceptual and logical structure of the OSRA Execution Platform. This results in a single system with defined regions, or elements, which apply the OSRA. We refer to this as the consolidated architecture. [53]
- In the long term, the aim is to permit the application of a harmonised component model across the complete space-ground system. This should allow all elements of the system to realise the development-time and maintenance benefits of a component-based and model-based system. The intention is to make parts of the component model optional, such as constructs which fully realise the separation of concerns and non-functional properties. Additionally, the model must be extended to permit various types of dynamic binding. Where separation of concerns constructs are applied, and the "local" system is restricted to have static binding, this corresponds to the OSRA component model and can therefore be analysed as such. We refer to this as the harmonised architecture. [14]

In order to realise application functions, multiple components may be connected together, this is referred to as binding. Interfaces are defined separately from a component and can be used in the definition of multiple component types. Each attribute on an interface is typed and may be read-only or read/write. Interface operations are defined with a signature, determined by an operation name and an ordered set of parameters.

Although a number of MO books define a component as the provider and consumer of services, components are not captured in an MO system. Service consumers and providers, however, are uniquely identified by an address, specified in the form of a URI.

As such, it might be possible to adapt component interfaces to function as MO services; however, this would be a mistake as there is a fundamental semantic difference between the two. Whereas an MO service defines a mechanism, a component

interface defines an entity. For example, where a component interface identifies an attribute, MO may use the Parameter Service to provide access to parameters, the types of which are defined in terms of the COM. Where a component emits or receives specific events, MO may use the Alert Service to provide a means for publishing and subscribing to alerts (which are, again, defined in terms of the COM). Although interface operations and service operations may look alike (and could easily be made more so) we contend that this similarity is deceptive and belies a difference in semantics. The choice of provider of a service operation in MO can be conducted at run time, meaning that the service operation is semantically independent of provider. A component operation is bound to a specific provider such that the operation is tied, once deployed, to a particular provider.

2.3 Market Polling

We started the survey in mid-2019 by submitting a questionnaire to several private companies and public bodies in the CubeSat market ("the entities"). Of these, twenty-four expressed their perspective on the existing standards and best practices when applied to CubeSats. To guide our work and gauge their interest, our investigation is centred around two topics: if the European standards satisfy their needs and how proposed solutions could bridge that gap, discussing how a to formalise a reference software architecture devoted exclusively to CubeSats operations. Particularly, we are concerned with how this architecture would differed from their current adopted solutions and whether they would be prone to adopt it in the future. After the response period closed, we reviewed their answers and characterized their distributions: entity size, mission needs, and unique entity characteristics.

2.3.1 Statistics

In the present section, the results of the questionnaire as well a discussion to understand the pig picture is provided. The discussion will go through the main results. The text of the questionnaire can be found in Appendix A.

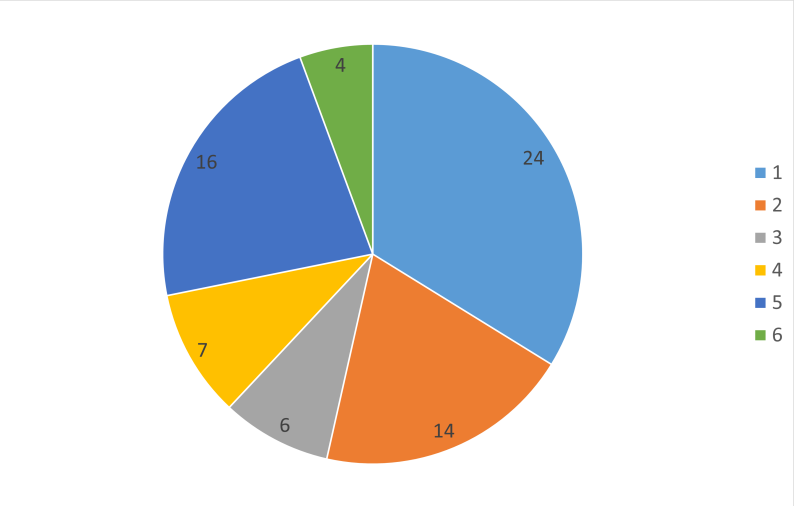

Fig. 2.10 Standards usage: *Have you used any of the following in your projects?* (1) CCSDS Space Packet Protocol [8]; (2) CCSDS Mission Operations Services [9]; (3) CCSDS Spacecraft On-board Interface Services [10]; (4) CCSDS Space Link Extension [11]; (5) ECSS Packet Utilization Standard [7]; (6) Savoir-Fair OBSW REFA.

The first questions, whose answers have been reported in the Figure 2.10 and Figure 2.11, intend to point out which existing standards, references, and best practices (e.g., CCSDS, ECSS standards) freely available to the European market the polled entities are familiar with, and which the respondents currently use in their CubeSat projects.

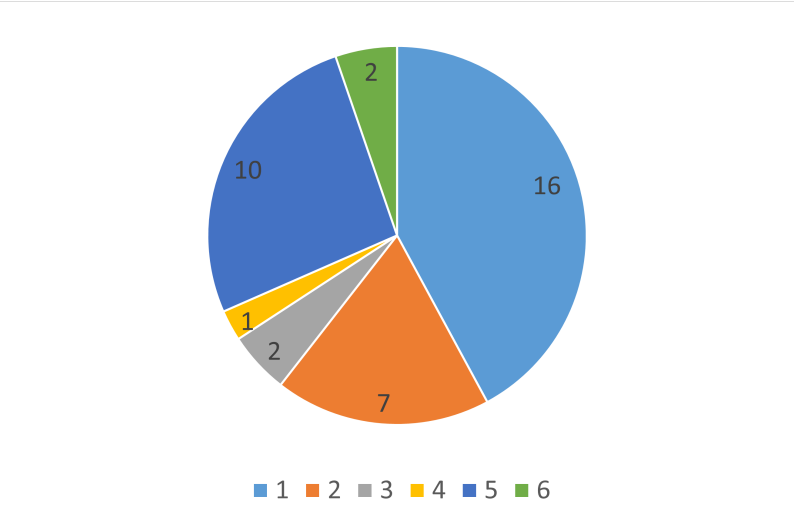

Fig. 2.11 Standards familiarity: *Are you familiar with any of the following?* (1) CCSDS Space Packet Protocol; (2) CCSDS Mission Operations Services; (3) CCSDS Spacecraft On-board Interface Services; (4) CCSDS Space Link Extension; (5) ECSS Packet Utilization Standard; (6) Savoir-Fair OBSWREFA. CCSDS, Consultative Committee for Space Data Systems; ECSS, European Cooperation for Space Standardization; OBSW, On-board software; REFA, reference architecture.

Such list was selected to cover the message transport issues both on-board and on-ground, the Space-to-Ground interface, and the orchestration of on-board activities, so to cover most of the macro-areas touched by the CubeSat operations.

The companies were also asked to point out additional references they apply, might these have not been considered in the questionnaire. Beside the CCSDS and ECSS standards, the companies mentioned other references they adopt in their CubeSat projects, listed below.

- Other ISO standards.
- Other OMG standards.

- MISRA coding standards.
- CubeSat Space Protocol.
- Compass Stack.
- Other ECSS standards (tailorable and non-).
- Non-disclosed in-house-developed standards).
- UNISEC Global.

According to our survey, 67% of the entities adopt the CCSDS Space Packet protocol (SPP) in their CubeSat project as a solution at the network layer, showing that it has become a de-facto standard even though most of them are not developing projects in collaboration with a public space agency. This provides a common baseline mission information data exchange. At the same time, respectively the 29% and the 42% of the entities adopt the ECSS Packet Utilisation Standard (PUS) and CCSDS Mission Operations (MO) services, as these provide a reference for managing data at application (operational) level. Of these entities, 43% and 50% respectively are contractors of the European Space Agency, which demands the use of these standards. Nevertheless, this suggests the CCSDS MO Services are more widely spread among CubeSat missions than ECSS PUS Services.

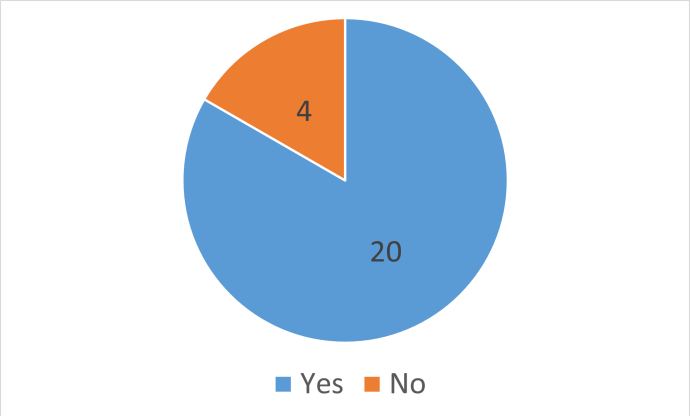

Fig. 2.12 REFA usage distribution: *Do you apply "a"/"your own" REFA in your projects?*

Focusing on the second half of the questionnaire, we are interested in understanding how many companies are adopting 'a'/'their own' reference architecture (REFA)

in their project development process and to which extent. As shown in Figure 2.12, the majority (83%) of the polled companies say they use a reference architecture developed in house or outsourced. Of those, 25% use an architecture that covers software interfaces while 35% apply a reference architecture for communication protocols.

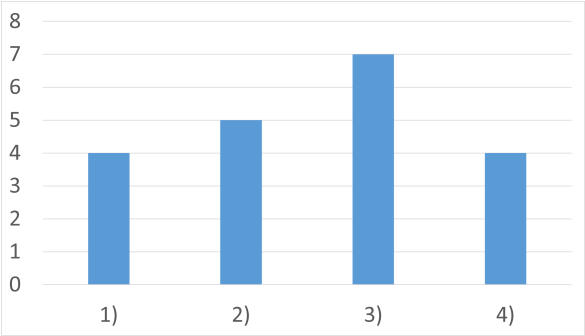

Fig. 2.13 REFA covered areas: *At what level do you apply "a"/"your own" REFA?* (1) At hardware interface level; (2) at software interface level; (3) at communication protocol level; (4) at operational concept level.

Such a result aligns with Figure 2.10 and Figure 2.11, which highlights the application-level software i.e., MO services, PUS services, and communication protocols i.e., SPP, CSP, as the most adopted technologies by the polled entities. Indeed, they tend to take advantage of existing standards as a starting point for their in-house designs, notwithstanding their non-mandatory adoption i.e., imposed by a contract. We lastly asked the polled entities to deepen the software REFA topic, Figure 2.14 and Figure 2.15, about the possibility to adopt a common CubeSat reference architecture. We are interested in understanding how this choice would impact their business model, and what should the reference architecture encompass in order to bring an added value.

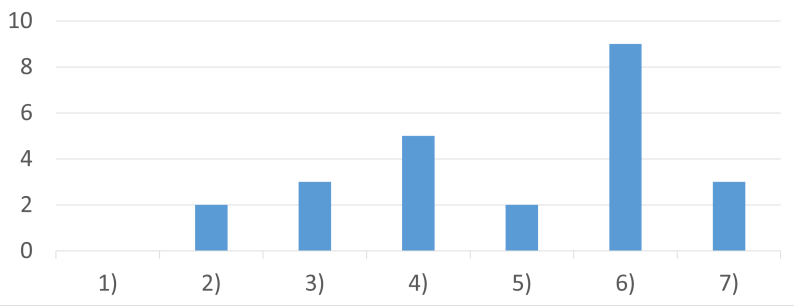

Fig. 2.14 REFA perspective distribution: *Would a CubeSat REFA bring value and facilitate your business model?* (1) No; (2) yes, if it is at hardware mechanical interface level; (3) yes, if it is extended at device interface level in form of APIs; (4) yes, if it is at on-board communications protocol level; (5) yes, if it is at space to ground interface level; (6) yes, if it also encompasses the composition of functional components on-board and on the ground; (7) yes, help in research or in student recruitment or in academic purposes. APIs, Application Programming Interfaces.

This first question evaluates what design level the reference should target (e.g., software/hardware interfaces/protocols), for tackling technology gaps. The second inquiry we make is to understand how to propose such a reference (e.g., as a paper-only design, in the form of Open-Source). What is immediately clear is that the request for an architecture covering on-board and on-ground functional components is, by far, the solution that best fits market needs. According to the polling, 46% of the entities ask for an architecture involving the core functional components as a non-mandatory reference. This aligns with what we reported before, showing companies and public entities taking advantage of existing technologies as a starting point for their internal design process, which in most cases results into an ad-hoc reference architecture. This further highlights the market need for a REFA and provides a line of research for a further development in our study. Moreover, from the buyer's perspective, the presence of an open-source reference implementation would mean greater competitiveness and higher quality products, with vendors offering their own design, but still compliant with the reference architecture differing for implementation details.

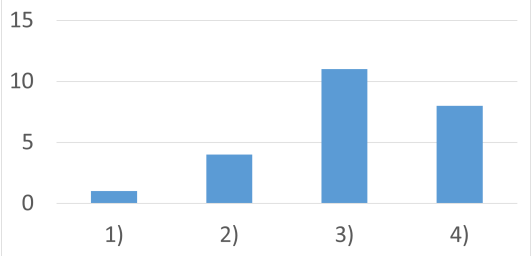

Fig. 2.15 *How far shall a REFA go?* (1) Tools for auto-generation of code; (2) high-level architectural design—paper only; (3) open-source reference implementation of the core elements of the REFA (as reference not mandatory to take); (4) Market Place with competitive vendors offering compliant and competing implementations of elements of the REFA.

2.3.2 Results

From the analysis done on the results of the questionnaire, three important behaviour the polling highlighted are here reported.

As first, the polled entities take advantage and make wide use, in their CubeSat projects, of existing technologies and standards for space application. An example of this is provided by the transport/network protocols, such as the Space Packet Protocol [8], or to the CubeSat Space Protocol [54, 55]. Another example is made by the families of services for operations, like the CCSDS Mission Operations services, or the Packet Utilisation Standard [7]. All of these have become de-facto standard for anybody in the market nowadays. As a first point, we highlight that exist a set of standards that are already spread among companies devoted to CubeSats.

The questionnaire also highlighted that the same companies use the mentioned existing technologies as a starting point for further developing their own in-house reference framework or reference architecture. The questionnaire shows that exists the trend of looking for a baseline, for a solid starting point, on top o which it is possible for the companies to focus on their further products or services rather than re-inventing the architecture wheel.

The polled entities are by the vast majority prone to the adoption of an open-source common software REFA as long as it provides a set of core functional elements on top of which application-specific solutions, compliant with the architecture, can be developed and customised.

Table 2.2 User Needs

UN-ID	User Need Text
UN-001	The architecture shall take advantage of existing standards, references, and best practices.
UN-002	The architecture shall allow the usage/introduction of other standards.
UN-003	The architecture shall offer a baseline on top of which further solutions can be developed.
UN-004	The architecture shall avoid unnecessary constraints.
UN-005	The architecture shall be easily tailorable.
UN-006	The architecture shall balance the proper level of details.
UN-007	The architecture shall allow vendors offering compliant and competing implementations of elements of the REFA.

Such results of the described polling were then reported into a list of User Needs, reported in Table 2.2, summarising the main elements that shall be taken into account in the definition of the requirements leading the REFA design.

Chapter 3

Architecture Design

3.1 Requirements Derivation

The identification of the proper level of details in the architecture definition can be considered as the main driver of the design process. The aim for becoming a spread de-facto standard, implies the capability of reducing at most the unnecessary constraints coming from the design process while providing a solid baseline that serves as a common reference. The identification of such aspects whose definition is deemed relevant for the architecture and those who can instead be accounted for tailoring processes is of paramount importance. The entire set on User Needs evolves around this hinge concept which we meet by making strategic choices in the design process, as we will sudden describe. It is worth to bear in mind that such Needs we are proposing do not constitute by any means a technical description of the architecture, and their lack of technicality would not allow such purpose. The statements reported in Table 2.2 are solely intended to resume the user needs as they result from the questionnaire. The present chapter provides a discussion focused on the system level, in order to better understand the considerations which led the requirements' derivation. An explanation of the derived requirements is provided and a 1-to-N mapping from the User Needs to the applicable set of High-Level Requirements, reported in Table 3.1, is described hereafter.

By the elicited UN-100, reported in Table 2.2, we do not intend to specify which of the existing standards to make use of and which to discard. Such indication serves instead for identifying the areas this architecture will span. Taking advantage

of the existing standards, references and best practices, it is possible to cover the different architectural domains. Such architecture shall not limit the specification to the only on-board segment but shall address either on-board, on-ground and Space-to-Ground functionalities. In SYS-0100, defining an on-board and on ground architecture results in multiple achievements, both from the User Needs alignment perspective and from the analysis done on the proposed questionnaire's results. The on-board segment constitutes the focus of the present work. Nevertheless, a functional set of core on-board components cannot be considered as a self-standing entity. By defining SYS-0100, we intend here to guarantee that the provision of those ground components, needed for run-time control, make the architecture an effective, functional set of tools for engineers operating the spacecraft.

