Rail to Digital Automated up to Autonomous Train Operation

D36.3 – Demonstrator implementation phase 2 concluded and test results consolidated

Due date of deliverable: 31/05/2025

Actual submission date: 31/05/2025

Leader/Responsible of this Deliverable: Kontron

Reviewed: Y

Document status		
Revision	Date	Description
0.1	27/08/2024	Initial Document Structure
	31/03/2025	Technical content provision deadline
0.2	14/04/2025	Internal review
0.3	29/04/2025	Final review for TMT submission
0.4	06/05/2025	TMT submission
0.5	01/10/2025	Corrected final version after JU review

Project funded from the European Union's Horizon Europe research and innovation programme		
Dissemination Level		
PU	Public	x
SEN	Sensitive – limited under the conditions of the Grant Agreement	

Start date: 01/12/2022 (WP36 Kick-off 25+26/01/2023) Duration: 42 months

ACKNOWLEDGEMENTS

This project has received funding from the Europe's Rail Joint Undertaking (ERJU) under the Grant Agreement no. 101102001. The JU receives support from the European Union's Horizon Europe research and innovation programme and the Europe's Rail JU members other than the Union.

REPORT CONTRIBUTORS

Name	Company	Details of Contribution
Manfred Taferner	Kontron	Functional Cluster FRMCS, Executive Summary, Introduction, Conclusion, editorial, review
Diallo Elhadj Thierno Sadou	Kontron	Functional Cluster FRMCS
Kevin Wriston	Kontron	Functional Cluster FRMCS
Ulrich Geier	Kontron	Functional Cluster FRMCS, review
Helene Farine	Kontron	Functional Cluster FRMCS, review
Thomas Martin	SBB	Functional Cluster FRMCS, review
Martin Lindenmaier	SBB	Physical Architecture
Thomas Buchmüller	SBB	review
Oliver Mayer-Buschmann	DB	Functional Cluster Diagnostics, Executive Summary, Introduction, Conclusion, editorial, review
Benjamin Labonté	DB	Functional Cluster Diagnostics, review
Andreas Messner	Hitachi/GTS	review
Stefan Resch	Hitachi/GTS	review

Disclaimer

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author's view – the Joint Undertaking is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

EXECUTIVE SUMMARY

This document is an integral part of the Rail to Digital Automated up to Autonomous Train Operation (R2DATO) project funded by the European Union's Horizon Europe - Research and innovation programme. This report presents the results of the second implementation and testing phase of the R2DATO Onboard Platform Demonstrator work package (WP)36. It builds upon the results of the first implementation report that set its focus on achievements validating the feasibility to showcase deployments for mixed-SIL (Safety Integrity Level) functions based on the Modular Platform concepts of R2DATO WP26. By doing this, key accomplishments now include the successful integration of two primary functional clusters into the Modular Platform implementation:

- The Functional Cluster FRMCS, which facilitates communication between onboard and trackside applications via the FRMCS network
- The Functional Cluster Diagnostics, which enables the collection and forwarding of onboard diagnostic data to trackside applying a standardised approach

The document provides comprehensive updates on the Modular Platform's architecture, detailing the design decisions and adaptations made to accommodate the integration of both functional clusters. It also describes the implemented functions and components necessary to fulfil selected user stories, including practical implementations and theoretical analyses.

The FRMCS cluster is tasked with establishing robust communication between onboard and trackside applications. This involves integrating FRMCS into the demonstrator and Modular Platform, notably through the implementation of an FRMCS Agent that supports the OB_{APP} interface for standardised communication with the onboard gateway. Additionally, a high availability architecture was developed to ensure continuous communication over FRMCS, which has been analysed and tested for failover scenarios to evaluate its reliability.

The Diagnostics cluster aims to enhance the collection and forwarding of diagnostic data from onboard applications and the Modular Platform itself to trackside systems, adhering to the Service-Oriented Vehicle Diagnostics (SOVD) standard. This involves implementing diagnostic tools and establishing a seamless data flow between the onboard and trackside environments, verified through automated DevOps deployments and testing.

The Tests and Results section offers insights into the strategies and methodologies employed to validate the user stories, with an emphasis on reproducibility and efficient deployment into the newly implemented and integrated physical laboratory setups. The conclusion highlights the significant progress made towards achieving a sophisticated onboard demonstrator setup and outlines the potential for future research, utilisation and development based on the findings.

Overall, this phase of the project demonstrates substantial advancements in the deployment and integration of FRMCS and diagnostics services onto the Modular Platform, paving the way for continued innovation as pivotal constituents of the project remits. This is enabling its utilisation for hosted applications within the project and future demonstrators

Underscoring the project's progress towards developing a sophisticated onboard demonstrator implementation, it outlines potential pathways for future research and product development, leveraging the findings to further innovation in autonomous train operations.

The collaborative efforts of project partners have been instrumental in achieving these milestones, reinforcing the commitment to advancing railway technologies and enhancing the safety, efficiency, and reliability of future automated train operations.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Definition
ATO	Automatic Train Operation
API	Application Programming Interface
BI	Basic Integrity
CAN	Control Area Network
CB	Communication Bridge
CCS	Command-Control and Signalling
CE	Computing Element
CG	Communication Gate
CGMS	Configuration and Group Management Server
CH	Control Handler (incl. FRMCS Agent)
CLIPS	C Language Integrated Production System
CM	Communication Manager (comm mgr)
CN	Computing Node (M out of N configuration of CEs)
COTS	Commercial Off-The-Shelf
CSCF	Call State Control Function
DIA-VEC	Vehicle Diagnostics
DTC	Diagnostic Trouble Codes
ECU	Electronic Control Unit
ETCS	European Train Control System
FFFIS	Form Fit Function Interface Specification
FRMCS	Future Railway Mobile Communication System
FRS	Functional Requirements Specification
GRE	Generic Routing Encapsulation
GTW	Gateway
HTTP	Hypertext Transfer Protocol
HSS	Home Subscriber Server
IAM/IdMS	Identity and Access Management / Identity Management System
I/O	Input / Output
IMS	IP Multimedia Subsystem
I-CSCF	Interrogating – Call State Control Function
IP	Internet Protocol

JWT	JSON Web Token
LMS	Location Management Server
MDCM	Monitoring, Diagnostics, Configuration, Maintenance
MCP-DIA	Modular Computing Platform Diagnostics
MCx/MCS	Mission Critical Services (MCPTT, MCData, …)
MCx-AS	Mission Critical Application Server
mDNS	Multicast DNS
OAuth	Open Authorization (protocol)
OB	Onboard
OB_{APP}	Onboard Application Interface
OB APP	Onboard Application
OCORA	Open CCS Onboard Reference Architecture
QEMU	Quick Emulator
P-CSCF	Proxy – Call State Control Function
RaSTA	Rail Safe Transport Application
REST	Representational State Transfer
RTE	Runtime Environment
SAL	Safe Application Logic
SIP	Session Initiation Protocol
SOVD	Service-Oriented Vehicle Diagnostics
S-CSCF	Serving – Call State Control Function
SDL	SOVD Definition Language"
SRS	System Requirements Specification
SW	Software
TAS Platform	Transportation Automation Systems Platform
TLS	Transport Layer Security
TRL	Technology Readiness Level
TS	Trackside
TS_{APP}	Trackside Application Interface
TS APP	Trackside Application
UDP	User Datagram Protocol
VM	Virtual Machine
VRRP	Virtual Router Redundancy Protocol

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

As introduced in the Executive Summary, his document represents the second of three implementation reports of the R2DATO work package WP36, proceeding with a step-by-step approach based on the Demonstrator Specification D36.1 [1]. One central objective of the Onboard Platform Demonstrator is to develop, integrate and validate onboard functions and services on top of the WP26's Modular Platform concepts and specifications. Demonstrating onboard applications within a laboratory setup aiming for Technical Readiness Levels (TRL) of 5 and 6 demands for sophisticated infrastructure, including test setups, components and prototypes realised and validated in relevant environments.

Similar objectives as for Deliverable D36.2 [5] apply: The report extends and refines the architecture and presents concrete results of the conducted implementation and testing phase, in this second implementation phase resulting in an higher TRL.

Whereas the Deliverable D36.2 was mainly dedicated demonstrating the feasibility of the core aspects of the Modular Platform, D36.3 focuses on its extension and usage for hosted applications. It includes a major shift from a virtual deployment environment towards a full-fledged physical laboratory realisation, incorporating all infrastructure needed implementing end-to-end applications and communicating from onboard to trackside.

Two functional clusters have been investigated, described, implemented and finally tested:

- The Functional Cluster FRMCS, dealing with communication of onboard and trackside applications via the FRMCS network
- The Functional Cluster Diagnostics, represented with the Modular Computing Platform Diagnostics (MCP-DIA), that is providing the ability to collect and forward onboard diagnostic data towards the trackside, based on the standard for Service-Oriented Vehicle Diagnostics SOVD [8]

The report is structured into the following sections: Architecture Update, Implemented and Studied Functions, Tests and Results and Conclusion.

In the section Architecture Update the Modular Platform architecture has been revised to detail the communication model facilitating as a blueprint for generic communication of any application hosted on the Modular Platform via the FRMCS communication layer. Additionally, the concept of Modular Platform Diagnostics (MCP-DIA) that has been introduced in D36.2 gets further elaborated on, deriving concrete design decisions and implementation proposals.

The section "Implemented and Studied Functions" provides detailed descriptions of all functions and components that have been implemented to fulfil selected user stories. This covers a FRMCS onboard communication gateway realisation on the Modular Platform.

The integration of the first MCP-DIA implementation marks a major corner stone in the provision of a harmonised diagnostics realisation on top of the Modular Platform.

Implementation details and specific adoptions of the proposed architecture get outlined in this chapter in detail.

The section Tests and Results outlines for each functional cluster the outcome of the applied tests and the analysis of the investigated user stories. It describes the test strategies and methods providing evidence generated by manually and automated performed test runs.

Finally, the Conclusion chapter summarizes the key outcomes, achievements, accomplishments and learnings.

2 ARCHITECTURE UPDATE

This chapter provides updates to the WP36 architecture, including detailed elaboration and design decisions for the further on discussed functional clusters. This chapter forms the basis for the realised implementations that get described in detail in the next chapter Implemented and Studied Functions, with the objective to fulfil selected user stories from D36.1 [2].

2.1 FUNCTIONAL CLUSTER FRMCS

The main concepts of the FRMCS architecture including the FRMCS onboard architecture as described in Chapter 3.6.4.1 “FRMCS Gate” of D36.1 Architecture [4], as well as in Chapter 2.2 of “D36.2 Demonstrator implementation phase 1 concluded, and test results consolidated” [5] are still valid. They are further elaborated in the following sub-chapters.

2.1.1 Communication Architecture

The TAS Platform employs a message-based communication model to facilitate interaction between tasks. Replica determinism is achieved through management of the sequence order and content of messages exchanged by replicated tasks, and the defined state of those tasks. Importantly, the platform does depend on replicated task execution on redundant computing elements but not on a strict synchronized operation in cycles to ensure deterministic execution among replicas.

Safe functions on the TAS platform are developed using two types of so called TaskSets: Model-1 TaskSets and Model-3 TaskSets. TaskSets refer to a logical grouping of Unix processes and threads that are managed and monitored by the TAS platform. In terms of the communication architecture, the primary distinction between these two models is that Model-1 TaskSets are restricted to utilizing the TAS Communication System for interactions with other tasks, specifically (but not limited to) through voted message queues. In contrast, Model-3 TaskSets have the flexibility to employ the complete range of POSIX communication functions provided by the operating system, including TCP/IP network communication sockets. Since such communication stacks are not feasible to maintain in a replica deterministic way, replication for safety is limited to Model-1 TaskSets while Model-3 TaskSets can only contain parts of the application that do not require replication for safety (e.g. Basic Integrity).

To align with the TAS communication paradigm, safe functions implemented on the platform must adhere to a specific high-level architecture when communicating with external systems. Such systems may either reside onboard and connect directly to the CCS Communication Network or be offboard and reachable via FRMCS.

As a result, safe functions consist of an architecture of Model-1 TaskSets providing the safe application logic (SAL) along with a combination of Model-1 and Model-3 TaskSets providing the infrastructure, i.e., Communication Gateway & FRMCS Agent, to communicate with external systems. Some of the communication infrastructure (e.g. the implementation of the safe communication protocol) must reside in a Model-1 TaskSet like the safe application logic.

For simplification, the proposed architecture assumes that each safe function incorporates its own communication infrastructure. This could be optimized by introducing a common communication infrastructure that could be shared by multiple safe functions. Currently at minimum a reuse based on separate copies of a library is implemented.

The below diagram depicts the communication architecture on a single computing element (CE). The full picture of a 2oo3 setup can be found further down in Figure 2.

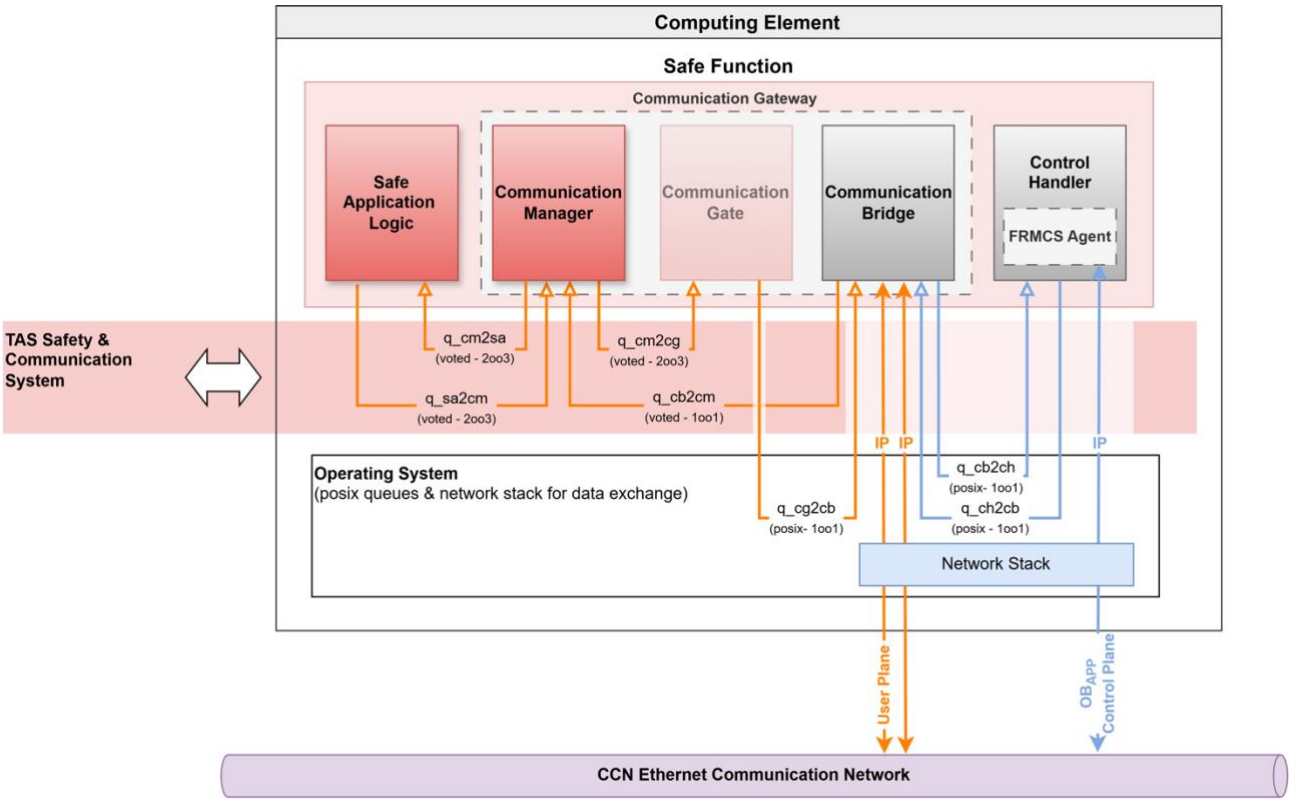


Figure 1: Communication architecture on a single CE

2.1.1.1 Safe Application Logic (SAL)

The SAL implements the safe function by receiving input messages through a voted message queue (q_cm2sa), performing its internal calculations and sending output messages via another voted message queue (q_sa2cm). For clarity, the application logic is represented as being executed within a single Model-1 TaskSet. However, in practice, a safe application logic would likely be developed using multiple TaskSets to enhance functionality and maintainability.

2.1.1.2 Communication Manager (CM)

The CM is responsible for managing the various communication sessions required by the safe application logic (SAL). This includes forwarding session control requests from the SAL, such as establishing and terminating sessions.

The CM implements the safety protocol for the respective communication sessions, overseeing their lifecycle through mechanisms like heartbeat messages, sequence numbers, timestamps, and other methods in accordance with CENELEC standards.

If the CM detects that an active communication session has failed (for example, due to missing heartbeat messages), it automatically initiates the appropriate recovery actions, including triggering the re-establishment of the session. The SAL may be notified of this event, but if no safety timeouts have been breached, the session can be re-established without requiring any safety response from the application.

The CM also determines which computing element is responsible for physical communication with the outside world, specifically which CE's Ethernet adapter is being utilized. If the CE responsible for physical communication fails, the CM automatically initiates the re-establishment of communication sessions for all active connections, utilizing one of the remaining computing elements. In case the communication sessions can be re-established without violating any safety timeouts, no safety response from the application is necessary.

2.1.1.3 Communication Gate (CG)

The CG is a generic helper task designed to facilitate the recovery of replicated Model-1 TaskSets that communicate with non-replicated Model-3 tasks via message queues. This functionality is a unique feature of the TAS platform and may not be required in other platform implementations. This TaskSet corresponds to the xfer TaskSet described in chapter 3.3.1 of D36 [5].

2.1.1.4 Communication Bridge (CB)

The CB serves as the link between the platform's internal message-queue-based communication and the CCN ethernet network, effectively managing the user plane connections with peer services.

The CB is responsible for managing the protocols at OSI layers 4 to 6 by establishing and maintaining TCP/IP connections with external services, whether they are located onboard (directly attached to the CCN) or offboard (reachable via FRMCS).

For offboard FRMCS connections, the CB utilizes the Control Handler (FRMCS Agent) to create the necessary IP tunnels and to obtain the corresponding IP addresses for the offboard services.

On one hand, the CB reads user plane messages from a queue (q_cg2bc) and forwards them via TCP/IP connections to the appropriate peer service. On the other hand, it receives messages from external services through TCP/IP connections and forwards them via a message queue (q_cb2cm) to the receiving TaskSet.

2.1.1.5 Control Handler (CH)

The Control Handler (CH) manages the control plane data exchange with the FRMCS Gateway (TOBA Box) through the OB_{APP} interface – as such the Control Handler implements also the FRMCS Agent function in case of FRMCS communication as lower transport layer. It receives control requests from the Communication Bridge via the POSIX message queue q_cb2ch and is responsible for establishing and managing the IP tunnels necessary for user data sessions between the onboard Safe Application and the corresponding offboard services. Once a tunnel is established, the CH sends the IP address to be used for communication with the specified peer service back to the CB via the POSIX message queue q_ch2cb. More details about the FRMCS Agent are then added in Section 2.1.1.6.

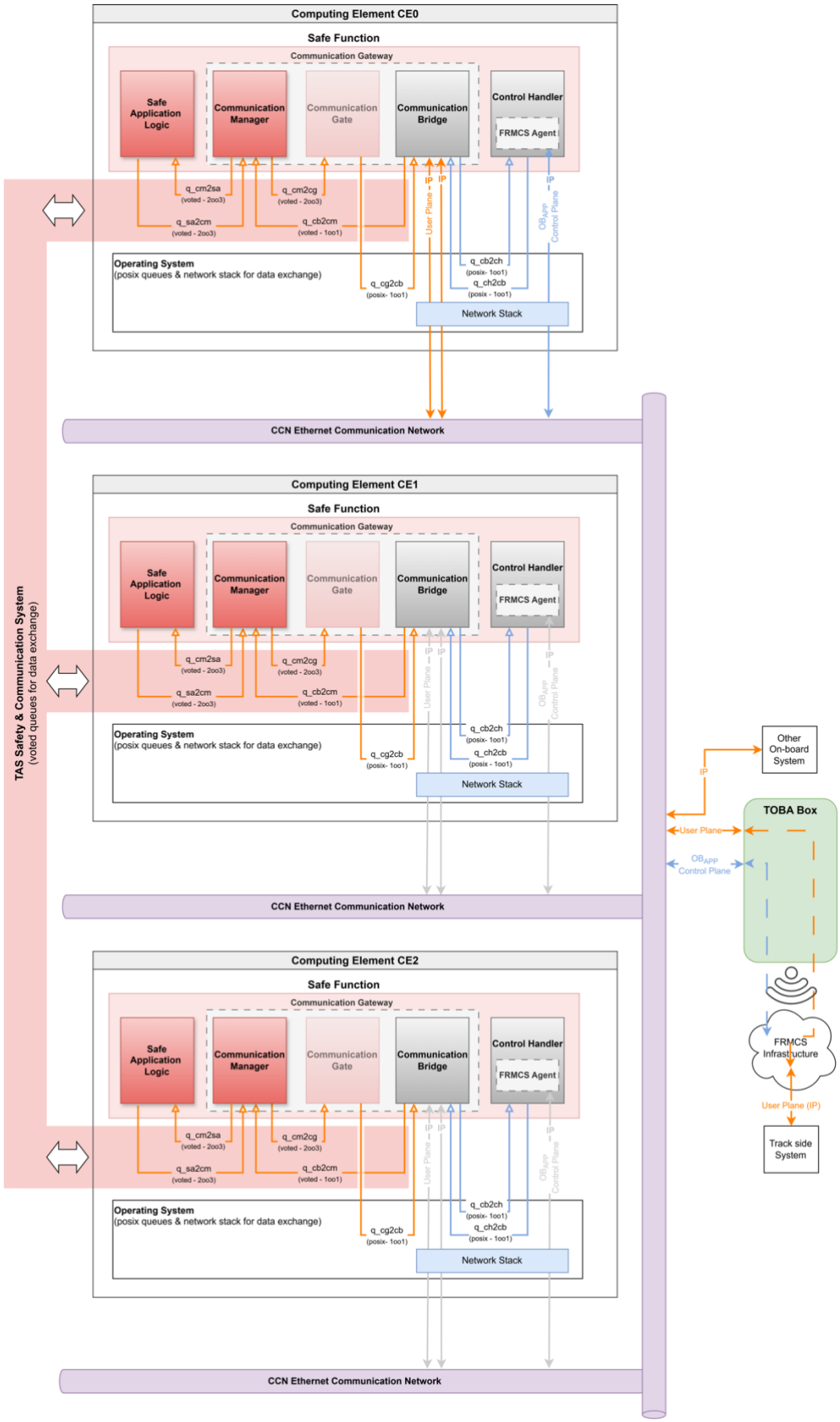


Figure 2: Communication architecture in a 2oo3 configuration

2.1.1.6 FRMCS Agent

The FRMCS Agent is the component which communicates via the OB_{APP} interface to the Onboard FRMCS TOBA and towards the Application via the communication pattern described in the previous chapter.