UN-002 constitutes the other face of the medal for UN-001, and it was indeed important, in the interpretation of UN-001, not to impose a selection of applicable technologies/standards. In seeking flexibility, by fixing the requirement SYS-0200 we intend to avoid an architecture's specification which refer to particular implementation choices, e.g. Transport/Network protocols; Java implementation language. The inter-process communication, i.e. component-to-component communication technology and implementation choices are left to the customer. Such feature plays a game-changer role in an architecture definition which aims at spreading among the most different customers. Fixing such choices at a too early stage would instead impact into its adoptability and would mean a cut off for potential many of the customers. SYS-0200 is a direct outcome of this consideration.

The translation from UN-003 to Requirement is straight forward. The identification and definition of a framework of core functionalities and transport components shall provide a solid baseline on top of which further solutions can be added. In SYS-0300, defining a baseline core of functionalities, i.e. application-level components, needed for controlling the on-board and ground activities shall constitute a minimal, yet fully functional starting point for further tailored improvements (more granularly specified in the next requirements) and additional components. Thanks to this core set, required by SYS-0300, the customer shall avoid re-inventing the wheel spending time in designing the spacecraft bus, and rather investing in mission-specific solutions.

The UN-004 balances the detailed level of definition of the core components of the architecture coming from SYS-0300. The need for avoiding unnecessary

constraints is required by two main needs. The need for the user to internally replace specific functionalities of the components, depending on the mission specific performances required; and possibility for the implementer to impose its own transport solution when getting to the inner components functionalities. Regardless to the component-to-component communication interfaces, the intra-component communications shall remain transparent to the architecture definition. This aspect is of paramount importance and therefore we consider that any application-level component aspect, e.g. execution timing and synchronisation, will be potentially compliant with mission specific requirements. Bearing in mind such considerations, the flexibility expressed by SYS-0400 and SYS-0500 foresee the possibility for the implementer to add and customise application-level components and internal components communications.

UN-005 differs from UN-003 for the non-trivial implications. As first, tailoring the core components means that the architecture shall allow substitution of existing functional block, by maintaining the component API. SYS-0600 intends defining a component's interface model the customers can align to, in the customisation process, and focus on the translation to the mission's-specific and/or customer's specific technology, which remains transparent to the architecture specification. By following SYS-0600, it is possible to derive SYS-0700, which concurs in achieving tailorability. By maintaining the internal modification of any functional component transparent to the other components, allows concurrent developers, working on different components, to not interfere with each other as long as they comply with the component model API. SYS-0700 guarantees transparency. It is important to consider that such internal modification shall never imply the deletion, omission, of the functions on which the other core components of the architecture rely, e.g. time-tagged commands parsing, which would affect the overall system's functionality. This requirement is expressed by SYS-0800, and the identification of such core elements is part of the architecture design process.

UN-006 poses a condition that cannot be directly translated into requirements, but that rather can further be evaluated by imposing specific conditions to our architecture. The proper level of granularity is reached as first by mapping any architecture macro-functionality, e.g. command ingestion, to a unique functional component. By making this segregation, it is possible to treat separately the granularity issue for every component, and varying such granularity for specific components, depending on the customer's need. For achieving internal granularity, SYS-0900 allows decomposing

each application-level component into lower functional elements. From what stated by SYS-0900, it is then possible to separate design concerns depending on their specific granularity, for each component. This will provide the means, at further development times, for identifying the proper level of definition as we dive deep into each component. It will also be possible for the customers to impose the needed granularity level for the tailored component. The same granularity is ensured by the proper separation of concerns. As coming from the marked polling analysis, UN-007 shall be framed into a specific market model. The current work targets small companies which can take advantage of the specification introduced by the architecture design. In such perspective, the architecture intends to be offered to the customers, i.e. not imposed as a mission requirement, as stated by SYS-1100. Therefore, aim of the architecture is not to reach the rank of a standard. The market model that best aligns to such doctrine is the Open-source model, which facilitate the adoption process, as remarked by SYS-1200. As a result, by fixing SYS-1300, the aim for realising a common repository where common components can be chosen/adopted, differing for implementation details as proposed by the customers' community, allows a central entity, i.e. ESA, maintaining the reference guidelines for adoption.

Table 3.1 High-level Requirements derivation

Requirement ID	Title	Text	Reference User Need ID
SYS-0100	System's Domain	The architecture shall define functional onboard and on ground components	UN-001
SYS-0200	Implementation	The architecture shall not tie users to a specific technology/implementation	UN-002
SYS-0300	Component Functionalities	The architecture shall provide a framework of core functionalities	UN-003
SYS-0400	Adaptability	The architecture shall allow the user to replace the composing elements depending on the specific mission's needs	UN-004
SYS-0500	Inter-Process Communication	The architecture shall not define the interprocess communication technology/implementation	UN-004
SYS-0600	Inter-Component COmmunication	The functional components' interactions shall be expressed in terms of abstract interfaces	UN-005
SYS-0700	System Composability	The internal modification of any functional component shall be transparent to other components	UN-005
SYS-0800	Architectural Structure	The composition of the core functional components shall ensure the overall system's functionality	UN-005
SYS-0900	Component Decomposition	Each functional component shall be internally decomposed into the different processes taking part to the activity	UN-006
SYS-1000	System Compositionality	Each system functionality shall be mapped to a different functional component	UN-006
SYS-1100	Market Model	The architecture shall not be mandatorily adopted	UN-007
SYS-1200	Licencing	The architecture shall provide Open-Source Reference Implementations of the core elements of the REFA	UN-007
SYS-1300	Documentation	The architecture shall be presented as a high-level descriptive standard with guidelines for its adoption and tailoring.	UN-007

3.1.1 Pros & Cons

For evaluating the pros and cons of the proposed architecture, we shall understand which aspect of the architecture will effectively ease the work for the adopters and which aspects potentially induce some drawbacks. The harmonisation process, as presented by the MOSS study in Section 2.2.6, provides well-aligned set of components and layers whose interaction with each other is mediated via a set of adaptation elements and a methodology for automating the adaptation.

It is worth bearing in mind that an adaptation process is needed for any non-native interfacing operation. Adopting a family of services, like the MO services, requires the application-level components to align to such choice. This can be achieved in multiple ways, and is worth mentioning two of them.

- As first, it is possible to imagine the app-level components as only MO Services users. This means that the components' role is solely to "collect" a set of services together and to orchestrate them as a single entity, or node. From a Service perspective this is indeed what an app-level component is. But if no further considerations are made on the nature of the functional components, it is much likely that tieing components to specific services will induce loosing the flexibility and abstraction that MO strains to reach.
- A second consideration can be made. Taking into account the general description provided by OSRA, for components interface, it is possible to keep the app-level components development segregated from their translation into services, whichever those are. This is why an adaptation element, between component interfaces and services interfaces is needed.

Nevertheless, in any real word application, realising such adaptation requires the introduction of an overhead, i.e. the adaptation components development and maintenance, which constitutes an addition to the system complexity and, as for every software tool, implies a cost. Here is why it is worth considering what kind of facilitation and drawbacks this segregation induces, though motivated by flexibility.

3.1.2 Harmonised Architecture

Given the premises, the solution that best aligns with the considerations above shall include a layered architecture where we can identify three main layers:

- A layer defining the only service framework used for data exchange among the architecture. Much likely the MO service framework, such services' families shall only cope with the common - horizontal - interfaces definition. From a structural point of view this is the lowest layer of the REFA i.e., the one providing communication capabilities, and to which any upper application shall in the end align. From an implementation point of view it can be considered as one of the data-models on which the architecture is based, similarly to what the *Software Bus* of the *cFS* is.
- A mid-layer defining the functional specification of the on-board applications intended as end-to-end systems handling operational data, including elements potentially belonging to different components i.e., on-board nodes. This mid-layer provides application-specific interfaces and includes the core functionality set that enables a basic range of operations and which can be further extended as needed by the customer. The data models combine the very convenient modularity feature, and the native interface enabling the overhead reduction whn coming to the components interaction.
- A top layer whose function is bridging between applications i.e., data-models, in order to put together functionalities depending on standard operations activities and mission-specific activities requirements. Bridging between applications and combining their functionalities enables a higher level of complexity in the operations while maintaining a relatively simple system.

3.2 Design Methodology

In the present section we described the methodologies adopted for modelling and reporting the architectural concepts, features and characteristics to the developer that will adopt it.

The design methodology involve both a model-based methodology, used as the principal description of the architecture, partially merged with a document-based for those areas for which such methodology is considered more fitting e.g. the present document for a discursive description of the architecture concept.

A model-based system engineering activity was conducted so to provide a *SysML-compliant* description of the architecture, as providing a convenient instrument for describing either the inner organisation of the architectural elements i.e., the Block Definition Diagrams (BDD), the internal composition i.e., Internal Block Diagrams (IBD), the functional description of the activities i.e., Activity Diagrams (AD) and a description of the interactions occurring between the architectural elements i.e., Sequence Diagrams (SD).

For the reference architecture it was considered convenient the use of independent models i.e., set of diagrams, for each of the architectural elements to be modeled.

In addition to this, it was considered more handy for such architecture to keep the requirements definition seprate from the model itself, to be then presented an additional document, for each of the architectural elements.

As a result, the architecture will be accompanied by a general description providing the main concept leading to the design i.e., the present document, and, for each architectural element, a set of models and a requirements document which can be separately handled by the developers. This design choice was inherited by the CCSDS blue books used for describing the MO service families.

3.2.1 Model Based System Engineering

One of the most useful definition of Model Based Design (MBD) [3] states that MBD is a mathematical modeling-based method for designing, analyzing, and validating dynamic systems. During the design phase, the MBD methodology uses computer modeling tools that are simulated and approved before code generation. A product's

deployment from the early concept of design to the final validation and verification testing is covered by this method, which includes numerous disciplines, functional behavior, and cost/performance optimization.

In this scope, we will provide hereafter a high-level description of the diagrams adopted in the model, as coming from SysML [56]. We provide here the main definitions.

Systems Modeling Language (SysML): SysML is a general-purpose system architecture modeling language for Systems Engineering applications.

Block Definition Diagram

A Block Definition Diagram is a static structural diagram that shows system components, their contents (Properties, Behaviors, Constraints), Interfaces, and relationships.

The purpose of Block Definition Diagrams is to specify system static structures that be used for Control Objects, Data Objects, and Interface Objects. When properly applied Block diagrams are recursively scalable and mathematically (parametrically) simulatable.

Internal Block Diagram

An Internal Block Diagram is a static structural diagram owned by a particular Block that shows its encapsulated structural contents: Parts, Properties, Connectors, Ports, and Interfaces. Stated otherwise, an IBD is a "white-box" perspective of an encapsuated ("black-box") Block.

The purpose of Internal Block Diagrams (IBDs) is to show the encapsulated structural contents of Blocks so that they can be recursively decomposed and "wired" using Interface Based Design techniques. When used correctly BDDs + IBDs are recursively scalable and mathematically (parametrically) simulatable.

Activity Diagram

An Activity (notation: rounded-rectangle or "roundangle") represents a flow of functional behaviors that may include optional Object (data) Flows. Control and

Object Flows can be sequential (default) or parallel (indicated by Fork & Join Nodes) depending upon conditions.

Sequence Diagram

A Sequence diagram is a dynamic behavioral diagram that shows interactions (collaborations) among distributed objects or services via sequences of messages exchanged, along with corresponding (optional) events.

Documentation

The documentation adopted for the elements description is the solely System Requirements Document (SRD). A software requirements document (also known as software requirements specifications) is a document that describes the intended use-case, features, and challenges of a software application.

Just like for the definition of the above models, an SRD shall be issued for every new components development.

3.3 Framework

The present Section describes the main changes and tailoring done on the CCSDS Mission Operation standard in order to get to the REFA design.

The service framework constitutes the connective layer of the architecture on which any on-board component, as either a peripheral adapter or a complete SW application, is plugged-in. The service framework provides a simple, yet functional set of core services intended for self-standing components to interact with each other. The service definitions are directly derived from the Mission Operations CCSDS Blue Book specifications, heavily tailored for simplification purposes. Such tailoring was possible, i.e. convenient, due to the implementation "from scratch" done for this project. Implementation from scratch was motivated by several benefits:

- Tailor the service definitions by making them more strict in terms of functionalities segregation and at the same time more simple, by cutting-off some invasive features. The stricter segregation is meant to offer the application-level components a service API which makes the service framework logic transparent to the component, e.g. a feature like the Actions and Parameters registration to the COM archive is not present anymore. Such segregation also makes it possible to get rid of a central, invasive elements as the COM archive.
- Rely on a new data representation methodology, such as Protocol Buffers (protobuf, in short) and grpc for the Remote Procedure Calls (RPC) implementation; which led to:
- Flexibility when getting to the implementation language selection. The protobuf compiler takes advantage of a language-independent service definition which can then be "compiled" for generating the actual language-specific implementation. Python is the chosen language for this prototyping phase of the project.

The framework provides services with two main purposes. First-class services for performing operational activities are intended for the satisfaction of primary application-level component needs. An example of these services is the Action Service, used for submitting general instructions from component to component. A

second-class set of services, conceptually derived from the MO COM services, provide additional support functionalities to the main services, for further presentation and analysis on ground. Example of these are the Activity Tracking Service and the Alert Service (the latter of which is more similar to the MO Event Service in terms of use-cases, rather than to the actual MO Alert service, but the Alert service interface is more suitable as an API), used for services' operations acknowledgements and eventual alerts raise.

The data classification is the base from which organising any data exchange operation and, in our case, from which harmonising the implementation of an architecture which intends to include technologies different from each others, in terms of functionality, domain of competence and role. Objective of this section is to describe the classification applied to any data within the architecture, the methodology adopted for orchestrating their exchange, and the tools that make such networking possible.

3.3.1 Data Representation

The data representation review, leading to the data model adopted for the architecture, started from the data model adopted by the current CCSDS Mission Operations standard, and reported in Section 2.2.4.

The main differences, with respect to the COM, consist in three main points, hereafter reported:

- There are no objects, but packet instances.
- There is no data-instance identification, as its identification is left to the components on top of the framework.
- packet Instances are only identified by the information in the packet header.

The first main change consists in the absence of data duplication for traceability purposes. The data who transit the framework, from one component to another, exist as only instance of its identity. E.g., a parameter exchanged in the service framework appears in the service data structure as a name and a value, with no reference to any definition or persistence. The duty of maintaining and being aware of the parameter

identity, possible value ranges and respective usage, remains under the component's responsibility. In other words, the consumer must know how to correctly interface with the provider in terms of data semantics, while the framework provides the data structure consistency and interaction capabilities.

The second change introduced in the architecture consists in the absence of any distinction between two different instances of a same service operation, in terms of data model. In other words, for correctly exploiting a service, the component is only required to provide a data instance whose type matches the service API, with no reference to the identity owned by such data, which remains a concern of the component's internal logic. The use of such data remains transparent to the service framework which keeps being unaware of what the parameter definition is, and whether the specific request exploiting the service is consistent or not. This implies that it is the component's responsibility to map the service API to the correct component API so to e.g., trigger the query of a parameter when receiving a getParameter service instance; and to correctly connect the lower-level functionalities of the architecture with the upper-level logic of the component.

The third change enables the framework to carry out the basic functionalities, namely the traceability and orchestration functionalities, by referencing the only headers of the packets used for exchanging service data.

One may argue that such choices tie many possibly distributed tasks to the component generating the data e.g., the statistics on data generated by a component to be implemented by the component itself, because the framework on its own does not provide such reference model representation which would abstract the provider away. This is only partially true. Let's make an example:

With reference to Figure 3.1, if a component (namely "Component A") requires statistics on data generated by another component (namely "Component B") via the statistics service (implemented by "Component C"), the statistics service references the data as modeled and stored in the framework environment, there available.

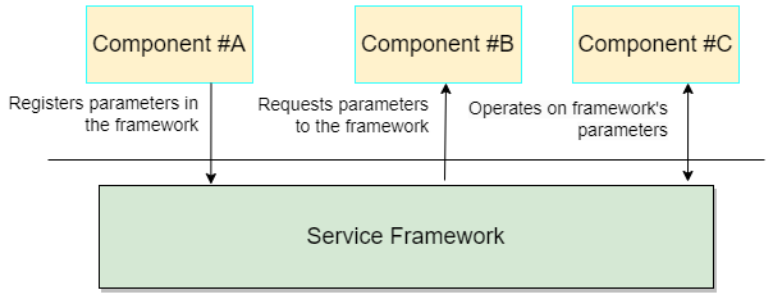

Fig. 3.1 If Data Model stored within the framework.

On the other hand, with reference to Figure 3.2, the same task requires a triangulation, for which *Component A* queries data from *Component B* and forwards the data, by means of a statistics service API, to *Component C* to get the statistics.

Such choice is oriented to the segregation of functionalities and allows the component to be fully responsible for executing a complex task (component-specific) without the need of involving the framework logic, which only serves data exchange purposes.