The FRMCS Agent is implemented as a library used by the Control Handler.

It receives commands via the Control Handler from the Communication Manager propagated via the message queue from the Communication Bridge and translates this as necessary to the OB_{APP} compliant control interface message sequences which are used to communicate with the FRMCS TOBA.

Via the OB_{APP} control protocol, the FRMCS user plane connection is setup with the trackside gateway which is the endpoint of the FRMCS communication path.

For details of the FRMCS Agent implementation please refer to Section 3.1.2

2.1.2 FRMCS Trackside Components

Although FRMCS Trackside is not the focus in the current project an end-to-end setup of the FRMCS service layer connection has to be implemented. This allows a setup using FRMCS service layer interaction between onboard FRMCS components and trackside FRMCS components providing an end-to-end communication.

The following components are defined

- FRMCS Trackside Gateway
- FRMCS Service Server (Core Network)
- Trackside Application
- Trackside IP Infrastructure

In the following sections these components are further described.

2.1.2.1 FRMCS Trackside Gateway

The FRMCS Trackside Gateway is a component of the FRMCS system. It allows the connections between OB and TS applications based on the MCx framework.

The main functions supported by the Trackside Gateway are:

- The TS_{APP} interface management
- The control plane (CP) management
- The user plane (UP) management

2.1.2.1.1 TS_{APP} Interface Management

The TS_{APP} interface management handles the TS_{APP} interface, which is the interface between the Trackside Gateway and the Trackside applications. It is compliant with the FFFIS v1.2 specification [17]. This interface is HTTP/2 based. The data contained in the body of the HTTP request and response is formatted as JSON.

The Trackside Gateway can manage two application types depending on the way their corresponding MCx client is managed:

- The Tight Coupled applications, this is currently mainly the voice application (Dispatcher). In such a case, the application manages, by its own, its MCx client.

- The Loose Coupled applications, these are other Trackside applications (such as ATO, ETCS, etc.). In such a case, the MCx client is managed by the Trackside Gateway, on behalf of the application.

Note that from the trackside gateway point of view, loose coupled and super loose coupled applications (using an FRMCS Agent providing the TS_{APP} interface client for the application) are not distinguished.

2.1.2.1.2 Control Plane Management

The Trackside Gateway manages the Loose Coupled applications' Control Plane. The Control Plane of the applications is based on the MCx framework. The Trackside Gateway manages the MCx client of each Loose Coupled application from its creation and its registration at the MCx server side to its deregistration and destruction.

The complete MCx Client life cycle is driven by the application's requests:

- On application's registration request, the Trackside Gateway create and registered the MCx client.
- On application's session opening request, the Trackside Gateway manages the SIP (Session Initiation Protocol) session via the MCx client and MCX server-side framework and the data plane is opened for data session.
- For an incoming session request for a Trackside application, the Trackside Gateway also manages the SIP invite acknowledgement and the data plane opening.
- On application's session closure, the data plane is ended.
- On application's deregistration, the MCx client is deregistered from the MCx server and it is deleted.

2.1.2.1.3 User Plane Management

The User Plane is managed per data session.

For each data session, a GRE over UDP (Generic Routing Encapsulation over User Datagram Protocol) tunnel is opened between the Onboard and the Trackside Gateways. This GRE tunnel allows the transmission of the IP data flow corresponding to the application's session. Inside the GRE tunnel, the data flow is bidirectional and handled at the IP level for efficiency and security.

2.1.2.2 FRMCS Service Server (Core Network)

In general, the FRMCS Service Server solution provided for this project is based on the client/server architecture of a 3GPP standardised MCx-based mission-critical communication system for fixed and mobile clients (Note that in a productive FRMCS deployment even more network elements and functions will be required – these are not deployed in this research project as their functionality is not required in this context). The FRMCS Service Server comes with Application Servers, which provide call-control-oriented services such as MCx-Application Server and application-oriented services such as Dispatcher Application Server. The MCx fixed and mobile clients rely on services provided by the MC core and use standardised secured MCx protocols. The central elements of the Kontron's FRMCS MCx solution provided for this are:

IP Multimedia Subsystem (IMS) -based MCx core in acc. with 3GPP TS 23.228

- MCx core with HSS, S/I-CSCF and Application Servers
- MCx application server MCx-AS in acc. with 3GPP TS 23.280 with IWF function

Media Resource Function

- Media Distribution and Transcoding

MCS Common Service

- Identity and Access Management (IAM/IdMS)
- Configuration Management (CMS)
- Group Management (GMS)

Session Border Controller

- HTTP proxy
- Firewall
- Proxy-Call State Control Function (P-CSCF)

IMS Service Layer

The IMS service layer is represented by modularly structured IMS-based SIP core implementing switching services that go beyond the usual range of functions of a conventional office switching system to satisfy whole range of needs of a mission-critical railway communication solution.

The core switching system allows real-time mission-critical communication over IP and it builds the base for providing enhanced services by the application servers.

The arrangement of the network functions of the SIP core corresponds to the 3GPP mission-critical architecture based on IMS architecture defined in 3GPP 23.228. The Mission-Critical Application Server, hosting the service logic, is connected towards the SIP core using ISC interface. All clients are provisioned in the HSS database to allow clients consuming application services via SIP core using client's MCx-UA.

The SIP core functions are combined into the components specific to the solution whereby functions are also included that go beyond 3GPP 23.228.

Application Service Layer

The application layer of the solution consists of:

- Mission-critical common services, including:
  - Identity and Access Management (IAM/IdMS): The IAM system is Identity and Access Management platform, which is managing digital identities and controls user access to the system by a framework of policies, processes and technologies. The IAM assigns user to a specific user group and ensures, that the user has the right level of access to resources and functions of the system. The main objective of the IAM is to assign one identity to each individual user and from there to maintain, modify and monitor access levels and privileges through each user's access live cycle.
  - Configuration and Group Management Server (CGMS): The Configuration and Group Management Server (CGMS) allows to configure one or more service application and configure data of the configuration and group management client. It provides as well functional role, profiles and group management functionality.
  - Location Management Server (LMS): Location management is realised as client/server architecture having location management server of the system on one side and the location management client of the client on the other side. The Location Management Server (LMS) of the Mission-Critical Common

Services is a central database of all location based data in the system. It retrieves and stores location information of the users and provides user location information to the MC service servers performing location-based functions such as location dependent addressing.

- Mission-critical application server (MCx-AS) The arrangement of the functions of the MCx-AS corresponds to the architecture specified by the 3GPP. The MCx-AS is responsible for all MCx signalling related to private, group, emergency calls incl. talker control, preconfigured and ad-hoc conferences, services known as supplementary services etc. For other needed but not call oriented functions it relies on mission-critical common services, providing Configuration and Group Management (CGMS) and Identity and Access Management (IAM/IdMS) functions.
- clients, fixed and mobile clients

The authentication and authorisation of the system users is handled at different levels:

- at the IAM/IdMS level for operators, administrators and users and clients,
- at the HSS for MCx identities of the clients.

2.1.2.3 Trackside Application

The FRMCS application is integrated with the trackside TOBA gateway, leveraging FRMCS services to facilitate seamless communication between an onboard application and a trackside application. This communication occurs transparently through an onboard gateway utilizing MCx FRMCS services.

The FRMCS services are independently invoked by an FRMCS Agent, ensuring the complete setup of mission-critical data communication. This setup enables secure and reliable data exchange between the onboard application operating within a controlled and secure environment and the trackside counterpart. The application is directly connected to the trackside gateway.

An FRMCS Agent is responsible for initiating FRMCS services using HTTP/2 and Server-Sent Events (SSE), facilitating bidirectional communication. These technologies enhance the efficiency of data transmission by enabling persistent connections and reducing the overhead associated with traditional polling mechanisms. This implementation ensures a scalable and resilient communication framework, supporting mission-critical railway operations.

The current application is a simple implementation designed to receive messages from the onboard application, replicate them, and send acknowledgment back to the onboard application.

2.1.2.4 Trackside IP Infrastructure

For the purpose of the research project, the trackside IP infrastructure is tightly coupled with the onboard IP infrastructure. This is due to the choice of the project to setup parts of the trackside infrastructure in the SBB lab premises and other parts in Kontron Lab premises.

The Edge Router is providing main connectivity between the components in the SBB lab and also to components via external connection to Kontron Lab components. The external connections are secured via VPN tunnels.

Please refer to Figure 10 for more details on the IP infrastructure and interconnection.

2.2 FUNCTIONAL CLUSTER DIAGNOSTICS

The objectives and concepts of the diagnostics implementation within this project have been introduced in chapter 2.3 of the former deliverable D36.2 [5], referring as well to documents of the Demonstrator Specification D36.1, e.g., particularly mentioning relevant sections as chapter 3.7 of the D36.1 Architecture document [4].

We are trying to provide self-contained and comprehensible content here, but the reader may be advised to study former deliverables of WP36 for better understanding, since we're aiming not to repeat content presented there. This applies particularly to the Modular Platform's general principles, which must be understood as a prerequisite.

The prior deliverables describe the scope of MCP-DIA, its peers, interfaces and main functions mainly on a logical level. Based on this architecture and former proposals for the realisation, the intention of this chapter is to provide further and deeper insights into design decisions, which have been the basis for the realised and partly still ongoing implementation. Physical deployments of relevant User Stories will be in detail depicted in the upcoming sub-chapter 3.2 of the section Implemented and Studied Functions and the sub-chapter 4.2 of the section Tests and Results.

The main purpose of the MCP-DIA diagnostics subsystem is the generic collection and propagation of fault codes or other application specific diagnostics data and the provision of diagnostics events. On top of that, MCP-DIA collects diagnostics data from the Modular Platform itself, as its health status and potential errors, e.g., resulting from faulty hardware components.

MCP-DIA does not detect undetermined failures of specific replicas. Such unexpected behaviour will be observed and detected by the safety layer of the platform and can only be obtained from the Platform Logging Services, which on the other hand can be utilised by the hosted applications as well, as described in the later sections of this chapter.

2.2.1 Components and Workflows

The following diagram has been introduced in D36.2 and is presented here again as the still valid starting point for the MCP-DIA sub-system design.

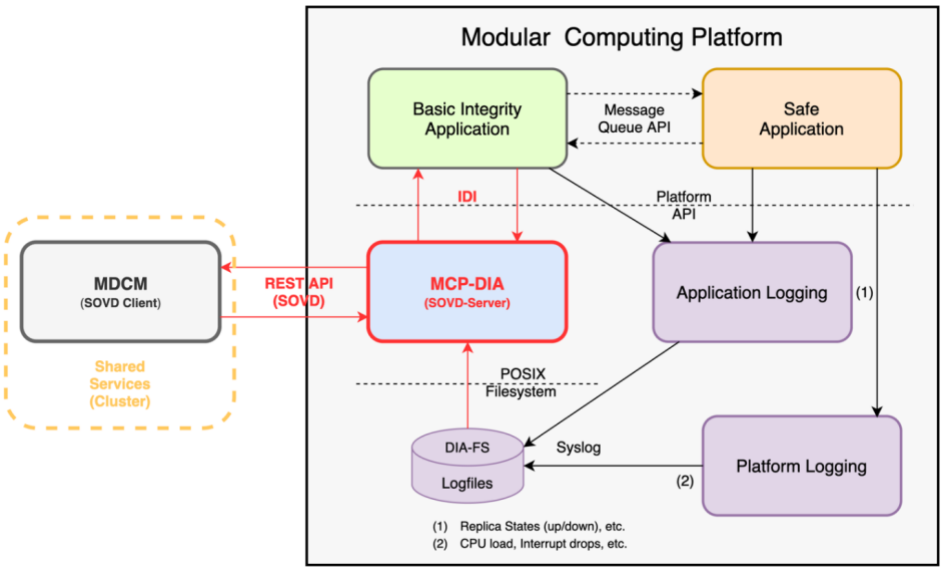


Figure 3: MCP-DIA Concept for R2DATO WP36 Demonstrator

1. **Safe Application** (Model-1 TaskSet(s))
   - Generates diagnostics data that needs forwarding to the SOVD Server (either via a Basic Integrity (BI) Application as Event Dispatcher or the Logging Service of the Platform)
   - Indirectly may receive diagnostics events/data via the Event Dispatcher BI Application
2. **Basic Integrity Application** (Model-3 TaskSet(s) using the IDI interface)
   - Exchanges diagnostics events/data with the SOVD-Server
   - Utilises the Logging Service of the Platform (e.g., for further root cause analysis)
3. **Basic Integrity Application serving as Event Dispatcher** (Model-3 TaskSet)
   - Dedicated application realising an Event Dispatcher between MCP-DIA and Safe applications, can be considered as an IDI to message queues dispatcher
4. **Diagnostics File System (DIA-FS) & Syslog**
   - Acts as an intermediary that forwards log messages from the hosted applications to the SOVD Server
   - Configured on the TAS platform to ensure centralized management and deployment
   - Providing additional status and health information of the platform itself
5. **SOVD-Server**
   - Provisioning of the IDI interface towards hosted application
   - Provisioning of the SOVD [8] REST API towards external (MDCM) client(s)
   - Receives log messages from syslog-ng via a TCP connection
   - Parses the log messages to extract the timestamp and source (to detect to which safe application the message belongs)
   - Stores the parsed log messages in a database, enabling efficient querying
   - Provides the diagnostic related logging endpoints for applications hosted on the platform
6. **MDCM** (Monitoring, Diagnostics, Configuration, Maintenance Service)
   - External SOVD client as described in [4] and [5], receiving collected data and exchanging diagnostics events through the REST API of the SOVD-Server

2.2.2 MCP-DIA Design Decisions

The idea of SOVD is to make systems diagnoseable using a widespread state-of-the-art technology stack for Vehicle Diagnostics. Using an SOVD server as MCP-DIA implementation is particularly well-suited for diagnosing POSIX-based systems due to several key advantages:

- **Standardized API:** SOVD uses a REST API that is independent of the programming language, making it versatile and easy to integrate with various systems, including those running on Linux.

- **Modern Technologies:** It leverages state-of-the-art technologies such as HTTP/REST, JSON, OAuth and mDNS which are commonly used in Linux environments.
- **Flexibility:** SOVD allows diagnostic workflows to be written in any programming or scripting languages that are suitable to trigger a REST interface.
- **Remote Diagnostics:** It supports remote diagnostics, enabling technicians to access and diagnose systems without needing physical proximity, which is particularly useful for Linux servers and embedded systems.

These features make SOVD a powerful tool for diagnosing and managing POSIX-based systems efficiently and effectively. The server deployed in the R2DATO project is implemented in RUST [9] to leverage its state-of-the-art frameworks and uses a widespread, powerful, performant and safe HTTPS stack to implement the SOVD standard.

The programming language RUST is memory safe by design and has very high potential to be qualified in future safety systems [10].

2.2.2.1 SOVD Service Client Interface (IDI)

The chosen SOVD server implementation provides a robust interface based on Protocol Buffers [11] serialization, ensuring efficient and reliable communication between clients and the server. The SOVD Server utilises a TCP based interface on the Message Listener that accepts Event-Envelops for faults and data.

2.2.2.1.1 Client Attachment

Clients can announce themselves as applications or components and attach their capabilities via the TCP Socket, leveraging Protobuf [11] serialisation for structured data exchange. Protobuf is widely established and utilised in different sectors due to its language-neutral, platform-neutral, extensible mechanisms, supporting quite easy integration and well documented supply-chain.

2.2.2.2 Communication Paradigms

The key challenge of implementing the SOVD standard on the TAS Platform is to combine the different communication paradigms used by the systems.

Request-Response is a very common diagnostics communication principle and provides:

- **Efficiency**: By using a request-response approach, diagnostic tools can precisely query for specific data, such as diagnostic trouble codes (DTCs), sensor readings, or system status. This targeted communication reduces unnecessary data transfer on limited communication channels and speeds up the diagnostic process.
- **Reliability**: The request-response paradigm ensures that each request is acknowledged with a response, providing a clear and reliable communication path.

These advantages can come into play in the communication between MDCM and SOVD server where applications want to receive specific diagnostic data from explicit applications within the system without consuming all information of all applications. The request response principle is supported in SOVD end-to-end up to the hosted applications and therefore as well over the IDI interface.

On the other hand, especially for the safe (Model-1 TaskSets) applications within the TAS platform, the requirement for deterministic and predictable execution times may outweigh

performance or efficiency requirements. A key requirement is that the redundant system shall always act predictable and deterministic. For this reason, the Model-1 and Model-3 TaskSets may be required to send fixed scheduled events to the SOVD Server reporting their diagnostic state in cyclic manner.

The following figure illustrates the basic communication concept, utilising a combination of both paradigms as an example. Data gets requested or continuously pushed from the hosted applications towards the MCP-DIA server and can be obtained by the external MDCM by using the request response REST interface of SOVD.

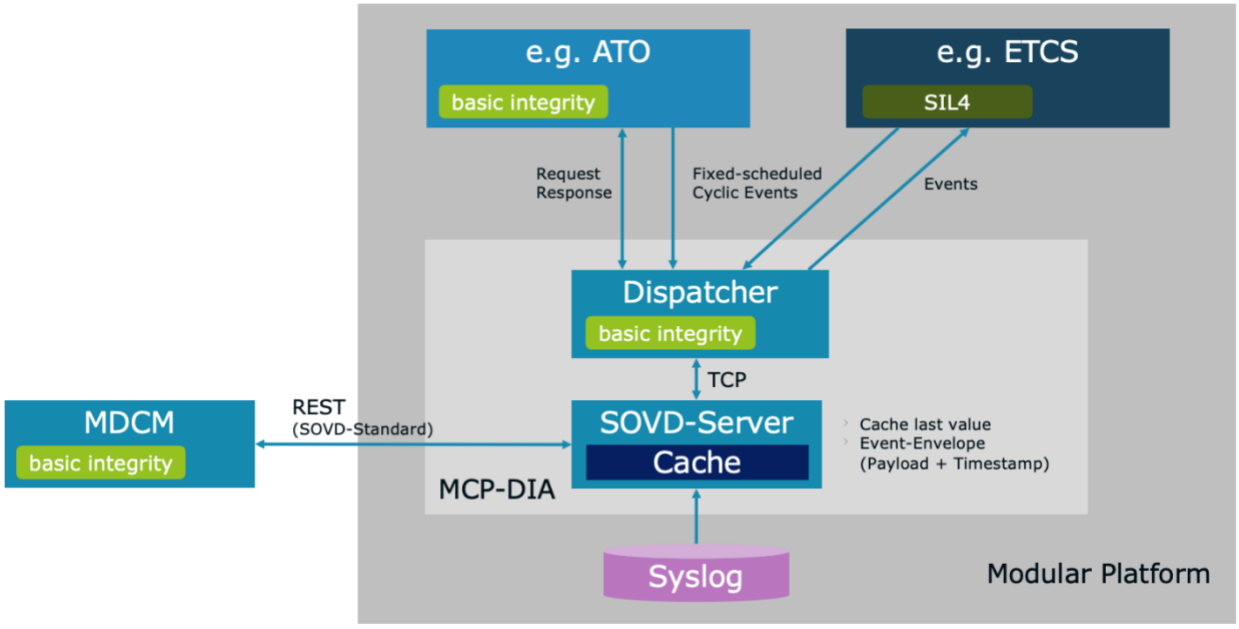


Figure 4: MCP-DIA Basic Communication Concept

The Diagnostics Concept [12] of the System Pillar Transversal domain proposes that diagnosis data gets pushed continuously to external diagnostics data aggregators (as the MDCM client). This paradigm can be realised with SOVD concepts as subscription to cyclic provision of data but needs further study and more concrete specification input from the SP Transversal domain.

2.2.2.3 Data Cache

To connect the above-mentioned communication paradigms, a Data Cache component is needed. It caches the latest data value in an "Event Envelope" that contains both the payload and its timestamp. The Data Cache component will consist of three main parts:

- The Message Listener, which is configured as diagnostics endpoint in the applications and listens to new messages.
- The Cache Manager, which stores the latest data value in an "Event Envelope" and updates the cache whenever a new message is received.
- The Request Handler, which processes incoming client requests by retrieving the latest "Event Envelope" from the Cache Manager and responding to the client.