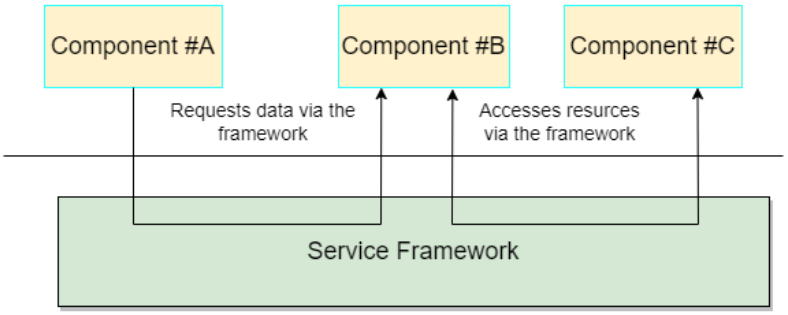

Fig. 3.2 If Data Model stored by the app-level components.

Such changes induce some implications, related to the traceability and storeability of data. In particular it requires a new methodology for trace and link the on-board activities to be analysed on ground and a common data model for any application-level task which requires multiple components to operate on the same data sets, which will be further described.

3.3.2 Data Classification

The classification of the data, as coming from the Mission Operations services, has a double purpose. As first it is meant to organise the data flowing into the framework, and as second it is meant to provide modularity to the packets building and addressing process.

The life-cycle of data instances within the framework follows a process of construction, exchange and persistence. This lifecycle is meant both to carry information from one point to another within the framework, an to track the activities for post-processing.

The first of these, is the message building process, meant to gather all the data needed for its distribution and interpretation, within the sessions of interaction between users and providers.

The architecture introduces the concept of "context", which uses for keeping information separated. The architecture therefore builds messages in three different contexts, which we can identify starting from the bottom as the "packets context", the "service context" and the "data model context". Such division reflect the layers constituting the framework architecture, and the separation in handling such different information allows making the architecture's reconfiguration modular.

Before any service or application is introduced, the framework defines a space domain in which to operate, constituted by a list of nodes (the components) and of links for connecting them (the network), which evolve along a time-domain. The packet data classification shall allow the identification, in the such time- and space-domain, of any link between any two components within the framework. Such classification works as the basis for further and more complex data handling orchestration.

The service definition adds a level of complexity to the interaction between the nodes and links introduced in the packet context description. On top of the link and nodes, the service definition defines for an hypothetical link between two nodes, the type of data exchanged via the link, and the interactions occurring between the nodes and relying on the link. The service definition provides the rules for formatting the data within the packet's payload, in order to semantic reflect the carried information. In addition the service definition describes the exchanging patterns fitted for the specific interaction between the two nodes.

One level up, we find the data models, which define application-specific rules in terms of data definition (e.g. in a clock application, the "Day", the "Hour", the "time reference"), data relations (e.g. conversion between UTC and CET) and the operations on such data (e.g. the creation or deletion of time references). Despite the architecture can include the definition of such data models, these are not part of the service framework, which is only in charge of orchestrating the data flow, and does not deal with the application-level activities. This implies that, beside the services definitions provided by the framework, any data model needs its own specification for its definitions, relations and operations.

The data classification just reported makes use of the message data structure for unambiguously identify the packet, the service instance and data model activity for which the specific message is issued. In particular, we will focus on these information contained in the message header, and we will show how they allow the unambiguous classification of the message.

The data organization and modularity objectives are met by a first data classification applied to the information being part of the MAL Abstrac Header, described by the MAL [43], so to divide such information into sub-groups and hierarchically organise them into a tree structure.

- At the bottom of the architecture, we define those data associated to the space- and time-domain, used for identifying any packet in the architecture. The values associated to each packet at this level can be deduced from the Table 3.2, by looking under the columns reporting a "Component" label. Such parameters can be dived between the static values defined at design-time i.e., the URIs and the Network Zone; and those who constantly update during operations i.e., the component's Time Stamp and the Transaction identifier. Such MAL header fields unambiguously identify any packet shared in the service framework during the framework lifetime.

- At the second level of the architecture we can find the services-related data. Each service includes one to multiple operations, which in the end define the proper data exchange logic, in terms of data structure and interaction patterns. Such parameters can be desumed from the same table under the columns reporting a "Service" label.

- On the top level of the architecture, there is a series of parameters which do not belong to the framework. Such parameters come from the internal logic of the data model and do not take part to the characterisation of the architecture layout. These are the Authentication ID, which is left to the component for an internal verification for access from a external user, the Domain, and the Priority Level of the delivered message.

The possibility to classify information, depending on their belonging domain i.e., component, service and operation, introduces a substantial reduction of efforts for managing data. By assigning the data grouped by domain to a different configuration element e.g., by using a different configuration file, it was then possible to structure the *data-assembling functions* to be dedicated to the specific data resource, resulting in an overall software modularity. As a result, it becomes possible to use the same assembling functions i.e., blocks of code, for building different services' packets. Every new element, introduced for composing new service structures, will refer to a self-standing data-structure and will implement its own assembling logic e.g., as either the data translation from incoming to outgoing packets, or the insertion of data from a configuration file to the relative packet structure portion. The software modularity topic will be the core of Section 4.1.1.

Table 3.2 MO Header Data Classification

MAL Message Header Fields	Framework Domain			Architecture Addressing			Application Domain	Traceability
	Service Descriptor	Operations Descriptor	Operation RunTime	Architecture Descriptor	Component Descriptor	Component RunTime	Application RunTime	
URI From					URI From			URI From
Auth. ID							Auth. ID	
URI To				URI To				URI To
TimeStamp						TimeStamp		TimeStamp
QoS level	QoS level							
Priority							Priority	
Domain					Domain			
Network Zone					Network Zone			Network Zone
Session	Session							
Session Name	Session Name							
Interaction Type		Interaction Type						Interaction Type
Interaction Stage			Inter. Stage					Interaction Stage
Transaction ID						Transaction ID		Transaction ID
Service Area	Service Area							Service Area
Service	Service							Service
Operation		Operation						Operation
Area Version	Area Version							Area Version
IsErrorMsg			IsErrMsg					IsErrMsg

3.3.3 Data Traceability

The Mission Operations makes use on objects for abstracting any information in the operational environment, and assigning an identification to them. This allows a global visibility on any on-board activity as if we were taking a snapshot of the data at any change. In the framework we implemented, the main difference, with respect to the Mission Operation services, consists in the Data Traceability methodology adopted. The data traceability within the devised framework relies on packets and on the data contained in the packets' headers. Now the on-board data exists on as an exchanged data. It does not exist before its exchange and persists after in the on-board memory only as a stored packet i.e., there is no identity associated to the data, but only data instances exist. The greatest impact of this major change, is that we get rid of the overhead constituted by the abstract representation adopted by the Mission Operations services, and relative rules the developers shall comply with.

Such choice is also made in order to confine the traceability issue to the service framework domain, with no extensions on -nor references to- the application-level data model adopted by the components. This modularity reflect then on the re-usability of the code, all across the framework.

With reference to Table 3.2 of Section 3.3.2, the last column is the results of the data classification analysis for traceability purposes. Traceability based on packets makes use of the packet header's fields needed for unambiguously identifying a packet in a time and space domain, and therefore, the data in the within. In our case the packet header is the MO MAL Header. From this, it is possible to select a sub-set of fields meant for data traceability purposes. Such selection is reported hereafter:

- URI From: indicated the URI where the packet was originated. It is needed in order to describe the path followed by the packet.
- URI To: same as for the URI From. It identifies the destination of the packet and contributes in the path reconstruction.
- Network Zone: every network zone consist of a set of unique URIs and the same URI can exist among different network zone. This field is needed for narrowing down the network of operations.
- Transaction ID: a component assigns a cyclic transaction ID value to every transaction it is performing. Since more transactions can occur between

two same components, the transaction ID is needed for distinguishing them. Nevertheless, the transaction ID is cyclic i.e., restarts from zero as it gets to its highest value, therefore an additional parameter is needed.

- Time Stamp: it is needed for distinguishing between interaction packets having the same Transaction ID, as repeating after a certain time span.

The information contained is this sub-set of MO header fields allows to unambiguously identify in a time and space domain any packet instance exchanged within the framework of services.

This is made possible because the *Network Zone*, the *URI From* and *URI To* enable identifying a unique "channel" between two nodes of the architecture. In the space/network domain, this triangle of information behaves like an identifier for the path of the information flow.

Beside this, within the same channel, the information flow contains as many packets as the two nodes' operations require. The *Time Stamp* on its own provides the capability to reference the packets in a time domain.

In addition, it might happen for the same application to issue multiple packets directed to the same node at the same time e.g., via different services, or because parallel processes share the same service interface. For solving this ambiguity, each application uses a so-called *Transaction ID*, working as a cyclical global counter, which increases by one every time the application issues a packet.

The three groups of parameters provided resolve the ambiguity issue concerning any packets exchanged between two nodes of the architecture and routed through the framework. It is worth noting that, up to this point, no data related to the structure or semantics of the packet's payload has been utilized. This implies that the traceability methodology has broad applicability across services. Regardless of the service used by the application, the same traceability methodology is adopted.

Furthermore, the traceability methodology remains symmetrical. This means that it is independent of both the service employed for data exchange and the data itself.

While we now possess all the necessary elements to uniquely identify any packet within the framework, we have yet to incorporate the required information for on-

board data handling and distribution. Consequently, it becomes possible to establish the service management logic on top of the packet logic.

3.3.4 Data Orchestration

The framework provides the data orchestration feature, along with data traceability, enabling the organization of data within the service layer for post-processing purposes. This facilitates the handling of operations performed within the system on the ground.

In the course of routine operations, the exchange of packets between two nodes adheres to a more intricate logic. The rules governing the coordination of information flow consistency within the system are defined by the services specification. Each service specification is tailored to serve a specific purpose, which is why the message header contains information specific to the respective service. Each information is assigned to a specific header field. The fields are:

- Interaction Type: Among the interaction patterns defined by the CCSDS Mission Operations specification, the Interaction Type identifies the specific combination of requests/responses adopted for the specific operation.
- Interaction Stage: Each interaction pattern consists of a finite sequence of requests and responses, and the combination of these elements can vary. The interaction stage indicates the specific step within the interaction life-cycle to which a message belongs.
- Service Area: A contextual grouping of services defines a Service Area.
- Area Version: As different versions of the same Service Area can be developed and deployed, this field ensure that both the nodes o the interaction i.e. consumer and provider, agree on the same service definition.
- Service: the service used for the data exchange. Each service is composed of multiple operations, sharing a common vocabulary of objects.
- Operation: the operation for such data exchange.
- Is Error Message: is a flag field which identifies the packet as containing an error message.

The data in the current section are meant to track any instance in the communication and in the service domain i.e., when generating the requests and response messages. Such fields are aggregated transparently as an additional data structure and appended to such packets/service for allowing referencing the request originating the activity. Here follows a more detailed description.

The service specification to be adopted in the Reference Architecture can then be inherited from the MO services [6] or PUS services [7], and the service tailoring will be treated in Chapter 4, for implementation purposes.

3.4 Data Models

The new structure just introduced, changes the traceability and the orchestration logic of the whole framework, but still, the service definition can be inherited from either the MO or PUS services. And this is the case, as the service specification adopted for the implementation (proof of concept) starts from the existing CCSDS Blue Book for the Monitor & Control services [6]. It is possible to further apply a simplification to the Mission Operations services definition. The result of such tailoring process, meant for the only proof of concept purpose, will be treated in Chapter 4.

On top of the presented logic, whose scope is mainly to create and keep track of an ordered flow of packets and services, a further layer is yet to be built: the Data Models. Such addition layer provides a methodology and rules for operating on mission-specific data. The data models family and their role into the devised architecture are yet to be defined, as specific to each mission. From common applications across different missions, it is nevertheless possible to shape a list of data models that provide basic yet critical functionalities.

In the present section the structure of a list of data models running on top of the framework is described. Objective of this Section is not only to provide a functional description of the on-board and on ground data models (which is provided by a dedicated SysML model, and the relative Python implementation), but rather understanding the role of the data models in the architecture and in the realization of *core system's functionalities*.

The purpose of data models is to provide the instruments for managing a multitude of application-related data in an organized manner. Data models, as treated in the present document, consist into:

- The set of *tools* for describing the data-sets of a specific application; which can be further decomposed into:
  - *Syntax* adopted for linking the data objects i.e., the properties defining the objects' relations; and
  - *Data Definitions* for providing an identity to the objects belonging to a data model and linked by using an agreed syntax.

- The *Rules* for operating on the such data as defined by the applications themselves i.e., relational conditions between the objects.

As an example we can refer to Figure 2.6 in Section 2.2.4. The MO COM tools involve the COM *syntax*, which consists into the *source* and *related* links between the objects, and the *data definitions* used for identifying objects into the MO framework i.e., the Type, the Object ID and the Object's Domain.

With respect to Figure 3.3, the tools are represented by the COM definition in the blue box within the common layer.

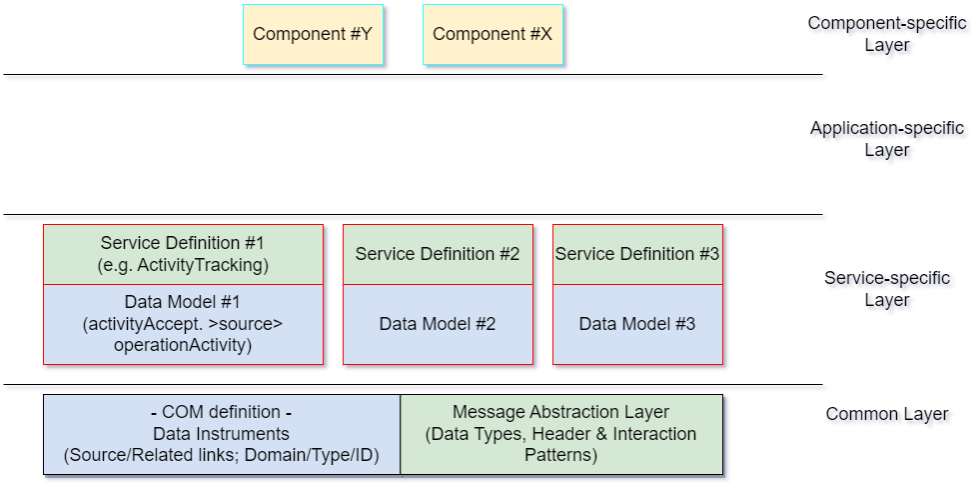

Fig. 3.3 Current MO Data Model.

In addition to the *tools*, every service in the MO framework provides the *Rules* for operating on the application-specific data accordingly to the provided *Data Definitions* and *syntax*. With respect to Figure 3.3, the service specification are represented by the different red-bordered boxes within the Service-Specific Layer. These include a data model specification which is generic and applied directly to the service definition, just like it is for the COM with the MO services. The same applies within the Common Layer, where the COM definition is kept together with the MAL definition.

New Framework Logic

More in general, it makes sense to develop a data model that provides ad-hoc instruments for describing the system of information typical of a particular activity e.g., the Mission Planning data model, while keeping it segregated from the services definitions used for accessing the application which operates on the data model.

A data model can e.g., operate both receiving inputs via commands and via files. Depending on which of the two inputs is chosen, the same data model will be accessed via either the *Action* service of the *File Transfer* service. Might the data model require e.g., a constant time update, this will most probably provided via the *Time* service.

An application operates on a data model in compliance to the data model definition and in compliance with the definitions of the services it uses for exchanging data. From the point of view of the architecture, Figure 3.4, given the services and the data models definitions, the application works as a junction between the two, in order to translate the incoming service request (compliant to the service definition) into the correct data model operation (compliant with the data model definition), and vice versa, sharing the data model output, via service packets.

Let's make an example. The *Mission Planning System (MPS)* generates the products i.e., command stacks, accordingly to the Mission Planning data model definition, in order for the commands to get to the on-board *Command Router*. The *MPS* implements the operations, which are defined by the data model, for generating the products, whose format is defined by the data model syntax and data definition. The *MCS* can now deliver the products (either on-bard or more probably to the MPS) via the File Transfer Service. Same as for the data model, the operations and data structure for transferring the files are defined by the File Transfer Service definition.

In order for the command to get delivered on board, the *Command Router* shall implement the File Transfer Service definition for command dispatching and the Mission Planning data model for un-packing the command stacks and perform the command checks. Ones the Command are on-board, the *Command Router* implements the Action Service definition for dispatching the commands to the on-board applications/peripherals.

It is now clear that different applications e.g. the MPS and the Command Router, implement the same data model i.e., the Mission Planning data model, for performing distinct actions on the data sharing an agreed syntax and data definitions.

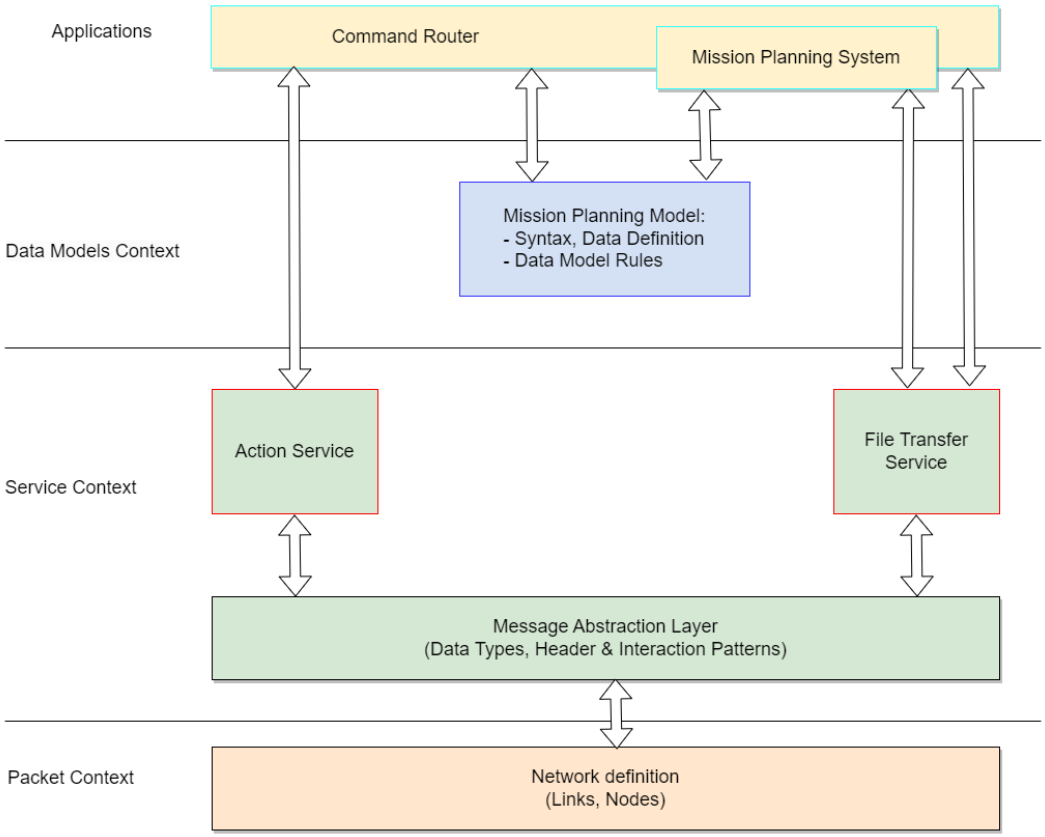

Fig. 3.4 REFA Data Model.

Regardless of the functionalities of the framework, guaranteed by the packet-based features treated in Section 3.3.3 and Section 3.3.4, it was possible to segregate the services API from the data model adopted by the above application. The data model is then adopted by the application which exploits the services for providing input and output data to and from the data model.

For several application to operate on the same data model, they shall share the same data model definition. One application is not limited to adopt one, but can freely operate on different data models depending on the activities it is dedicated to at run-time.

In this way it is possible to develop a mission planning system whose products structure i.e., the data model, are independent from the the service used for their transport. The mission planning components are based on the data model in use and on the mission. It makes sense to align multiple missions to the same MPS data model, and this can be done transparently to the framework of services.

It becomes possible to abandon the source/related links for defining any data relation as per the COM model and to develop more intuitive and fitting data models devoted to the specific application.

In the next section we present a list of data models which can be reused across different missions and which constitute a *core* i.e., the set of basic functionalities needed for operating a CubeSat, notwithstanding the possibility of extending such set, to a wider family of data models.

3.4.1 Command Router

The Command Router data model (CR in short) is conceptually derived from the *core Flight System* Command Ingestor component. Its role is to handle incoming commands generated on ground and distribute them on board, according to the command distribution logic of the data model run by the component. The command distribution distinguishes between two distribution methodologies:

- ASAP commands i.e., those commands which get delivered to the target as they are received and checked by the CR.
- Time-Tagged commands i.e., those commands that get released and dispatched to the target at a certain epoch in time.

The data model defines three parallel routines, respectively responsible for:

- Classifying incoming commands and forwarding them to the correct sub-routine.

  When a command stack is received from ground, the data structures containing the commands must be checked against inconsistencies, as well as the commands in the within. The command stacks are then handed over either by the ASAP queuing routine or TTQ routine depending on the checks' results.

- Processing the ASAP commands. When an ASAP command is received on board it is handled by this second routine, which queues the ASAP commands and dispatches them following a *First In - First Out* logic.

  There is no time-constraint in the dispatching of such commands, as this will solely rely on the available processing capabilities and peer-components response times, in order to get to the following command processing.

- Processing the Time-Tagged commands. Such dispatching routine is conceived for dispatching commands based on a time constraint. With respect to the ASAP dispatcher, that could dispatch a command only after the previous one terminated, the TTQ dispatcher is meant to handle every new command handling independently from the previous ones, so that they can be run in parallel and align to time constraints.

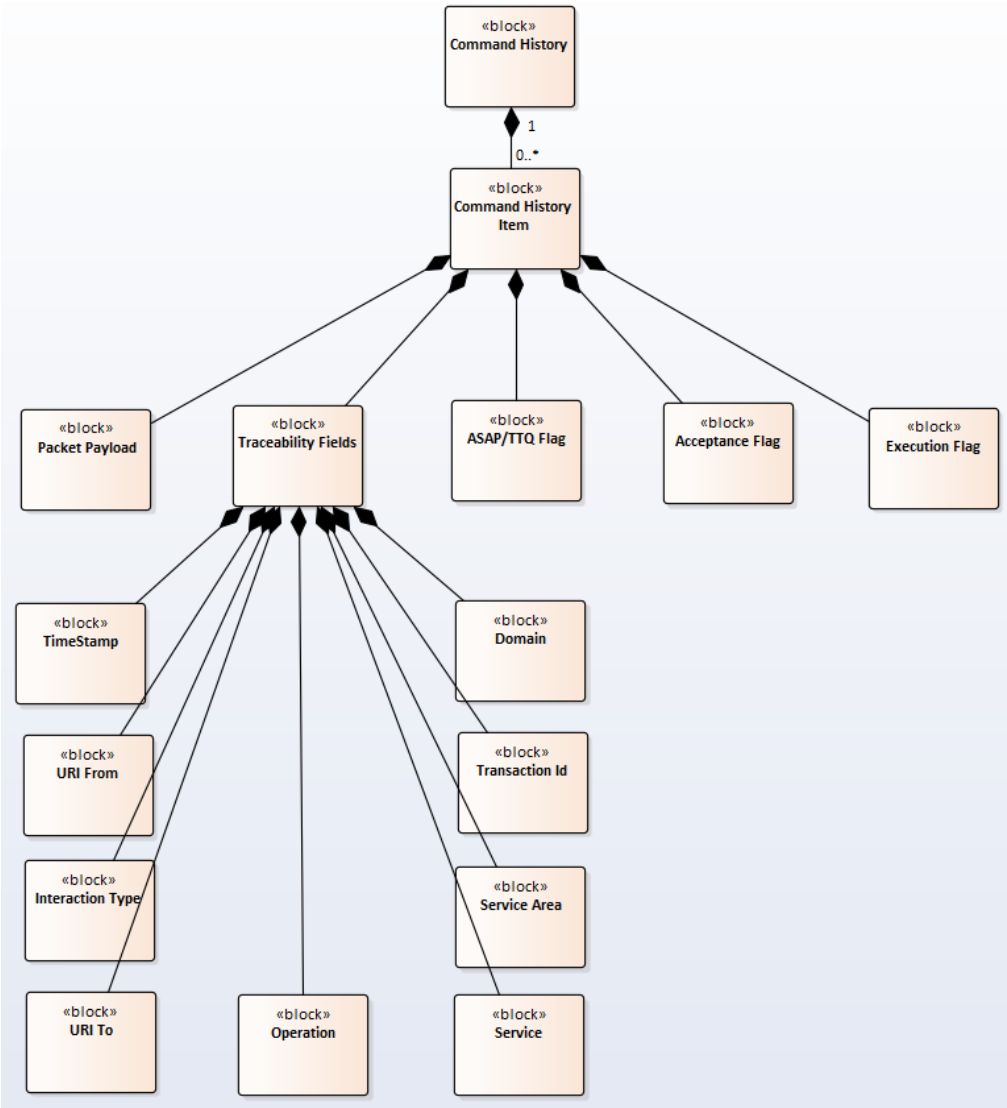

Fig. 3.5 Command Router Data Model.

The Command Router itself allows the distribution of commands coming from ground among the on-board subsystems as if such commands were generated directly on-board. The commands distribution on-board relies on the data classification described in Section 3.3.2.

When the CR receives a command directed to the CR itself, the first of the three processes receiving such command halts the listening and starts the CR-command execution. As its execution is complete, the process restarts listening for incoming commands.

Commands directed to the CR component itself can include: ASAP queue update, TTQ update, component shutdown.

The TO handles both file-based and packet-based telemetry.

3.4.2 Telemetry Output

The Telemetry Output data model component (TO in short) works in parallel to the CR and provides telemetry availability to ground as a single source point. Its routine involves the collection of telemetry from the on-board components, the consistency checks and the gathering of checked data for making them available for down-linking.

Together with the CR, the TO constitutes the other gateway used for routing data to ground. The TO data model is meant for handling to types of telemetry:

- Packet-based telemetry. Such telemetry includes all the packets generated on-board during operations. As the framework is responsible for keeping track of such packets, the TO data model is responsible for handling them for the downlink.
- file-based telemetry. Such telemetry is not generated by the exchange on on-board satabut rather is the collection of data as generated by the components themselves during operations e.g., the payload data.

The operations TO includes are hereafter reported:

- Collecting available telemetry. The operators on ground can explicitly specify which of the generated telemetry to retrieve, and in our case this can be done by exploiting the data classification and storage capabilities of the framework.
- building the packets for down-linking the gathered telemetry. Such telemetry shall be addressed to components on ground. The components relying on the TO data model on ground shall comply to this packetisation methodology for handling telemetry routed via the TO.
- Making the gathered telemetry available to ground. The operators on ground can actively specify which packetised telemetry to deliver to ground. This can be done exploiting the packetisation feature.

3.4.3 Autonomous Events Response

The Autonomous Events Response (AER in short) data model is a basic FDIR data model. It is meant to receive events subscription commands for those events requiring a reaction, as well as response action indications. Nevertheless, it is not just a matter of triggering an action in response to an incoming events. The event handling is more sophisticated than this and shall cover many peculiar casistics. In order to do this, the AER relies on an Event-Action table which gathers all the FDIR configurations needed for providing the correct response to any potential event. It is shaped as follows, and is reported in Figure 3.6:

- Event list, is the list of events the AER shall response in case any of this is notified (to the component implementing the AER data model). Every event part of the list is formally linked to two other lists.
- Threshold list. Some events can be part of the routine activities without threatening the status of the on-board system, until they get reported beyond a certain pase e.g., an Out of Limit (OOL in short) can occur in the transient stage of an operations up to a spare number of occurrences, without harming the system. This changes in the case the OOL is repeatedly triggered, indicating it is not related to a temporary condition but rather to a permanent contingency. The Threshold list is meant to gather such kind of i formation related to th event in order not to trigger an action if the event is not harming.
- Action list. An incoming event can require multiple actions to be taken in response to its triggering. The Action list contains the information for handling the action sequence in response to every event in the Event list.
- Parameter list. Every action may require parameters, whose value can be conditionally defined. The present list contains the conditions to be respected for assessing the correct parameter's value.

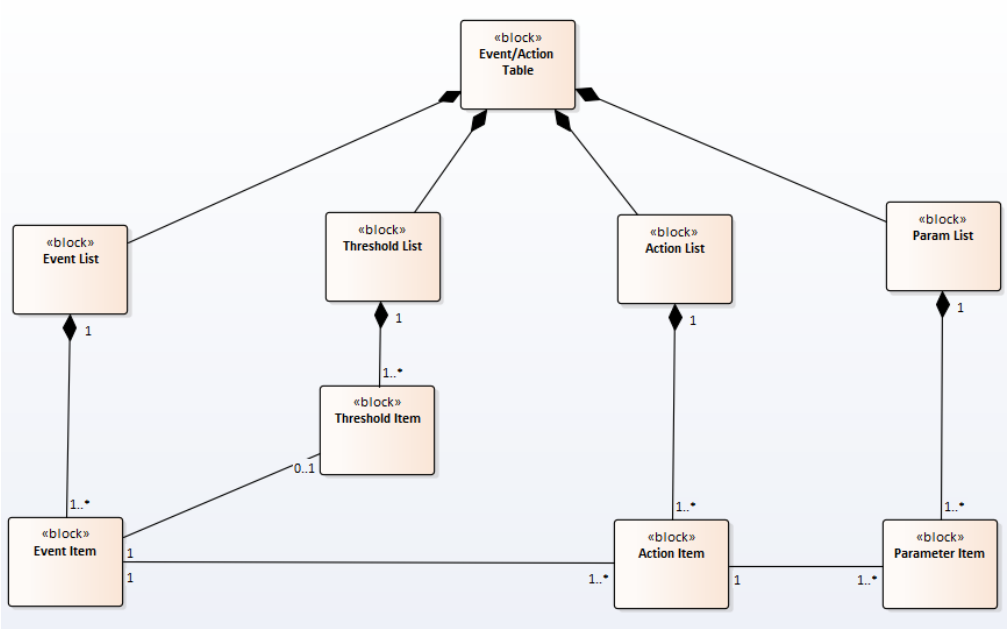

Fig. 3.6 FDIR Data Model.

3.4.4 Generic Component Model

The generic component model is not a data model *per sé*, but rather a collection of common features that can be re-used for the design of any additional, mission-specific data model. Its elements have already been mentioned and described above, that a data model shall own in order to be compliant with i.e., can run into, he reference architecture described in the present document. Such Generic Component Model includes the prescriptions regarding:

- The interaction definition occurring between the application-level component and the framework-level services, as depicted in Section 4.1.3.
- The Memory area segregation, as described in Section 4.2.1.
- The methodology for packet addressing, as described in Section 4.1.2.

3.4.5 Components

As mentioned above, in order to operate on the data model, the generic services' API shall be translated to the specific data model's implementation, including both the data model's API and functionalities. The component's are responsible for both the tasks i.e., collecting the data model's requests/responses so to translate them into a service-compliant format (and vice versa, for the incoming requests/responses), as well as implementing the operations needed for model's data handling i.e., computing requested operations, in compliance with the data model's prescription.

It becomes now clear that while the data model's definition can be inherited as it is from mission to mission, the components operating on such data models and run on top of the framework are implementation-specific as well as mission-specific. This becomes a key feature if we look at modularity and re-usability across different missions. While a Command Router data model is expected to be present in every mission i.e., every mission can adopt the same data model, each mission can either exploit existing component's implementation or re-design the components in compliance to mission-specific requirements e.g., in terms of execution responsiveness or prioritisation.

Chapter 4

Proof of Concept

In the present chapter we will go through the prototyping of the design elements, their implementation and validation of both the framework software elements and components

4.1 Prototyping

The current section focuses on the implementation aspects on the prototyped architecture that serves the proof of the architectural concept described in the previous chapters.

4.1.1 Functional Modularity

When getting to the implementation, the structure of the code takes advantage of the design features for the modularity property. The code structuring presented here constitutes the first validation step of the design, since it is a direct consequence of the choices made during the design. The code here presented is composed by three layers (namely from the lower to the upper: Definition layer, Constructor layer and Specification layer), operating in compliance with the data classification reported in Section 3.3.2. The framework here implemented is conceptually located between the application layer on top (described previously) and the transport on the bottom of the stack, further described.

The framework provides the upper application layer with an API enabling access to the lower services and transport logic almost transparently.

From a structural point of view, with respect to the architecture stigmatization described in Section 3.4, the three levels in which the code is structured - depicted in Figure 4.1 - can be grossly defined as implementing the orange and green blocks reported in Figure 3.4 i.e., the data-elements belonging to the service framework.