The Event-Envelope contains the following data:

- Source (ID of the source application)
- Sequence Number
- Timestamp
- Payload-Type (Data/Fault/…)
- Payload

The individual payloads for fault, data and further SOVD elements are defined in the protocol buffers interface of the SOVD server.

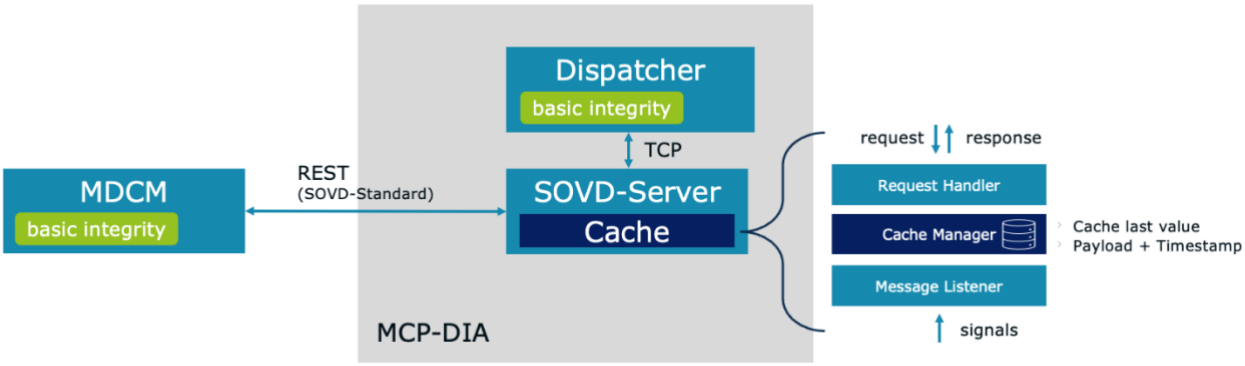


Figure 5: MCP-DIA Cache Concept

2.2.2.4 Diagnostics for Safe Applications

2.2.2.4.1 Communication Restrictions

Safe Apps are restricted to communicating only with Model-1 and Model-3 TaskSets over message queues.

2.2.2.4.2 Communication Methods

In case safe applications exchange non-replica local data, this data typically requires voting and can be sent via voted message queues, ensuring end-to-end safety, high reliability and security.

2.2.2.4.3 Fixed Deterministic Scheduling

Safe applications are often running in a deterministic fixed schedule and replica deterministic data provided to MCP-DIA should not be recorded duplicated from each replica. Thus, all messages provided by replicated applications shall be deterministically sent without being triggered prior by a request. The paradigm of sending data on a fixed cyclic schedule is chosen. This concept also enables cyclic provision of diagnostics data as demanded by the System Pillar Transversal Diagnostics concept [12].

The following sequence diagram proposes message flows that include both communication paradigms, Request-Response and Fixed Schedule Cyclic data transfer. The configuration part is optional and might fall back to a default static configuration.

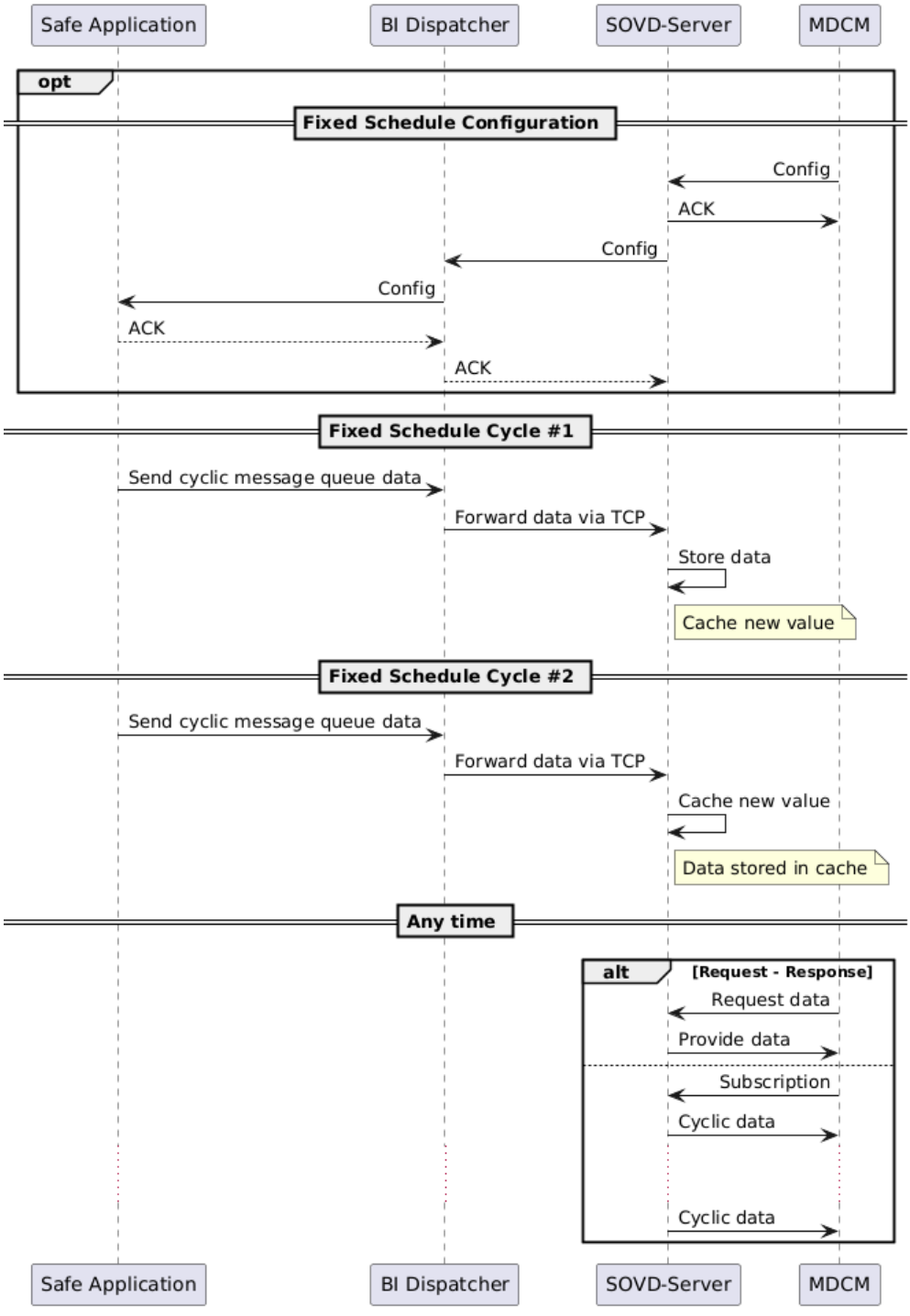


Figure 6: Message flows and caching in SOVD-Server

2.2.2.5 Basic Integrity Application as Event Dispatcher

The user stories 2.5.3 and 2.5.7 introduced in D36.1 “Demonstrator Specification, User Stories & Test Cases” [2] are proposing diagnostics events and bidirectional data exchange

to basic integrity and safe applications hosted on top of the Modular Platform Runtime Environment.

Within the TAS Platform, safe applications (Model-1 TaskSets) cannot open a TCP socket. To allow safe applications to publish messages to the SOVD server, a dispatcher application (Model-3 TaskSet) needs to be implemented in between. The communication paradigms between the Basic Integrity Dispatcher to safe applications is derived from the design pattern introduced in the chapter 3.3 “Demo Applications” of the former deliverable D36.2 [5]. This pattern demands for communication over message queues, the only communication paradigm allowed to exchange data with Model-1 TaskSets.

The dispatcher application reads message queues filled by the safe application and forwards the data to the SOVD-Server. Since the source of the data is contained within the Event-Envelope, the dispatcher does not need to parse the message and can just forward it to the TCP stream towards the SOVD-Server.

The following diagram is depicting the principle of bi-directional diagnostic event distribution via message queues between the BI Dispatcher and safe applications. Asynchronous and cyclically sent events are supposed to ensure a reliable diagnostic communication process while not interfering with the replica determinism of safe applications.

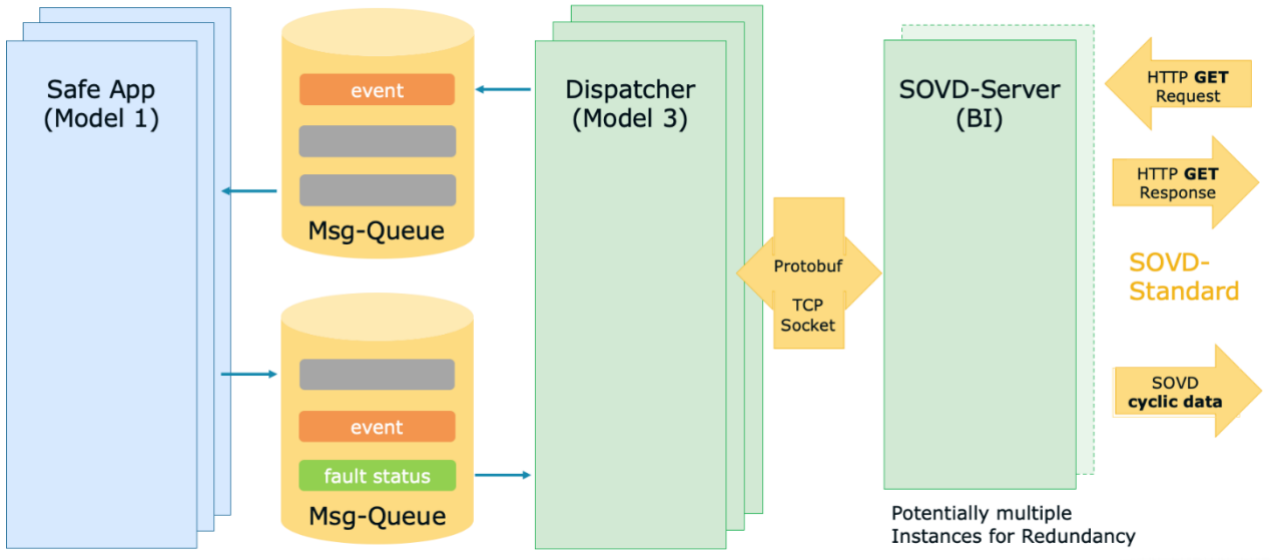


Figure 7: Event and exemplary fault provision via BI Dispatcher

Since replicas of safe applications run on multiple Computing Elements (e.g., in a 2oo3 setup), the BI Dispatcher as a Model-3 TaskSet needs to be deployed on all corresponding CEs as well. As result data received from a Model-1 TaskSet will be replicated to multiple dispatcher instance (deployed on each CE) and must be discarded on those instances that are not connected to the active SOVD-Server instance.

2.2.2.6 Logging

2.2.2.6.1 Syslog vs Message-Queues

Syslog and syslog-ng as a network enabled messaging system are widely established and a commonly utilised approach for monitoring and logging on POSIX systems, in particular as well on the TAS platform. Syslog-ng can be used as a source to collect diagnostics data

from multiple Computing Elements running the TAS platform, but the approach has some disadvantages and known limitations:

- Syslog in general is designed for unidirectional communication of applications towards a textual only logging output, but typical diagnostics need a bidirectional interface to explicitly trigger data reads or to start data subscriptions.
- Syslog is expected rather to cover unexpected events as logging data of the applications that gets written occasionally, not within a concrete diagnostics context but as implicit events for root cause analyses of misbehaviour.
- Syslog-ng may lose messages under heavy load, making it less reliable for critical diagnostics.

Message-Queues (plural, since one message queue is unidirectional as well) are conceptually more reliable but on the other hand not very commonly used for diagnostics system. MCP-DIA as well utilises them only where necessary, e.g., for event distribution towards Model-1 TaskSets. Since Message-Queues are capable of transporting any kind of data, they can be used for diagnostic event and data distribution in the combination with TCP as described above in Figure 6 and Figure 7.

2.2.2.6.2 The SOVD Logging API

According to ASAM SOVD Standard [8] (V1.0 - 6.14 API Methods for Logging) the SOVD Server provides options to query logs in the RFC 5424 - The Syslog Protocol format.

Query Parameters for RFC5424 Logs with Entity Path

1. **Entity Path**:
   - **Description**: Specifies the component or subsystem of the vehicle.
   - **Format**: {base_uri}/apps/{app-id}/logs
2. **Severity**:
   - **Description**: Filter logs based on their severity level.

SOVD Log Level	RFC5424 – Syslog Protocol
Fatal	Emergency, Alert, Critical
Error	Error
Warn	Warning
Info	Notice, Informational
Debug	Debug, Verbose

Table 1: SOVD vs. Syslog Levels

3. **Date and Time**:
   - **Created after**:
     - **Description**: Retrieve logs created after a specific date and time.
     - **Format**: YYYY-MM-DDTHH:MM:SSZ

- **Created before**:
  - **Description**: Retrieve logs created before a specific date and time.
  - **Format**: YYYY-MM-DDTHH:MM:SSZ

Example Query with Entity Path

To retrieve log entries with severity error created after January 1, 2025, an example query like this can be used:

- GET {base_uri}/apps/{app-id}//logs/entries?severity=error&created-after=2025-01-01T00:00:00Z HTTP/1.1

Example RFC5424 Log Entry with Entity Path


```
{
  "timestamp": "2025-02-06T12:00:00Z",
  "context": {
    "type": "RFC5424",
    "host": "Linux",
    "process": "systemd",
    "pid": 1
  },
  "severity": "error",
  "msg": "Failed to start service",
  "href": "{base_uri}/apps/{app-id}//bulk-data/logs/server.log"
}
```


2.2.2.6.3 The SOVD Bulk-Data API

The actual log file will be provided via bulk-data endpoints according to ASAM SOVD Standard [8] (V1.0 - 6.13 API methods for handling of bulk data).

The transfer of bulk data between the SOVD server and SOVD client is highly suitable to transfer batch data due to the use of MIME (RFC 2045/2046) [14] standards and for signed/encrypted multipart MIME security (RFC 1847) [15]. The MIME standards mandate the inclusion of a Content-Type header in requests and responses, ensuring that both the SOVD server and client can accurately interpret the data.

When an SOVD client requires extra information, such as a file name, this information is provided as a JSON object, making the Content-Type multipart/form-data a common choice.

2.2.2.6.4 Logging Concept - TAS Platform to SOVD Server

The concept is to utilize syslog-ng to forward log messages from a safe application to the SOVD Server. The SOVD Server then parses these log messages, extracts the timestamp and source, and stores them in a database. This allows for efficient querying and retrieval of log messages based on timestamp and data source through the SOVD logging interface.

Since syslog-ng already provides means to forward logging via TCP there is no additional dispatcher needed.

2.2.2.6.5 Syslog-ng Extension of the SOVD Server

The syslog-ng extension to the SOVD server will provide an interface for performant querying of TAS platform logs.

It will perform these steps for all incoming log messages:

1. Capture the messages of the TAS platform logging system
2. Parse the messages to extract data-source and timestamp
3. Filter the messages based on predefined filter criteria
4. Store the parsed messages in a database with indexing of data-source and timestamp

2.2.2.6.6 Sequence Diagram Logging

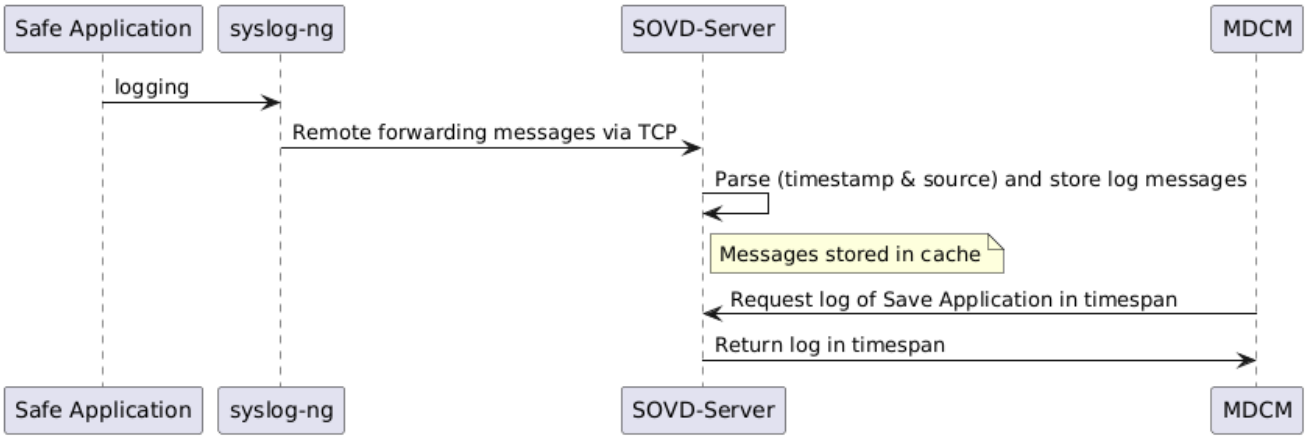


Figure 8: Example sequence for logging via syslog-ng

2.2.2.6.7 Configuration of syslog-ng on TAS Platform

- Define the source to capture log messages from the safe application
- Configure the destination to forward log messages to the SOVD Server over TCP
- Set up the log path to connect the source and destination

Example syslog-ng configuration sketch

```
source s_safe_app {
    file("/var/log/safe_app.log");
};
destination d_sovd_server {
    tcp("10.36.100.20 port(514));
};
log {
    source(s_safe_app);
    destination(d_sovd_server);
};
```

2.3 PHYSICAL ARCHITECTURE

A high-level view of the demonstrator setup has already been introduced in D36.2 [5], Figure 15. This Figure is represented below again as the starting point to derive a much more detailed physical architecture including all major hardware and software components deployed in the laboratories of SBB and Kontron.

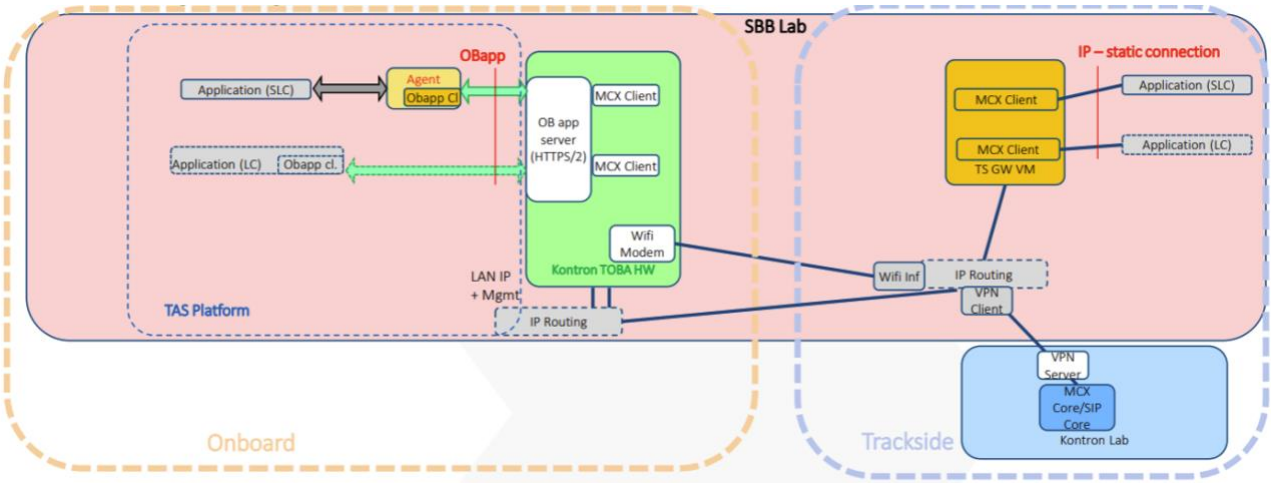


Figure 9: High Level Architecture and Laboratory Split of the end-to-end Test system

The Onboard Demonstrator consists mainly of:

- The TAS Platform and all its integrated software components
- The FRMCS TOBA system as the link to the FRMCS infrastructure
- The routing and connectivity items for communication
- Simulation and test environments deployed on the laboratory server dedicated hardware

The following sections describe the components that are integrated into the laboratory in more detail.

Based on the initial architecture described in D36.2 [5], the physical lab was developed as depicted in the following overview.