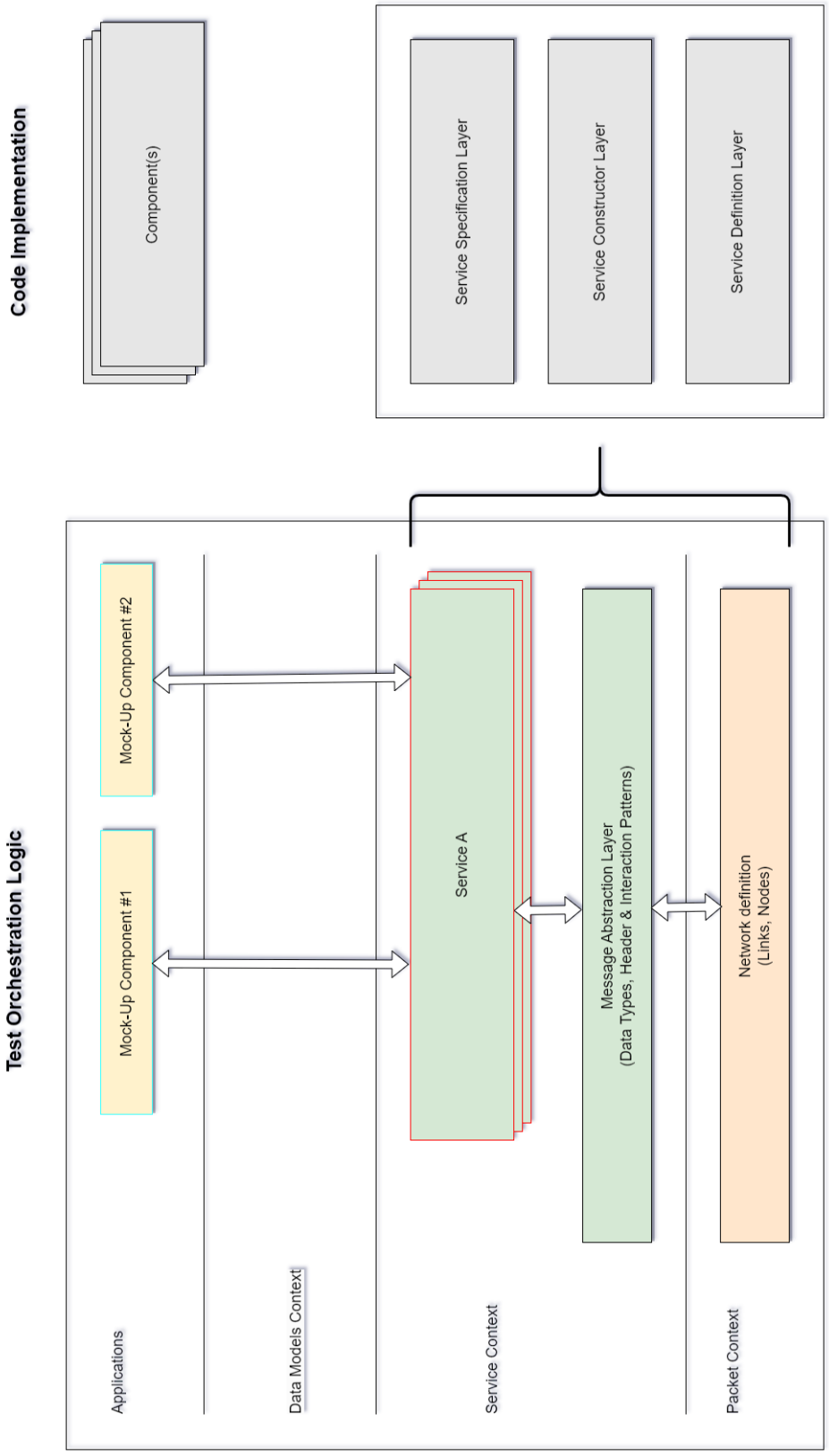

Fig. 4.1 Mirroring of the data management logic in the implementation

Within the framework functional subdivision, the Definition and Constructor portions of the code, provide functionalities and definitions inherited by the Specification. Such hierarchical sequence, following the organisation presented in Section 3.3.2, is here detailed.

The first layer of code from the bottom – the Definition layer - is intended to translate the document-oriented service specification into machine readable data structures and RPC definitions. Here the basic data types the framework uses are defined as well as, on top of these, more complex data structures, then exchanged via refined RPCs. An example of data structure definitions belonging to the Definition layer is the Packet Header definition and the service-specific payloads data structures. Such definitions make up the objects that the upper layers use for storing and moving information between components, i.e. the packets - their definition is part of the SysML model produced during the research activity.

Above the Definition layer, there is a second layer acting as a work-force for the framework – the Constructor layer. The Constructor layer implements the proper framework functionalities and hides much of the operations complexity by following static data-handling rules, which therefore enable automation. Each service, being either a first-class service or a support service, has it's own data structures definition and a defined set of rules for moving data among such data structures. Such automated operations reside in this layer.

An example of these is the header_constructor, whose responsibility is to receive data from the layers above (further described) and place that data into the header data structure, define by the lower Definition Layer, for the service requesting it.

A related example is represented by the ActivityTracking constructor. Its responsibility is to receive packet-related information from the layers above, e.g the Packet Header data structure, collecting the data required for tracking purposes, and moving such information accordingly to the ActivityTracking packet, for exchanging acknowledgments among components.

The third layer of the framework – the Specification layer - provides the MAL information characterising any on-board component, service and service's operation and a set of methods for retrieving such information. Such configuration data are hierarchically inherited (Component < Service < Operation). By doing this, the component instance has then access to the constructors for populating the data structure accordingly and the possibility of making use of specific services.

4.1.2 Component Addressing

The present section describes how the hierarchical classification provided in Section 3.3.2 enables API simplifications.

The inheritance logic makes the interfacing with the framework a lightweight task by reducing the amount of input the developer shall provide. In addition, the functions presented to the developer exploiting the framework, make the framework itself quite intuitive, thanks to the way such hierarchisation is presented.

As first, let's understand how the framework's configuration information is shared within the system. Within the system we find many segregated memory areas, either visible or hidden from each other. The main central visible memory area is handled by the framework itself, and includes the services' configuration information that every component will then inherit. Every component then shares a second public memory area where every component *introduces* itself to the rest of the architecture, by making available its component-configuration information i.e., who he is and where to contact him (URI), as well as which services exposes (service, area, operation). Such component-configuration information (enclosed in a component-configuration file) inherits the services' configuration files publicly visible, so that different component have access to the same services implementation, when exchanging data with each other.

In addition to this, every component hides a self-representation configuration-file which inherits the other components' configuration files, therefore inheriting all the information needed for reaching any component sharing such configuration information in the shared memory area, and thus making the communcation between segregated components possible. This *discover-ability* methodology oriented to the overhead reduction, requires any component's developer to provide a reduced amount of information for making the component reachable, and no additional configuration activities at run-time is required.

As mentioned above, the system presentation the framework provides, makes the use of such *libraries* quite intuitive. By inheriting the shared files, any third party component in the architecture is now available. The hierarchical structure adopted for such inheritance enables the user to access nested functionalities starting from the component (e.g., ComponentA, as per component's configuration), digging to the service the components wants to assess (e.g., Action service, as per frame-

work's configuration) and as last getting to the operation (e.g., submitAction, as per framework's configuration).

4.1.3 Autonomous Packet Delivery

Let's now describe from a functional point of view how the autonomous delivery property is achieved.

As a command is dispatched on board, the transport layer handles the routing, in order to reach the target component. The framework auto-generates the API code for dispatching/receiving the packet (Orange flag-shaped block on the left - "Received Packet" Figure 4.2). As the packet is received on the provider side, the information is saved into a Definition-compliant variable for the further handling steps. The simple reception process plays no role on the received packet's data handling. Here is where the app-level components jumps into the scene, helped by the Constructor layer.

The app-level component is required to perform internal compliance checks - "Acceptance Checks routine" - and, based on this, calling the Constructor layer for building the acceptance packet - "ActivityAcceptance_constructor - Figure 4.2", which can either confirm or reject the acceptance of the received packet, to the client.

For such ActivityAcceptance packet construction process, the component is required to provide the Constructor function with both the header of the received packet and the internal checks' results. The framework provides three hidden functionalities to the component:

- Traceability data structure building, then placed into the ActivityTracking payload of the packet. This allows the traceability between the incoming request and the acknowledgments in response - Traceability feature.
- ActivityTracking packet header building.
- On-board persistency of the generated packets, into a framework's memory area. This is meant to store the exchanged packets into a central memory area for further orchestration - Persistency feature.

For building the data structure dedicated to traceability, the Constructor function disassembles the header for extrapolating the only data required for traceability

purposes - see Section 3.3.3 - and places such data into a dedicated payload data structure, within the ActivityAcceptance packet, targeting the client. Such moving of traceability data preserves the addressing information i.e., URIs and Services, so to make the response process lightweight to the provider component. In this case, the adopted *grpc* make this process automatic, without relying on service-layer's functionalities.

For building the header of the ActivityTracking packet, the constructor function is provided with the header of the incoming packet. The majority of the fields is preserved, while some automatic data moving and update is done.

As the payload and the header are built, the packet is assembled into a single data structure compliant to the Definition layer's specification. This data data strucure i.e., the packet, is saved into a memory area dedicated to the packet's service i.e., the ActivityTracking service.

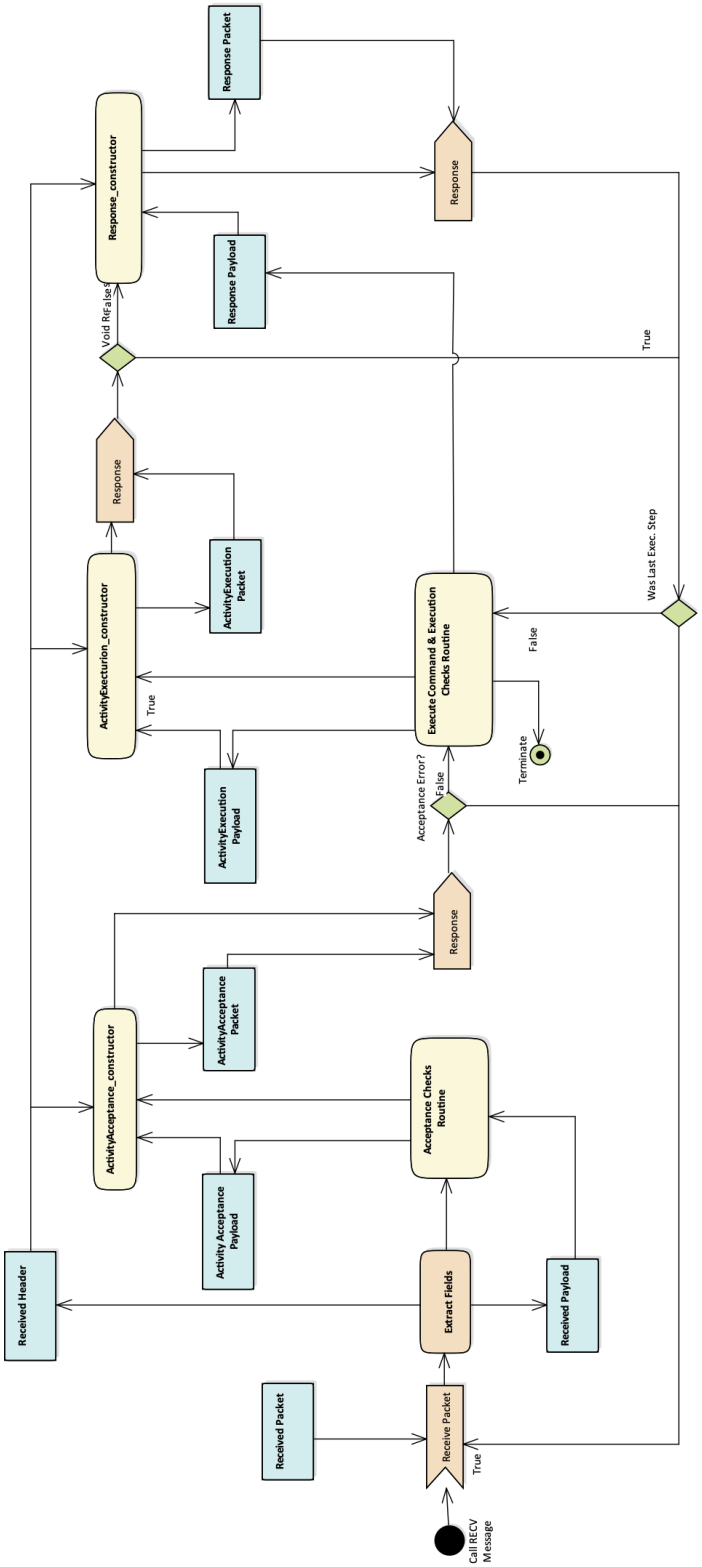

Fig. 4.2 Generic Provider interaction sequence - Activity Diagram

The process is very similar both for the next acknowledgment packet i.e., ActivityExecution, and for the eventual response packet i.e., in case the incoming request expects some mission data back. Such functionalities are identical and can therefore rely on the same portions of code, thus saving much of the effort needed for its development.

Let's now understand, how the on-board domains are managed by the framework, so to make it possible to orchestrate the depicted operations.

4.2 Deployment

The deployment Section here reported presents the key aspects related to the working environment, as compliant to the implementation criteria adopted. We will first go through the management of the network environment properties and the methodologies adopted for keeping the software components modular. Secondly, the Section focuses on the lower communication protocols on which the architecture relies, highlighting those aspects that enhance the architecture's time- and cost-saving properties i.e., the auto-generation of code.

4.2.1 Network Management and Resources Segregation

The present section describes the methodologies adopted within the architecture for providing the components with a global visibility among the architecture and the instrument adopted for arbitrarily expose and hide their respective resources to and from the third party components.

Network Management

From the networking point of view i.e., the network- and transport-level protocols adoption, the architecture is meant to exploit as much as possible of the existing technologies. The decision of adopting the Transport Control Protocol over Internet Protocol (TCP/IP) for the communication among components, was motivated by the opportunity of relying on decades of proven traffic orchestration within a distributed architectures, such as the World Wide Web. The choice has pros and cons.

Among the pros, we find a double outcome. As first, the developer is not required to have additional know-how. Any Computer Science engineer has a wide familiarity with network management.

As second, such protocols enable the adoption of common software elements used for the networking, hereafter reported, and whose usage will be treated in further sections:

- Virtual Private Networking (VPN)
- Network segregation via access gateways i.e., routers

- Possibility to secure communication between remote ten works i.e., via Secure Shell (SSH)

In addition to this, the proved reliability of such protocols and their worldwide adoption allows the deployment process of the architecture over any distributed network remaining almost transparent to the developer. As a consequence, such choice has a direct impact on the architecture structuring.

Talking about the cons, fixing the network/transport protocols of the architecture may constitute an excessive prescription, inducing a limit to further developments and tailoring processes. While this is true from the design-specific point of view, it is worth bearing in mind that the architecture here described aims at being a first prototype, a proof of concept. For such reason, certain choices, that in the beginning design stages could be left open for any further specific application need, now require to be fixed in order to identify, among these, the technology that of our architecture a *Ready to use* system. Moreover, despite the tailoring process done on the framework, described in the next sections, and despite the focus on the network transport technologies, the portability over different protocols i.e., the protocol mapping, is not further prevented.

Said that, we focused the research on certain properties that the network shall provide to the architecture, and that were fixed during the second design iteration of the architecture networking.

- Every software component belonging to the architecture, being either one of the core components or a tailored i.e., mission-specific component, is addressable via an IP address assigned to that component, in such a way that every component is potentially reachable by any node of the architecture.
- The adoption of a Virtual Private Network prevents the need for the software component belonging to the same network zone, to be running on the same physical machine or Local Area Network (LAN).
- On every component i.e., from any IP node of the architecture, a set of sockets for communication via services are exposed.
- To each service a specific TCP port range is reserved, so that the same service is made visible and available at the same TCP port(s) by any component providing the same kind of resource.

Given such properties, every component is now made available to any other component in the architecture.

Nevertheless, it is of primary importance for any component to segregate its resources in order to arbitrarily decide which to expose and which to keep hidden from the rest of the architecture.

Resources Segregation

The methodology adopted for wrapping the resources is the containerisation, which allow every component to be isolated from the rest of the environment and expose specific interface relying on the below network.

The adopted technology, [57] Docker, uses a client-server architecture and is widespread such applications, with multiple distributed elements shall interact by solely exposing standard interfaces. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon can run on the same system, or it is as well possible to connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets the user work with applications consisting of a set of containers.

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client (docker) is the primary way that many Docker users adopt for interacting with this powerful tool. When using commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

By the use of this tool, any user can design, integrate and deploy its own component over the architecture and gain full access to the services the architecture provides.

4.2.2 Communication Patterns

The defined communication patterns adopted, consist in two main methodologies:

- Peer-to-peer communication, by means of remote procedure calls, Figure 4.3.
- Data stream communication, by means of publish-subscribe interaction, Figure 4.4.

There is no prescription on what service should adopt which communication pattern, as this choice mainly resides in the service logic.