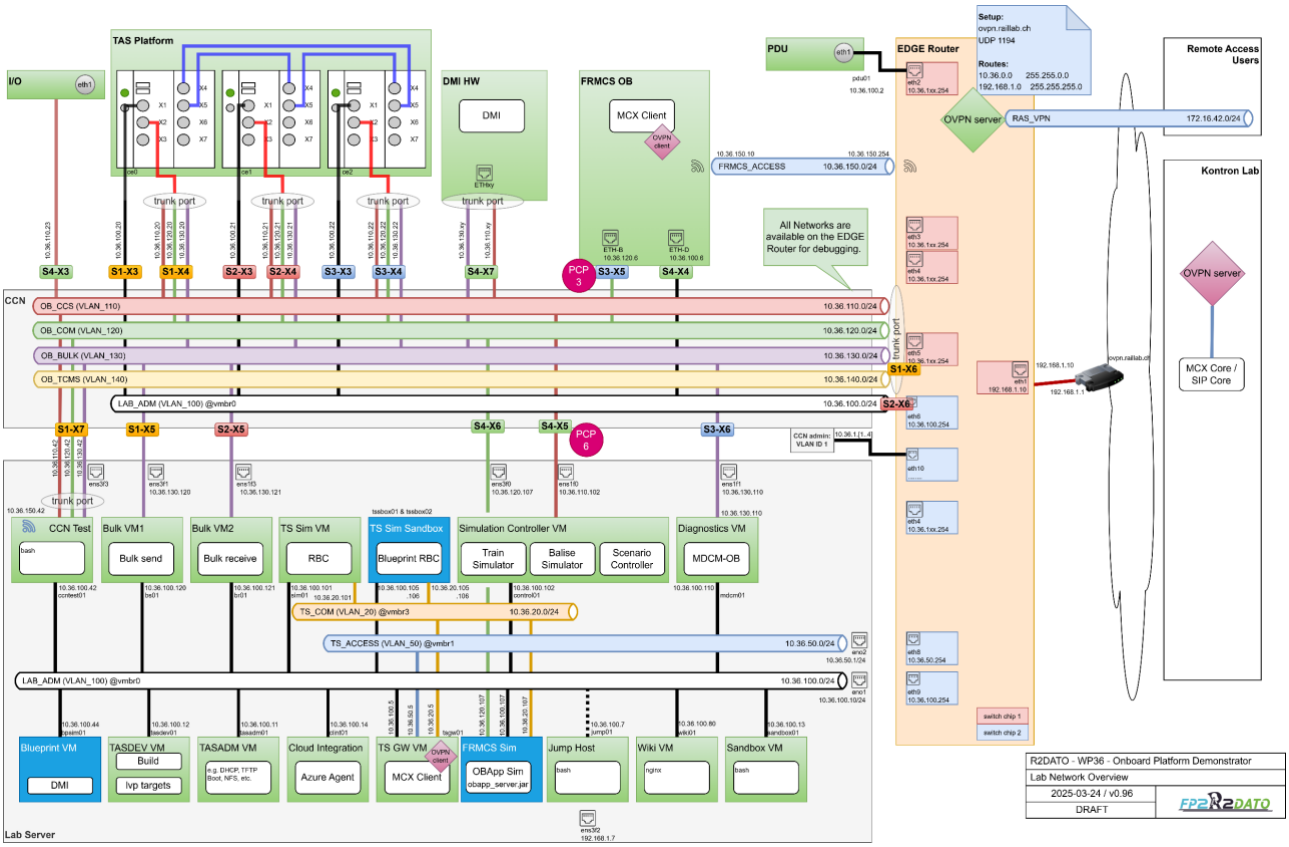


Figure 10: Physical Architecture Overview

2.3.1 Description of Laboratory Components

2.3.1.1 TAS Platform Computing Node

The TAS Platform consists of 3 independent G25A Computing Elements from Duagon. They are directly interconnected with each other in a ring configuration over a dedicated ethernet link to fulfil the required safety and redundancy requirements of the 2oo3 configuration.

Each Computing Element has separate ethernet links to the administration network (LAB_ADM VLAN) and a trunk port providing the segregation and quality of service of the different network traffic profiles.

There exists a dedicated power supply unit for each Computing Element.

Figure 11 shows the Computing Node with three Computing Elements. Each have the same setup with (from left to right) a Duagon G25A embedded single board computer, a Duagon G211 ethernet interface providing four additional ports with M12 connectors and an individual Duagon PU21 120W power supply.

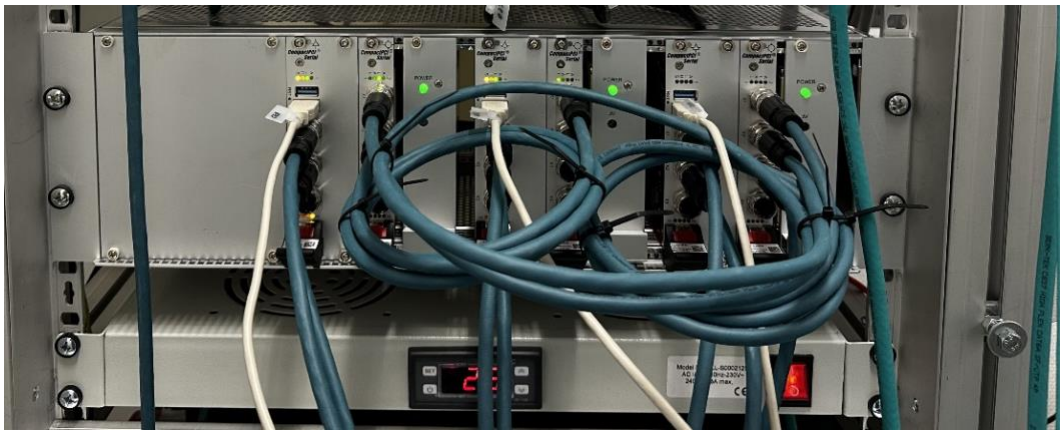


Figure 11: TAS Platform Computing Node

2.3.1.2 Driver Machine Interface

The Driver Machine Interface (DMI) provides the user interface to the ETCS application running on the TAS Platform. It is connected over the CCN using the OB_CCN VLAN dedicated to communication between all CCS Onboard subsystems.

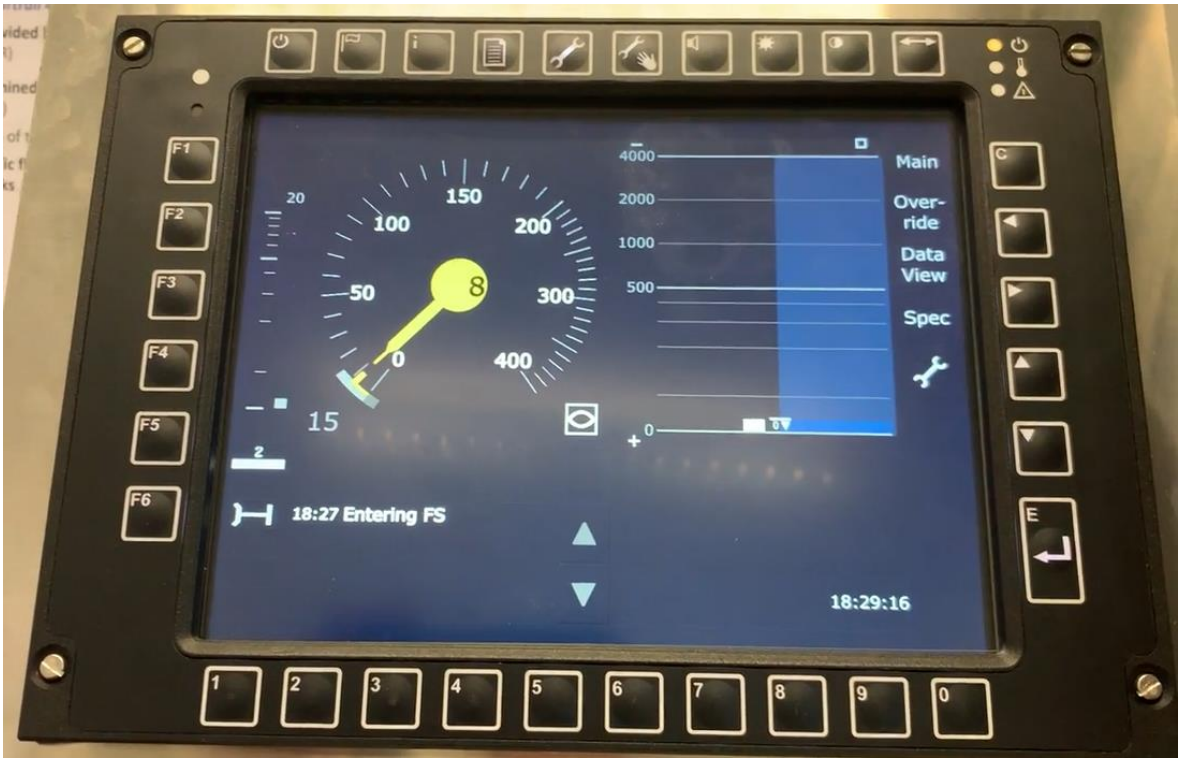


Figure 12: Driver Machine Interface

2.3.1.3 FRMCS OB

The FRMCS Onboard Gateway implements the FRMCS Gateway and FRMCS Radio Functions used by the Onboard Applications (e.g., diagnostics, ETCS) to connect to the simulated trackside applications (e.g., RBC Simulator).

Instead of using the 5G Mobile Communication, the FRMCS Onboard Gateway uses a WiFi connection to the EDGE Router of the Lab. Note that the usage of WiFi instead of 5G Mobile Communication may affect Performance during the tests. However, as the main goal of the tested user stories are functional testing, there is no major impact expected.

Figure 13: FRMCS OB

2.3.1.4 PDU

Power Distribution Unit (PDU) allows to remotely power cycle the computing elements of the TAS platform in case of an unrecoverable lockup or to simulate a power failure on one of the computing elements.

Figure 14: PDU

2.3.1.5 EDGE Router

The EDGE Router connects the Lab to the Internet and provides the OpenVPN access to the partners of the R2DATO WP36 to deploy, test and administrate their applications.

Figure 15: EDGE Router

2.3.1.6 I/O Unit

The I/O Unit is connected to the OB_CCN VLAN of the CCS Communication Network (CCN) and can be used by the application(s) running on the physical targets of the TAS Platform.

It provides four sockets, which can be used as output channels. Additionally, the first two sockets are equipped with indicator light to visualize the status of the output.

Figure 16: I/O Unit

2.3.1.7 CCN

The CCS Communication Network (CCN) is based on four railway grade ethernet switches interconnected in a ring to provide network resilience. The switches implement the Rapid Spanning Tree Protocol (RSTP) to react on topology changes due to physical link failures within the CCN.

The CCN is configured with multiple VLANs to segregate and prioritize the network traffic based on the application requirements. For details, please refer to Figure 10 and Table 2.

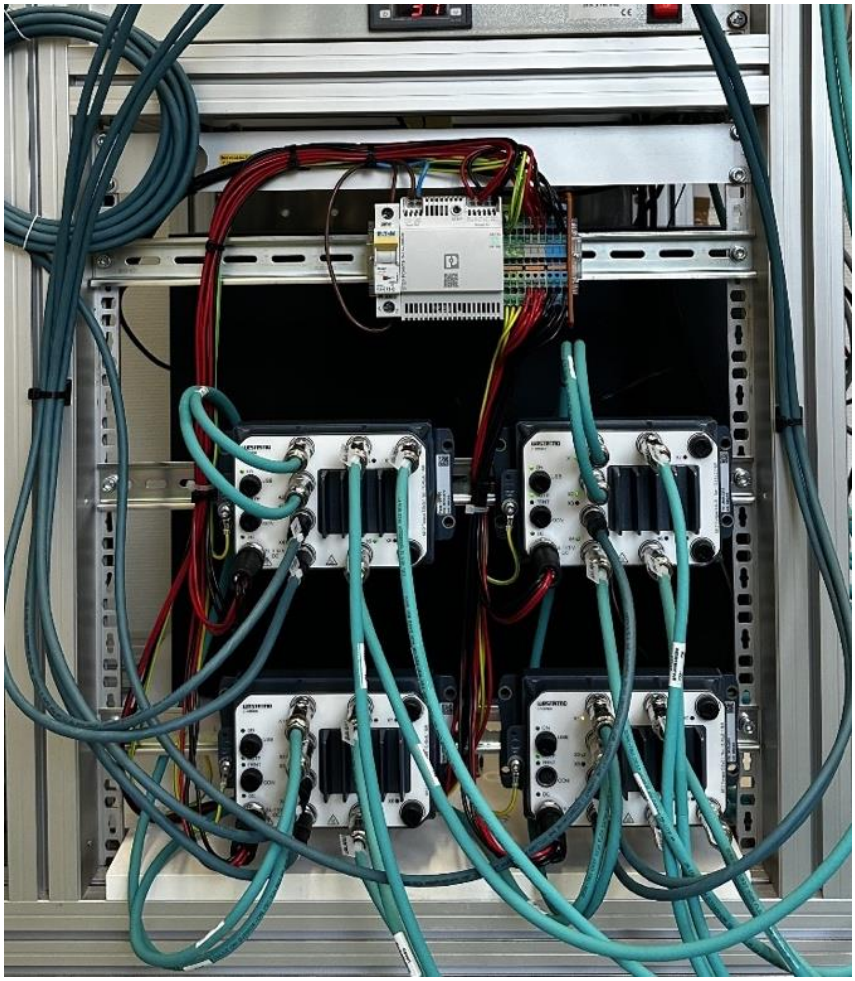


Figure 17: CCN

2.3.1.8 Laboratory Server

The Laboratory Server consists of a standard enterprise tower server running the Proxmox Virtualization Environment as the Hypervisor. The server is equipped with multiple ethernet interfaces assigned exclusively to the different virtual machines for proper segregation of the network traffic and, where required, to link the applications to a dedicated ethernet port of the CCN.