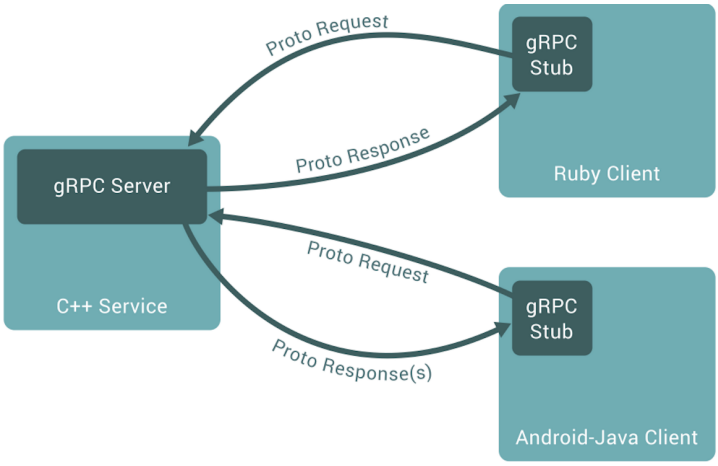

Fig. 4.3 Protocol Buffers/grpc - remote procedure calls

For an e.g.,Action service, the peer-to-peer communication pattern is more suited, because the dispatched action is intended for being interpreted and executed by a single entity i.e., the peer entity on the other node of the remote procedure call.

On the other hand, for the e.g., Alert service, the data stream communication pattern enables the multiple notification needed for a system-wide anomaly report.

From the implementation point of view, for the proof of concept of the proposed design, Protocol Bufers/grpc [58] and ZeroMQ [59] were adopted.

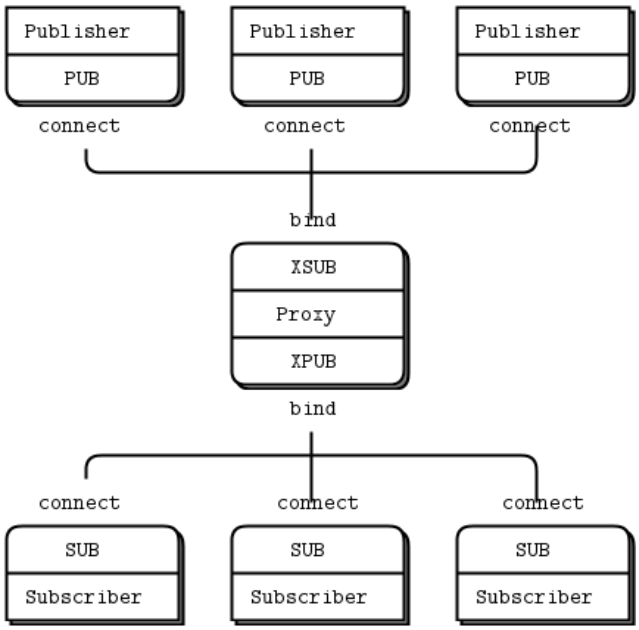

Fig. 4.4 ZeroMQ Publish-Subscribe interaction pattern

The protocol Buffers and grpc enable both a standard data structure definition whose content can be seamlessly translated from the service definition. At the same time, based on such definition, Protocol Buffers and grpc enable the auto-generation of large portion of code which can then be easily introduced in the implemented architecture. Beside the reduction of custom code by means of auto-generated code, the Protocol Buffers and grpc were chosen because they provide a standard API which reduces at most the adaptation elements needed for its adoption by the upper layer of the architecture. From an operational point of view, whichever the service, the interaction logic and implementation pattern is fixed, and any further design making use of them needs no customisation. This is observable from the e.g., generic provider interaction sequence described by the activity diagram reported in Figure 4.2.

The *Response* red elements in the activity diagram are implemented by the same code i.e., API and the only difference in the implementation is presented at application-level, when it comes to the selection of the packet content which respects the service-specific logic.

4.3 Validation

Given the size of the project and the workload required by implementing and validating the whole architecture, it was chosen to follow an approach that take advantage of the modularity property of the architecture. It was therefore decided select elements to be validated, regardless of the relations occurring between each other. The modularity features of the architecture allow validating every component's functionality as a self-standing element of the architecture, without the need of the adoption of any software-in-the-loop validation.

On such premises, the validation addresses the highest number of components starting by the key ones, i.e. the framework services, and the ones that offered a validation opportunity of wider scale i.e., the ground components working as a proof of concept in a parallel Politecnico's project [60].

The validation of the implementation of the architecture involved elements located in different layers of the architecture:

- Framework; the validation of the framework is intended to proof that the design properties imposed by design, described in Sections 3.3.1 to 3.3.4, guarantee the achievement of the modularity and quick deployment stated in the requirements definition.
- Ground Components; As a proof of concept, it was decided to adopt the design drivers defined by the REFA in the implementation of three ground components involved in the management and operation of a ground station project developed at Politecnico di Torino [60].

The present section focuses on the mentioned components and describes how the validation purposes were met.

4.3.1 Framework Validation

Thanks to the high level of modularity achieved in the design of the architecture, the framework validation will not involve any upper architectural layer.

The framework validation is related to three main types of software components, already mentioned in Section 4.1.1, being:

- Instantiation description provided by the Specification Layer. The validation here proofs that the definition files provide a consistent description of the component's layout. This means that the component's description in the framework domain is sufficient for the component to access the functionalities offered by the services.

- Information integration developed in compliance to the data classification of Section 3.3.2. Here the validation proofs that the modular functions implemented for moving information between portions of different packets i.e., for maintaining the packets traceability transparent to the component, cover the whole information range belonging to the MO header. As different information undergo different processing in the reception-response interaction, the validation proves that every processing works independently and acts on its own data-section of competence.

- Data translation from a data format specific to the REFA, mostly derived from the MO definition, to a Protocol Buffer data format, that can then be translated into remote prodedure calls. As the implementation requires the selection of an encoding convention and a fixed representation of the buffers structure, this this validation proofs that the REFA convention is correctly translated into ProtoBuf/GRPC convention, enabling auto-generation of big portion of code, therefore leaving the transfer concerns out of the REFA domain. Despite negligible, this feature ensures a huge reduction of effort required for the actual implementation and a programming-language-independent data convention, that unties component's language of implementation from each others.

Objective of the framework validation is to prove that the modularity and inheritance structure of the defined software elements both provide the components with a user-friendly operation-invocation syntax, that reflects the architecture layouts; and that the successful data traceability and persistency associated to any data exchange is transparent to the interacting components.

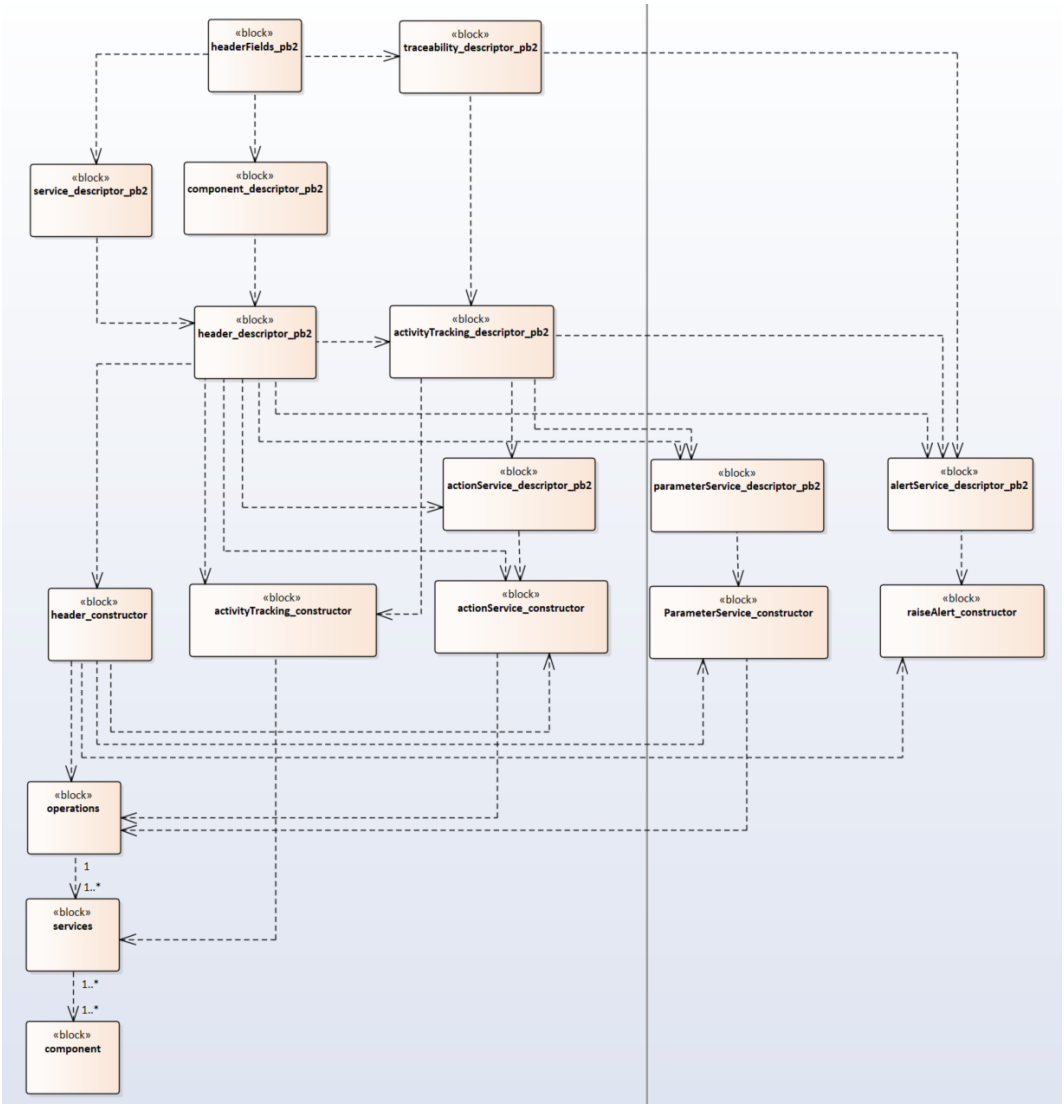

Fig. 4.5 Data hierachisation within the REFA.

As the validation of such properties do not depend on the exchanging components or on the underlying services, it was decided to implement a pair of mock-up components, working as the user and the provider of the exchanged data. In addition it was decided to adopt a generic service that as the Action Service, and to implement the submitAction operation.

Client.py

With reference to the Client code in Appendix B.

The component is required to import (line 1 to 4) generic libraries responsible for the transport of the message i.e. grpc and asycio. It is then required to import the service specification of the service he wants to use. Finally, he shall import the file descriptor of the component he wants to communicate with.

Lines 5 to 13 are the mock-up data structures used for this experiment, that shall match the payload data fields of the *submitAction* operation.

Lines 14 and 16 initialise the *grpc* pipeline. The initialisation of che communication requires the address i.e., IP address and TCP port, of the target component. Such info are extrapolated by the component descriptor file which provides a method for retrieving the information. This is indeed one of the purpose of the file descriptor that any component shares in the architecture.

In line 17 to 21 the submitAction packet to be delivered is built. Here we can see what already described in Section 3.3.3. The Client is only required to provide the data concerning the application level of the architecture i.e., from PriorityLevel to args. Any other information regarding the service level is hidden from the component and extrapolated from the component file descriptor, which provides information such as the services the Provider components provides. This is the second purpose of such file.

In this case the Provider provides the Action Service and therefore we can dispatch the message, in line 22.

Line 23 prints the response to the screen, as shown later int he current Section.

Lines 24 and 25 are part of the Python script checking routine.

Let's have a look at the other end of the channel, at the code of the Provider.py.

Provider.py

With reference to the Client code in Appendix B.

The Provider is required to import (lines 1 and 2) generic libraries responsible for the transport of the message i.e. grpc and asycio. It is then required to import

the service specification of the service he wants to provide (lines 3 to 5). It shall import its own file descriptor, where its information are stored (in line 6). Then it shall import the descriptors needed for the support services handling the traceability (line 7 and 8).

On line 10 the Provider starts the routine that waits for any incoming message, line 1 saves the header data in a variable and on line 12 the request is printed to screen, whose content will be shown later in the present Section. Then the component descriptor is instantiated as an object so to have access to its dependencies. Lines 14 and 15 are mock-up functions.

On line 16 the traceability packet is generated starting from the request-packet header. Such operation generates data structures that allow linking requests and acknowledgments together for ground operations. On line 17 to 19 the response is populated with the generate acknowledgment packet and then dipatched, on line 20. The same iter is run twice, from line 22 to line 27, because in the chosen example the data exchange involves a two-step process execution, requiring two distinct acknowledgment packets.

A similar process is done for sending back the response closing the execution. On line 28 to 32 the activityExectution packet is populated with mock-up data and finally dispatched on line 33.

Lines from 34 to 42 are part of the Python script checking routine.

Test

The validation consist in a test where the two mock-up components, namely the Provider and the Client, will exchange mock-up data. The test will be considered successfull if:

- The Client will receive from the provider the confirmation of the delivered command execution; and
- The traceability data linking between the requests and response packets are built transparently to the two moc-up components

In Appendix C the Json format of the request packet as received from the Provider side is printed.

The packet is divided into two main sections, the header and the body. The header's information has been extrapolated from the descriptor files describing the architecture and the packet could be successfully build and delivered via the correct service to the correct component.

In Appendix C the Json format of the Acceptance packet and of the two Execution packets delivered from the Provider to the Client is reported.

The packet is composed of three sections. The first is the header. This has been correctly handled by the framework during the response, by picking up the Client information at request arrival and using them for re-addressing the responses. This has been done for all the the acknowledgments.

The second section is the packet payload. As this is part of an ActivityExecution Service packet, the payload contains the traceability information of the incoming request. Such data are extrapolated from the request packet header by the *activity-Tracking_constructor* SW component and made available to the packet constructor before the packet is being delivered.

The third and last section of the packet is made of the application-level data, as provided by the mock-up Provider. Such data indicate whether the application-level tasks requested by the Client have been accepted and executed on the Provider execution area.

As can be seen from the four packets i.e., request, acceptance packet and two execution packets, the framework has handled those data in the request that shall be kept transparent to the application-level component. The modular elements of the framework have successfully translated both the addressing data of the incoming packets and the traceability data from the requests to the responses. On top of this, it was possible for the Provider to just run its internal (mock-up) processing without caring of the packets routing or third-parties components details.

4.3.2 Ground Components Validation

In the present chapter, we demonstrate how the conceptual architecture envisaged for the on-board components can be translated into the ground components with slight modifications, by relying on existing technologies. The implementation here

reported involves the definition of three ground software components devoted to the operations and management of a ground station facility.

Software Ground Components

Again, as described in the previous chapters, the system architecture is based on the concept of multiple modules connected via well-defined loose-coupled interfaces. Starting from software architecture, in computer science we can easily find a paradigm based on the same principles: micro-service architecture. This kind of architecture can be realized using multiple software solutions. Docker Container Engine, as presented in Section 4.2.1, is a Platform as a Service (PaaS) open-source tool designed to manage and create containers. Containers are a form of light-virtualization based on namespace features provided by a Linux kernel. Each container is an isolated environment, with its own persistence (file system) and network stack, meaning that multiple instances of the same "image" (container root file system) can run at the same time without interacting with each other. These features make it possible to decompose a monolithic application into multiple simple components that can be packaged as containers, and then being orchestrated as atomic entities, each one with its specific multiplicity and execution environment [61].

The decomposition pattern is the base of our proposal, as any component is packaged and deployed as a container, connected to the others via network sockets. In the current implementation, there are three principal subsystems:

- Secure Remote Access Gateway
- Digital Signal Process Service
- Antenna Control Service

The first one consists of a Virtual Private Network (VPN) access gateway with a Public Key Infrastructure (PKI) authentication based on the OpenVPN software. This represents the main entrance point for the GS operators, as all communications and interactions with the station subsystems are managed in a controlled and secure way. The PKI infrastructure is self-hosted, and it uses EJBCA software. Our service model is based on the following pattern: a X.509 certificate is issued to every user,

and represents the access token to the internal network, on which all services are exposed. Other subsystems (that will be referred to as "services" later on) are organized in "stacks", which are sets of containers connected via internal networks.

The second subsystem, the Antenna Control Service, is composed of three containers: the main one hosts an instance of GPredict [62], which is responsible for calculating the satellite orbit and the relative position to the GS at a specified time. The position is then converted in azimuth and elevation parameters that are communicated to the antenna rotator via meta-commands. These are then transmitted to a "driver" container (HAMlib) that translates them into real commands to be sent to the antenna rotator controller: the container in this case acts as adaptation layer. The antenna rotator controller is a hardware device that controls multiple actuators mounted on the top of the pole on which the antenna is installed to orientate it. The operator interacts with the system (GPredict user interface) using a VNC remote desktop session.

The third subsystem, the Digital Signal Process Service, contains three main components: the front-end, the modem, and the payload handler. The front-end module is composed of multiple instances of GNURadio that are responsible for controlling the SDR devices, making them accessible via network socket. This adaptation layer allows to decouple the SDR from the software modem. One of the benefits of this approach is the segregation of the signal acquisition from the signal processing module, meaning that the software modem can be modelled as a functional block, abstracting it from the SDR hardware device and its handling. Another positive aspect is the fact that the software modem (which is a CPU bound workload) can be run on high-performance host that doesn't need to be located near the SDR or the antenna system. The software modem component is another GNURadio instance, which implements all the signal processing logic, to act as a mid-level gateway to communicate to the satellite. At the end of the pipeline, we can find the last component of the system which is the payload handler block. Payload handler block implementation won't be part of this description, as it needs to be designed specifically for the communication protocol used by the specified satellite. Anyway, our design allows us to simply change that module on the fly, just running and stopping the right container, which will be implementing the specific protocol used by the target satellite. The other modules are in fact common for all satellites and are so designed to be sufficiently generic to not represent a limit in the protocols that the GS can handle.

Hardware Ground Implementation

Together with the software solutions , the present chapter provides also an off-topic on the hardware solutions adopted for the realisation of such architecture, as an additional proof that the proposed architecture can be implemented in a real-world scenario.

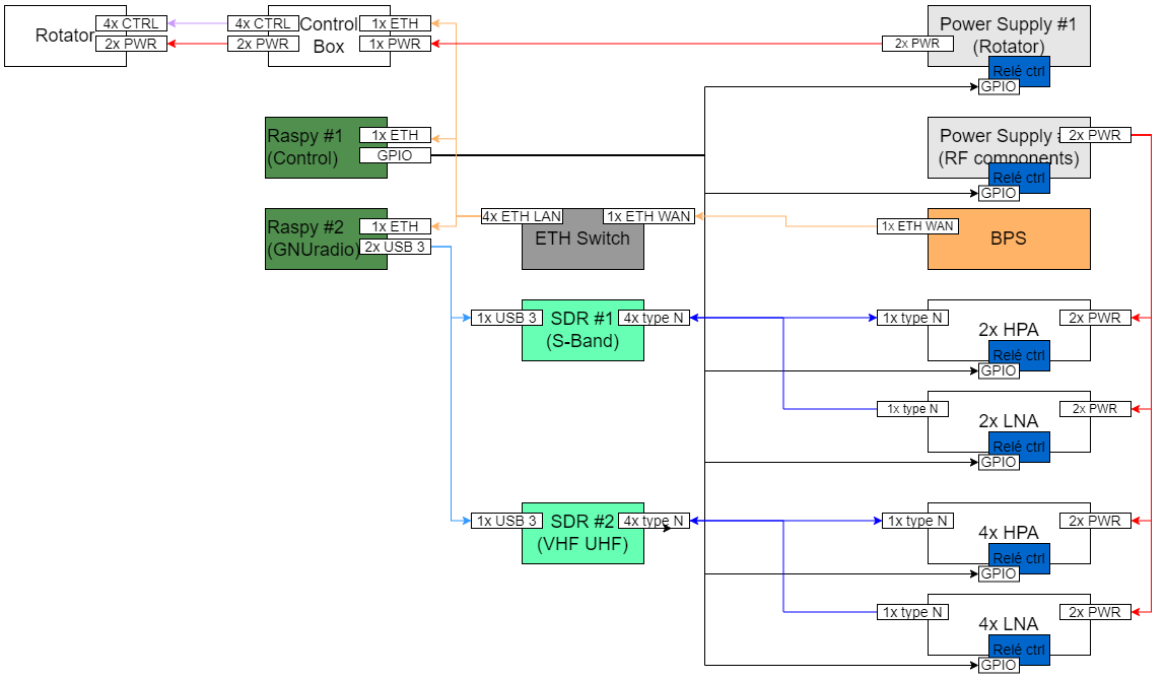

Fig. 4.6 GS hardware diagram block

Moving to hardware architecture, the system is composed of Commercial Off-the-Shelf (COTS) modules integrated together in Figure 4.6. This paradigm allowed us to concentrate on the software implementation, abstracting us from the details of hardware design and its specific concerns. As we stated before, our GS implementation is entirely based on SDR (software defined radio) for its communication system. Using this approach our GS is not locked to a specific protocol or modulation scheme and it can be adapted to any satellite, having SDR and RF pipeline operating bands as limit. We have three different RF pipelines: one for UHF band, one for VHF band and one for S band. UHF and VHF pipelines are composed of SDR (shared between pipelines), a LNA (low-noise amplifier - on rx chain) and a HPA (hi-power amplifier - on tx chain), a combiner/diplexer and the antenna system. The S band

pipeline has the same components, but with its own SDR and antenna. A dedicated RaspeberryPi is responsible to power-on the amplifiers when a specific chain is in use. Same Raspberry can also control the Power Supply of the Antenna Rotator. A python script exposes the status and control interface using a REST API.

Chapter 5

Conclusions and Next Steps

The current research has successfully showcased that by adhering to the designated design patterns, it becomes feasible to create the service framework and define data models for the same architecture, allowing them to progress independently and function cohesively.

Furthermore, this study has demonstrated a loosely connected group of elements forming a software architecture, while presenting an initial, yet efficient design for their integration and collaboration.

5.1 Conclusions

Considering the broad scope and ambition of this project, which encompasses the entire spectrum of a multi-domain software architecture, we have successfully presented an initial and valid example of what can be accomplished through exploration in this field. A significant achievement of this project is the collection and harmonization of a diverse set of architectural elements, which may not have been directly related but are often interconnected in a monolithic manner. As a result, the output of this work largely covers many of the initial objectives set for this project, offering a flexible and fully functional architecture.

- It has revealed insights into the opinions and trends within the small-satellite market concerning software architecture. This was accomplished by providing

a focused perspective on various technologies, along with their advantages and disadvantages.

- The work has delivered the design of key components essential for a software architecture tailored to small-satellites and CubeSats.
- Additionally, a simple yet reliable proof of concept was implemented to demonstrate the practicality of this design.

When examining the more technical aspects, we can evaluate the results and identify the areas that remain open for further exploration.

Throughout the study, different layers of the architecture were developed in parallel, with a specific focus on the key elements and tasks within each layer. An important takeaway from this approach is that the independent development of the framework and data models contributes to the architecture's flexibility, making it a viable starting point for new small-satellite and CubeSat missions. This approach eliminates the need to reinvent the wheel when it comes to software functionality for each new mission.

The framework development primarily emphasized data traceability and orchestration, making it applicable to any mission that requires data handling. On the other hand, the data models were designed to enable new missions to utilize a standardized set of application-level components, promoting consistency and ease of adoption across different projects.

Regarding Section 2.1, it is evident how the developed design successfully accomplished the proposed objectives. In this section, the design goals and objectives were outlined, and the subsequent implementation of the architecture demonstrated how these goals were met.

- The research successfully attained its goal of aligning system requirements with user needs through the utilization of a market survey. This survey provided valuable insights into the essential needs of recently established companies within the industry. This outcome represents a pivotal accomplishment in the research, as it extends beyond mere technical evaluations and serves as a fundamental reference point throughout the architecture design process.

  By conducting the market survey, the research team gained a comprehensive understanding of the primary requirements expressed by users in the

small-satellite industry. This user-centric approach played a crucial role in shaping the architecture, ensuring it meets the real-world demands of the target audience.

In summary, the incorporation of the market survey to inform system requirements is a key milestone that highlights the research's focus on meeting user needs and guiding the architecture design effectively.

- The goal of developing and implementing both on-board and on-ground software architecture based on existing technologies has been successfully accomplished from various perspectives. The distribution of applications between space and ground is easily defined within the data models domain, as outlined in the previous chapters.

  For instance, the mission planning data model comprises data created on the ground using dedicated ground applications. These data are then transmitted to the on-board system for orchestration. This particular data model, the Mission Planning data model, serves as a compelling example of how data can be distributed between the ground and space domains.

  Moreover, the ground and space components responsible for implementing such data models can seamlessly leverage the provided service framework for the space-to-ground interface.

  In summary, the achievement of this objective showcases a well-structured and efficient software architecture, incorporating both on-board and on-ground components while optimizing data distribution and leveraging the service framework for effective communication between space and ground elements.

The simple data models definition and design provided in Chapter 3,

The incorporation of modular self-contained components aligns perfectly with the driving design principles proposed, as evidenced by the formulated design model. The creation of the new software architecture was achieved through the adoption of Model-Based System Engineering (MBSE) methodologies, which signify the future of designing complex systems. This approach has the significant advantage of streamlining the development process and expediting the adoption and enhancement of similar systems in the future.

It is essential to highlight that MBSE greatly reduces the management effort required for many aspects of the design. However, it is important to note that while

MBSE is highly beneficial, it cannot completely replace the traditional document-based approach in certain aspects of the design process. Combining these approaches can lead to more comprehensive and efficient system development.

- On one hand, it offers the tools and terminology to represent the multitude of relationships that arise between elements within a system, particularly as the system becomes more complex. However, on the other hand, there are certain aspects where it may have limitations or challenges.
- MBSE establishes a structured framework for the system, but it may lack the intuitive understanding of design choices, presenting a significant limitation to the MBSE paradigm. Despite the modeling capabilities, this approach still necessitates the use of traditional documents to fully comprehend the innovative aspects of the project.

Flexibility plays a crucial role in our architecture, considering standardization as a fundamental requirement. This approach allows seamless interaction with various on-board peripherals without imposing rigid implementation choices.

In fact, the devised architecture emphasizes modularity and flexibility as its defining features, transforming the REFA into a valuable tool rather than a limiting constraint. Each architectural component was conceived as an independent element capable of serving the architecture's objectives. This achievement was made possible by leveraging the advancements of technologies already prevalent in the market, and described in the previous chapters.

The technology survey revealed promising opportunities for cross-fertilisation between technologies intended for different architectural areas. For instance, the MO Services integrated seamlessly with the SAVOIR-OSRA environment, just like the Protobuf/GRPC, which significantly reduces overhead related to code re-utilization efforts.

By placing the framework service layer in charge of service-specific domains, the system properties necessary for orchestrating the traffic have been preserved. This approach ensures that the information is accessible to the ground while also allowing room for expanding the service families to be incorporated into the model, as well as accommodating additional components to run on the service layer without obstacle.

Consequently, the architectural structure, which features a layered schema with components operating atop a shared service layer, has significantly improved modularity and segregation among its constituent elements. This design choice has not only simplified the adoption process but has also made the system scalable and manageable.

5.2 Next Steps

Despite the accomplishments made in recent years, the present work represents only the initial phase of what could be a potentially extensive process of collaboration and cross-contamination.

Although the project has made considerable efforts to address numerous areas, the focus on comprehensiveness has, in turn, affected the level of refinement achieved in certain areas.

The proposed Framework already provides a strong foundation on which further development can be built. By adding new services and expanding the range of functionalities, this process becomes nearly effortless due to the high level of modularity embraced in the design. However, it is important to note that a comprehensive family of Monitoring and Control (M&C) services still requires further refinement to align seamlessly with the architecture design drivers.

Further development in the framework domain shall start from existing MO services and porting them to the simpler and modular specification provided in the present document, up to complete at least the whole family of Monitor and control, services [6].

Indeed, while the developed data models show promise, they currently lack the necessary sophistication to handle the full orchestration of operational activities. This limitation is not due to neglecting certain areas but rather reflects the need for further refinement to effectively manage the diverse scenarios that may arise during operations. Enhancements are required to create a comprehensive solution capable of accommodating the complexities and variability inherent in operational scenarios.

The strict separation between the data models and the communicating components, a fundamental aspect of the REFA paradigm, should be preserved in the future development stages of the architecture.

In conclusion, the author expresses hope that the devised software architecture is just the initial step of a fruitful cooperative journey, contributing to the ongoing efforts in the challenging environment of small satellite software development.

References

[1] MC Working Group. *CCSDS 520.0-G-3, Mission Operations Services Concept*. Consultative Committee for Space Data Systems (CCSDS), December, 2010.

[2] ESA. https://essr.esa.int/project/osra-onboard-software-reference-architecture, November, 2021.

[3] D. H. Chang J. C. Jensen and E. A. Lee. A model-based design methodology for cyber-physical systems, 2011.

[4] NASA. https://cfs.gsfc.nasa.gov/, June, 2021.

[5] MC Working Group. *CCSDS 521.1-B-1, Mission Operations Common Object Model*. Consultative Committee for Space Data Systems (CCSDS), February, 2014.

[6] MC Working Group. *CCSDS 522.1-B-1, Mission Operations Monitor Control Services*. Consultative Committee for Space Data Systems (CCSDS), October, 2017.

[7] European Cooperation for Space Standardisation (ECSS). *ECSS-E-ST-70-41C - Telemetry and telecommand packet utilization*. Aprile, 2016.

[8] MC Working Group. *CCSDS 133.0-B-2, Space Packet Protocol*. Consultative Committee for Space Data Systems (CCSDS), June, 2020.

[9] MC Working Group. *CCSDS 520.1-M-1, Mission Operations Reference Model*. Consultative Committee for Space Data Systems (CCSDS), July, 2010.

[10] *CCSDS 850.0-G-2, Spacecraft Onboard Interface Services*.

[11] *CCSDS 910.3-G-3, Cross Support Concept — Part 1: Space Link Extension*.

[12] Cal Poly SLO The CubeSat Program. Cubesat design specification, rev 13. page 42, 2014.

[13] James R Wertz, David F Everett, and Jeffery J Puschell. *Space mission engineering: the new SMAD*. Microcosm Press, 2018.

[14] OHB Systems Bright Ascension, RHEA. *CCSDS MO Services, CCSDS SOIS, and SAVOIR for Future Spacecraft: Final Report*. Reserved Document, 2011.

[15] https://directory.eoportal.org/web/eoportal/satellite-missions/c-missions/CubeSat-launch-1 EOPORTAL. Cubesat - launch 1. Accessed: Sept-2020.

[16] C. Turner J. Puig-Suari and W. Ahlgren. Development of the standard cubesat deployer and a cubesat class
npicosatellite. In *IEEE Aerosp. Conf. Proc.(Cat. No.01TH8542), vol. 1*, pages 347–351, 2001.

[17] J. Bouwmeester and J. Guo. Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology. In *Acta Astronaut., vol. 67, no. 7*, pages 854–862, 2010.

[18] A. J. Ricco K. Woellert, P. Ehrenfreund and H. Hertzfeld. Cubesats: Cost-effective science and technology platforms for emerging and developing nations. In *Adv. Sp. Res., vol. 47, no. 4*, pages 663–684, 2011.

[19] Laura Niles. Largest flock of earth-imaging satellites launch into orbit from iss. In *http://www.spacedaily.com/reports/Largest$_F$lock$_o$f$_E$arth$_I$maging$_S$atellites$_L$aunch$_i$nto$_O$rbit$_F$rom$_I$SS Aug* – 2020.

[20] J. Bouwmeester B. Zandbergen E. Gill, P. Sundaramoorthy and R. Reinhard. Formation flying within a constellation of nano-satellites: The qb50 mission. In *Acta Astronaut., vol. 82, no. 1*, pages 110–117, 2013.

[21] D. Blaney R. Staehle and H. Hemmati. Interplanetary cubesats: Opening the solar system to a broad community at lower cost. In *CubeSat Dev. . . . , vol. 2, no. 1*, pages 1–30, 2011.

[22] F. Cabral S. Ilsen F. De Wispelaere K. Mellab B. Garcia Gutierrez M. Kuppers J. Naudet, A. Pellacani and I. Carnelli. Aim: A small satellite interplanetary mission. In *4S Symposium*, 2016.

[23] A. Klesh J. Schoolcraft and T. Werne. Marco: Interplanetary mission development on a cubesat. In *AIAA SpaceOps Conference*, pages 1–8, 2016.

[24] J. H. Saleh G. F. Dubos, J. Castet. Statistical reliability analysis of satellites by mass category: does spacecraft size matter? In *Acta Astronautica, vol. 67(1-2)*, pages 584–595, 2010.

[25] D. Verma R. Nilchiani E. Hole R. Cloutier, G. Muller and M. Bone. The concept of referece architectures. In *Wiley Interscience, Hoboken, New Jersey*, 2008.

[26] M. W. Maier and E. Rechtin. The art of systems architecting. In *Boca Raton, Florida: CRC Press*, 2009.

[27] Chantal Cappelletti, Simone Battistini, and Benjamin Malphrus. *Cubesat handbook: From mission design to operations*. Academic Press, 2020.

[28] John Bowen, Al Tsuda, John Abel, and Marco Villa. Cubesat proximity operations demonstration (cpod) mission update. In *2015 IEEE Aerospace Conference*, pages 1–8. IEEE, 2015.

[29] ESA. https://www.esa.int/esa/enabling support/space engineering technology/shaping the future, 2016.

[30] David McComas. Nasa/gsfc's flight software core flight system. In *Flight Software WorkshopFlight Workshop*, number GSFC. CPR. 7525.2013, 2012.

[31] Charles P Wildermann. cfe/cfs (core flight executive/core flight system). In *Flight Software Workshop 2008*, 2008.

[32] Alan Cudmore. Nasa/gsfc's flight software architecture: core flight executive and core flight system. In *NASA Flight Software Workshop*, 2008.

[33] David McComas, Susanne Strege, and Jonathan Wilmot. Core flight system (cfs) a low cost solution for smallsats. In *Annual Small Satellite Conference*, number GSFC-E-DAA-TN25822, 2015.

[34] David McComas. Nasa/gsfcs flight software core flight system community. In *Flight Software Workshop*, volume 12, 2014.

[35] Space Avionics Open Interface Architecture. https://savoir.estec.esa.int/savoirdocuments.htm, November, 2021.

[36] *SAVOIR-TN-002 Onboard Software Reference Architecture*.

[37] Jan Sommer, Raghuraj Tarikere Phaniraja Setty, Olaf Maibaum, Andrea Gerndt, and Daniel Lüdtke. Evaluation and development of the osra interaction layer for inter-component communication. In *2019 IEEE Aerospace Conference*, pages 1–12. IEEE, 2019.

[38] *SAVOIR-GS-005 Execution Platform Functional Specification*.

[39] *SAVOIR-TN-001 SAVOIR Functional Reference Architecture*.

[40] *SAVOIR-TN-003 SAVOIR Model Based Avionics Roadmap*.

[41] César Coelho, Mario Merri, Otto Koudelka, and Mehran Sarkarati. Ops-sat experiments' software management with the nanosat mo framework. In *AIAA SPACE 2016*, page 5301. 2016.

[42] MC Working Group. *CCSDS 524.2-B-1, Mission Operations–Message Abstraction Layer Binding to TCP/IP Transport and Split Binary Encoding*. Consultative Committee for Space Data Systems (CCSDS), November, 2017.

[43] MC Working Group. *CCSDS 521.0-B-2, Mission Operations Message Abstraction Layer*. Consultative Committee for Space Data Systems (CCSDS), March, 2013.

[44] Stefan Gärtner, Jens H Hartung, and Michael Wendler. Implementation of ccsds mission operations services at the german space operations center. In *SpaceOps 2014 Conference*, page 1869, 2014.

[45] Mario Merri, Bryan Melton, Serge Valera, and Andrew Parkes. The ecss packet utilization standard and its support tool. In *SpaceOps 2002 Conference*, page 06, 2002.

[46] M Schmidt. The ecss standard on space segment operability. In *Space OPS 2004 Conference*, page 504, 2004.

[47] Ignacio Clerigo, Andrea Accomazzo, Peter Collins, Nic Mardle, Elsa Montagnon, Jose-M Morales-Santiago, and Ignacio Tanco. One standard to rule them all: the tailoring of pus-c for future esa missions. In *2018 SpaceOps Conference*, page 2399, 2018.

[48] Andreas Jung. Introduction to mo services, sois and savoir harmonisation [moss] study, October, 2015.

[49] A. Wortmann A. Jung M. Sarkarati P. Mendham, S. Reid. Mo services, sois and savoir harmonisation: project status, 2015.

[50] OHB Systems Bright Ascension, RHEA. *CCSDS MO Services, CCSDS SOIS, and SAVOIR for Future Spacecraft: Consolidated Architecture*. Reserved Document, 2011.

[51] OHB Systems Bright Ascension, RHEA. *CCSDS MO Services, CCSDS SOIS, and SAVOIR for Future Spacecraft: User Needs and High-Level Requirements*. Reserved Document, 2011.

[52] OHB Systems Bright Ascension, RHEA. *CCSDS MO Services, CCSDS SOIS, and SAVOIR for Future Spacecraft: Prototype User Manual*. Reserved Document, 2011.

[53] OHB Systems Bright Ascension, RHEA. *CCSDS MO Services, CCSDS SOIS, and SAVOIR for Future Spacecraft: Prototype Detailed Design*. Reserved Document, 2011.

[54] Kubos. https://docs.kubos.com/1.2.0/apis/libcsp/index.html, 2017.

[55] GomSpace. Cubesat space protocol (csp), June, 2011.

[56] A. Moore M. Hause. The sysml modelling language, September 2006.

[57] Docker Inc. https://docs.docker.com/get-started/overview/, 2023.

[58] *Google LLC*. https://protobuf.dev/, 2022.

[59] *The ZeroMQ Authors*. https://zeromq.org/, 2022.

[60] A. Cornacchia R. Tuninato A. Arcieri F. Stesina S. Corpino L. M. Gagliardini, G. Maiolini Capez. An academic ground station as a service (gsaas) devoted to cubesats. ESA TTC 2022.

[61] A. Martin T. Combe and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, vol. 3, no. 5:54–62, October, 2016.

[62] http://gpredict.oz9aec.net/. Gpredict. Accessed: Oct-2020.

Appendix A

Questionnaire

Which is the main focus of your company/organisation in CubeSats domain?	Cubesat OBSW developer
	Cubesat OBSW developer
	Complete Cubesat project implementer
	Research, Education
	Downstream services provider
	Cubesat Operator
Is your company/organisation	Business/Revenue driven
	Public Organisation/Educational
Company Size	< 5 employees
	5-10 employees
	10-20 employees
	> 20 employees
Where is your company/organisation based?	<Open Answer>
Number of CubeSat projects already involved with	1
	< 5
	5 - 10
	> 10
Yearly company turn-over in CubeSats domain	<200KEuros
	200 - 500 KEuros
	500 - 1000 KEuros
	> 1000 KEuros
Are you already involved in one of ESA CubeSat projects?	No
	No, but we would like to
	Yes, as consultant
	Yes, as sub-contractor
	Yes, as prime contractor

Are you familiar with the following Standards	CCSDS Space Packet Protocol
	CCSDS Mission Operations Services
	CCSDS SOISCCSDS Mission Operations Services
	CCSDS SLE
	ECSS PUS
	SAVOIR-FAIRE - OBSW Reference Architecture
Have you used any of the above standards in your CubeSat projects	<Open Answer>
Do you apply 'a' / 'your own' Reference Architecture in your CubeSats projects	Yes / No
If yes, at what level	at communication protocol level
	at operational concept level
	at software interfaces level
	Reference Architecture at hardware interfaces level
Would a CubeSat REFA bring value and facilitate your business model	Yes, if it is at on-board communications protocol level
	Yes, if it is extended at device interfaces level in form of APIs
	Yes, if it encompasses also the composition of functional components on-board and on the ground
	Yes, if it is at hardware mechanical interfaces level
	Yes, help in research or in student recruitment or in academic purposes
	Yes, if it is at space to ground interface level

How far shall a CubeSat REFA go?	Market Place with competitive vendors offering compliant and competing implementations of elements of the REFA
	High Level Architectural Design - Paper Only
	Open-Source Reference Implementation of the core elements of the REFA (As reference not mandatory to take)
	Tools for auto-generation of code
How urgent is the introduction of a common REFA to your organisation?	Short term need (within next 2 years)
	Mid-term need (within next 5 years)
	Long term need (within next 10 years)
	Not required
Would you like to be directly involved in the specification of a REFA for CubeSats	Yes / No

Appendix B

Client & Provider

Client.py

```
1   import asyncio
2   import grpc
3   import configurations.constructors.pb2.actionService_descriptor_pb2_grpc as actionService_descriptor_pb2_grpc
4   from configurations.componentA_specification import ComponentA

5   componentA = ComponentA()
6   actionId = 150
7   argumentValues = [1, 3]
8   argumentIds = [2, 4]
9   args = [argumentValues, argumentIds]
11  priorityLevel = 1
12  transactionId = 200
13  session = 2

14  async def run() -> None:
15      async with grpc.aio.insecure_channel(componentA.get_actionService_address()) as actionService_channel:
16          actionService_stub = actionService_descriptor_pb2_grpc.ActionServiceStub(actionService_channel)

17          submitAction_packet = componentA.construct_submitAction_packet( priorityLevel,
18                                                                           transactionId,
19                                                                           session,
20                                                                           actionId,
21                                                                           args)

22          async for response in actionService_stub.submitAction(submitAction_packet):
23              print(response)

24  if __name__ == "__main__":
25      asyncio.run(run())
```

Provider.py

```
1   import asyncio
2   import grpc
3   from configurations.constructors.pb2.actionService_descriptor_pb2_grpc import ActionServiceServicer
4   from configurations.constructors.pb2.actionService_descriptor_pb2 import ActionInstanceDetails
5   from configurations.constructors.pb2.actionService_descriptor_pb2_grpc import add_ActionServiceServicer_to_server
6   from configurations.thiscomponent.thiscomponent import ThisComponent
7   from configurations.constructors.pb2.activityTracking_descriptor_pb2 import ActivityTracking_packet
8   import configurations.constructors.pb2.activityTracking_descriptor_pb2 as activityTracking_descriptor_pb2
9   class ActionService_server(ActionServiceServicer):

10      async def submitAction(self, action: ActionInstanceDetails, \\
                               context: grpc.aio.ServicerContext) -> ActivityTracking_packet:
11          action_header = action.header_descriptor
12          print(action)
13          self.__thisComponent = ThisComponent()

14          commandId = str(action.actionInstanceDetails.actionIdentity)
15          executionCount = 2

            #Perform any kind of checks and return packet acceptance

16          response = activityTracking_descriptor_pb2.ActivityTracking_packet()
17          response.CopyFrom(self.__thisComponent.get_activityAcceptance_packet(action_header,
18                                                                               commandId,
19                                                                               True))
20          yield response

21          #Perform any kind of execution checks
```

```
22                  response.CopyFrom(self.__thisComponent.get_activityExecution_packet(action_header,
23                                                                                      commandId,
24                                                                                      True,
25                                                                                      1,
26                                                                                      executionCount))
27                  yield response

                        #Go on with execution..

28                  response.CopyFrom(self.__thisComponent.get_activityExecution_packet(action_header,
29                                                                                      commandId,
30                                                                                      False,
31                                                                                      2,
32                                                                                      executionCount))
33                  yield response

34      async def serve() -> None:
35          server = grpc.aio.server()
36          add_ActionServiceServicer_to_server(ActionService_server(), server)
37          thisComponent = ThisComponent()
38          server.add_insecure_port(thisComponent.get_actionService_server_address())
39          await server.start()
40          await server.wait_for_termination()

41      if __name__ == "__main__":
42  asyncio.run(serve())
```

Appendix C

Framework Packets

Request Packet

```
header_descriptor {
  component_descriptor {
    URI_From {
      _URI_From: "127.0.0.1"
    }
    URI_To {
      _URI_To: "127.0.0.5"
    }
    domain {
      _domain: 1
    }
    networkZone {
      _networkZone: 1
    }
  }
  service_descriptor {
    service {
      _service: "5010"
    }
    qoslevel {
      _qoslevel: 1
    }
    serviceArea {
      _serviceArea: 1
    }
    areaVersion {
      _areaVersion: 1
    }
  }
  operation_descriptor {
```

```
    operation {
      _operation: 1
    }
    interactionType {
      _interactionType: "submit"
    }
  }
  session_descriptor {
    session {
      _session: 2
    }
    interactionStage {
    }
    isErrorMessage {
    }
  }
  realTime_descriptor {
    timeStamp {
      _timeStamp {
        seconds: 1675864879
        nanos: 315421000
      }
    }
    priorityLevel {
      _priorityLevel: 1
    }
    transactionId {
      _transactionId: 200
    }
  }
}
actionInstanceDetails {
  actionIdentity: 150
  argumentValues: 1
  argumentValues: 3
  argumentIds: 2
  argumentIds: 4
}
```

Acceptance Acknowledment Packet

```
commandAccetance_packet {
  header_descriptor {
    component_descriptor {
      URI_From {
        _URI_From: "127.0.0.5"
      }
      URI_To {
        _URI_To: "127.0.0.1"
      }
```

```
      domain {
        _domain: 1
      }
      networkZone {
        _networkZone: 1
      }
    }
    service_descriptor {
      service {
        _service: "5010"
      }
      qoslevel {
        _qoslevel: 1
      }
      serviceArea {
        _serviceArea: 1
      }
      areaVersion {
        _areaVersion: 1
      }
    }
    operation_descriptor {
      operation {
        _operation: 1
      }
      interactionType {
        _interactionType: "submit"
      }
    }
    session_descriptor {
      session {
        _session: 2
      }
      interactionStage {
      }
      isErrorMessage {
      }
    }
    realTime_descriptor {
      timeStamp {
        _timeStamp {
          seconds: 1675864879
          nanos: 317146000
        }
      }
      priorityLevel {
        _priorityLevel: 1
      }
      transactionId {
        _transactionId: 200
      }
    }
  }
  traceability_descriptor {
```

```
    service_traceability {
      service {
        _service: "5010"
      }
      serviceArea {
        _serviceArea: 1
      }
      areaVersion {
        _areaVersion: 1
      }
      operation {
        _operation: 1
      }
      interactionType {
        _interactionType: "submit"
      }
      interactionStage {
      }
    }
    instance_traceability {
      transactionId {
        _transactionId: 200
      }
      URI_From {
        _URI_From: "127.0.0.1"
      }
      URI_To {
        _URI_To: "127.0.0.5"
      }
      timeStamp {
        _timeStamp {
          seconds: 1675864879
          nanos: 315421000
        }
      }
      domain {
        _domain: 1
      }
    }
  }
  acceptancePayload {
    commandId: "150"
    commandSuccess: true
  }
}
```

Execution Acknowledment Packet #1

```
commandExecution_packet {
```

```
header_descriptor {
  component_descriptor {
    URI_From {
      _URI_From: "127.0.0.5"
    }
    URI_To {
      _URI_To: "127.0.0.1"
    }
    domain {
      _domain: 1
    }
    networkZone {
      _networkZone: 1
    }
  }
  service_descriptor {
    service {
      _service: "5010"
    }
    qoslevel {
      _qoslevel: 1
    }
    serviceArea {
      _serviceArea: 1
    }
    areaVersion {
      _areaVersion: 1
    }
  }
  operation_descriptor {
    operation {
      _operation: 1
    }
    interactionType {
      _interactionType: "submit"
    }
  }
  session_descriptor {
    session {
      _session: 2
    }
    interactionStage {
      _interactionStage: 1
    }
    isErrorMessage {
    }
  }
  realTime_descriptor {
    timeStamp {
      _timeStamp {
        seconds: 1675864879
        nanos: 317570000
      }
    }
```

```
      priorityLevel {
        _priorityLevel: 1
      }
      transactionId {
        _transactionId: 200
      }
    }
  }
  traceability_descriptor {
    service_traceability {
      service {
        _service: "5010"
      }
      serviceArea {
        _serviceArea: 1
      }
      areaVersion {
        _areaVersion: 1
      }
      operation {
        _operation: 1
      }
      interactionType {
        _interactionType: "submit"
      }
      interactionStage {
      }
    }
    instance_traceability {
      transactionId {
        _transactionId: 200
      }
      URI_From {
        _URI_From: "127.0.0.1"
      }
      URI_To {
        _URI_To: "127.0.0.5"
      }
      timeStamp {
        _timeStamp {
          seconds: 1675864879
          nanos: 315421000
        }
      }
      domain {
        _domain: 1
      }
    }
  }
  executionPayload {
    commandId: "150"
    commandSuccess: true
    executionStage: 1
    executionCount: 2
```

```
  }
}
```

Execution Acknowledment Packet #2

```
commandExecution_packet {
  header_descriptor {
    component_descriptor {
      URI_From {
        _URI_From: "127.0.0.5"
      }
      URI_To {
        _URI_To: "127.0.0.1"
      }
      domain {
        _domain: 1
      }
      networkZone {
        _networkZone: 1
      }
    }
    service_descriptor {
      service {
        _service: "5010"
      }
      qoslevel {
        _qoslevel: 1
      }
      serviceArea {
        _serviceArea: 1
      }
      areaVersion {
        _areaVersion: 1
      }
    }
    operation_descriptor {
      operation {
        _operation: 1
      }
      interactionType {
        _interactionType: "submit"
      }
    }
    session_descriptor {
      session {
        _session: 2
      }
      interactionStage {
        _interactionStage: 2
```

```
      }
      isErrorMessage {
        _isErrorMessage: true
      }
    }
    realTime_descriptor {
      timeStamp {
        _timeStamp {
          seconds: 1675864879
          nanos: 317812000
        }
      }
      priorityLevel {
        _priorityLevel: 1
      }
      transactionId {
        _transactionId: 200
      }
    }
  }
  traceability_descriptor {
    service_traceability {
      service {
        _service: "5010"
      }
      serviceArea {
        _serviceArea: 1
      }
      areaVersion {
        _areaVersion: 1
      }
      operation {
        _operation: 1
      }
      interactionType {
        _interactionType: "submit"
      }
      interactionStage {
      }
    }
    instance_traceability {
      transactionId {
        _transactionId: 200
      }
      URI_From {
        _URI_From: "127.0.0.1"
      }
      URI_To {
        _URI_To: "127.0.0.5"
      }
      timeStamp {
        _timeStamp {
          seconds: 1675864879
          nanos: 315421000
```

```
        }
      }
      domain {
        _domain: 1
      }
    }
  }
  executionPayload {
    commandId: "150"
    executionStage: 2
    executionCount: 2
  }
}
```