Figure 18: Laboratory Server

The table below provides an overview about the relevant VMs respectively containers running on the Lab Server and their purpose (refer also to Figure 10: Physical Architecture Overview).

Name	Description
CCN Test VM	The CCN Test VM has direct access to all VLANs on the CCN for the diagnostics of the CCN.
Bulk VM1 & VM2	The Bulk VMs are directly connected to the Onboard bulk VLAN (OB_BULK) of the CCN and can be used to generate network load on the CCN.
TS Sim VM	The trackside simulation VM is prepared to run the RBC simulator planed in task 3 of this work package.

Name	Description
Simulation Controller VM	The simulation controller VM is prepared to run the simulators required by the ETCS application as well as scenario controller to configure and orchestrate the involved simulators.
Diagnostics VM	The diagnostics VM runs the MDCM diagnostics client collecting diagnostics information from the Onboard applications.
TASDEV VM	Dedicated VM for development tasks, as building software artifacts in the laboratory.
TASADM VM	Dedicated VM for administration, e.g., provides an NFS share that gets mounted on VMs and the target hardware for provision of software deployments.
Cloud Integration VM	The Cloud Integration VM acts a proxy between the WP36 Azure DevOps environment and the Lab. It runs the Azure DevOps Pipeline Agent and allow continuous integration pipelines to be executed on the Lab. This VM can access the network file share and the administration VLAN to configure and restart the TAS platform computing elements for automated tests.
TS GW VM	The TS GW VM runs the trackside FRMCS Gateway service required by the trackside application (e.g., RBC Simulator) to register with the FRMCS Core.
Wiki VM (LXC)	The Wiki Container runs the Lab Wiki providing the documentation of the Lab setup and how-to use it.

Table 2: VMs on the Laboratory Server

3 IMPLEMENTED AND STUDIED FUNCTIONS

Based on the Architecture described in the previous Chapter 2, the current chapter focuses on the implemented and studied functions. The description is grouped according to the two considered Functional Clusters FRMCS and Diagnostics.

3.1 FUNCTIONAL CLUSTER FRMCS

The Functional Cluster FRMCS performs the necessary functions to provide data communication over the FRMCS network for the onboard applications.

This chapter describes only the entities of the Functional Cluster FRMCS, which have been developed and/or investigated during this project task. Other – standard FRMCS components which are used for the data transmission are already described in the relevant FRMCS Architecture section 2.1 above.

The investigated functions are either physically implemented or theoretically investigated if a practical implementation was not feasible within the project's timeframe.

The following functions have been implemented in the laboratory for the Functional Cluster FRMCS:

- Safe Application Logic
- Communication Manager
- Communication Gate
- Communication Bridge
- Control Handler (FRMCS Agent)

The trackside functions to provide the endo-to-end connection are the following:

- FRMCS trackside gateway
- Trackside application

As the focus in this project is on the Onboard Demonstrator, the trackside functions are not detailed further in this section.

Theoretically investigated functions in the Functional Cluster FRMCS:

- FRMCS Onboard Gateway redundancy options
- FRMCS Onboard Gateway deployment options

3.1.1 Safe Communication Architecture (Blueprint)

The communication architecture outlined in chapter 2.1.1 has been established as a foundational reference for future porting activities related to a specific ETCS application on the platform. It is important to note that this blueprint intentionally has its limitations; the primary objective was not to create a comprehensive example that addresses all potential exceptions and edge cases. Instead, it serves as a conceptual implementation to illustrate the overarching approach.

This blueprint features a safe onboard function that integrates with an onboard service to facilitate user interaction and exchanges information with two offboard services, which are accessible via FRMCS.

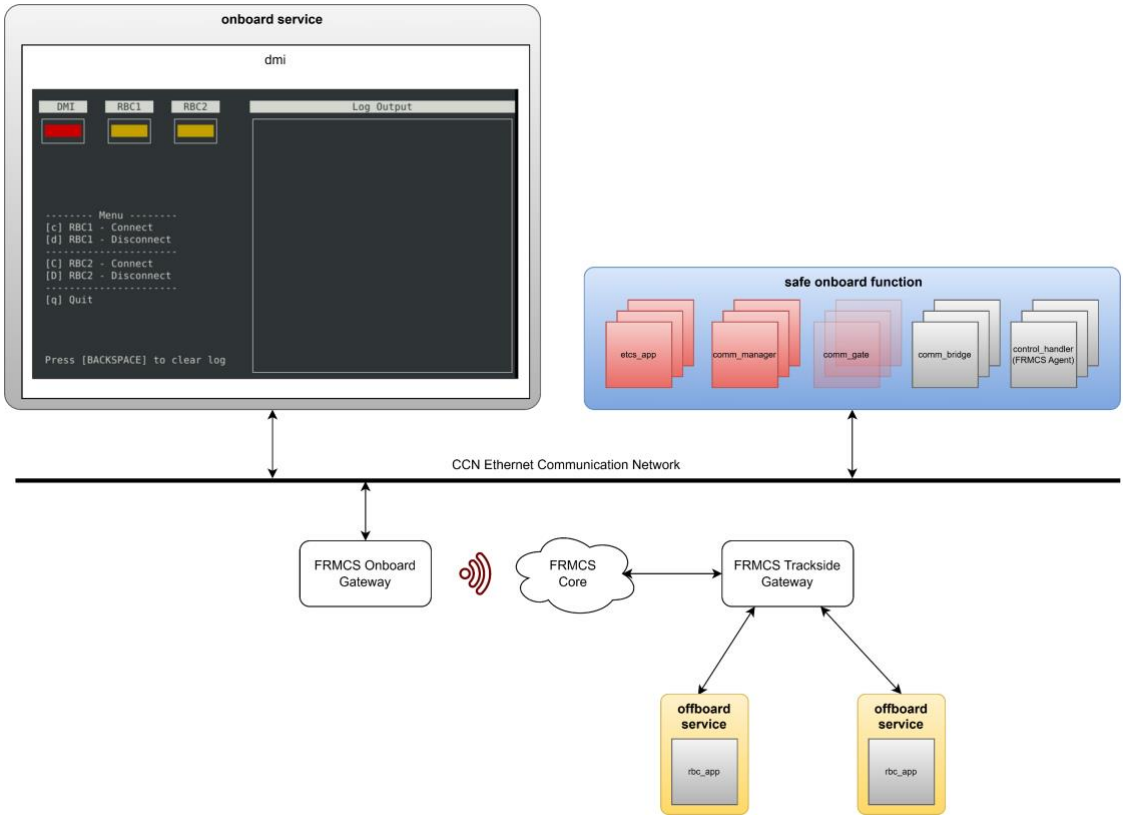


Figure 19: High-level blueprint architecture

The implemented blueprint realises a unified Communication Bridge for managing both onboard and offboard TCP/IP sessions. To facilitate the handling of various communication sessions through a single set of Communication Manager and Communication Bridge, a straightforward internal control message protocol has been established. Offboard services are exclusively accessible via FRMCS. Therefore, prior to establishing a TCP/IP session, the Control Handler must initiate a request for the corresponding FRMCS communication tunnel using the OB_{APP} interface to the onboard FRMCS Onboard Gateway (TOBA Box).

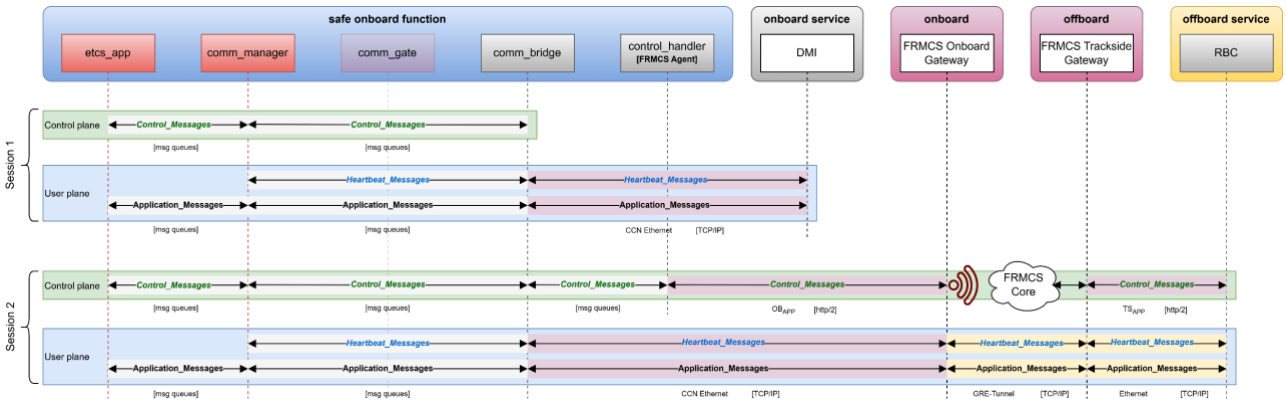


Figure 20: Control and user message flow

To effectively test a failover scenario involving the transition of communication from a faulty computing element to a functional one, the Communication Manager (comm_manager) must continuously monitor the health of the communication session. To facilitate this, it implements a straightforward heartbeat message exchange mechanism that enables the detection of any breakdown in the communication link.

The diagram below illustrates the concept of utilizing heartbeat messages to monitor the health of active sessions with both onboard and offboard services. It is important to note that this diagram serves as a simplification; the safe onboard function encompasses a comprehensive set of Model 1 and Model 3 TaskSets, as detailed in Chapter 2.1.1. As mentioned in the previous paragraph, the heartbeat monitoring is handled by the Communication Manager.

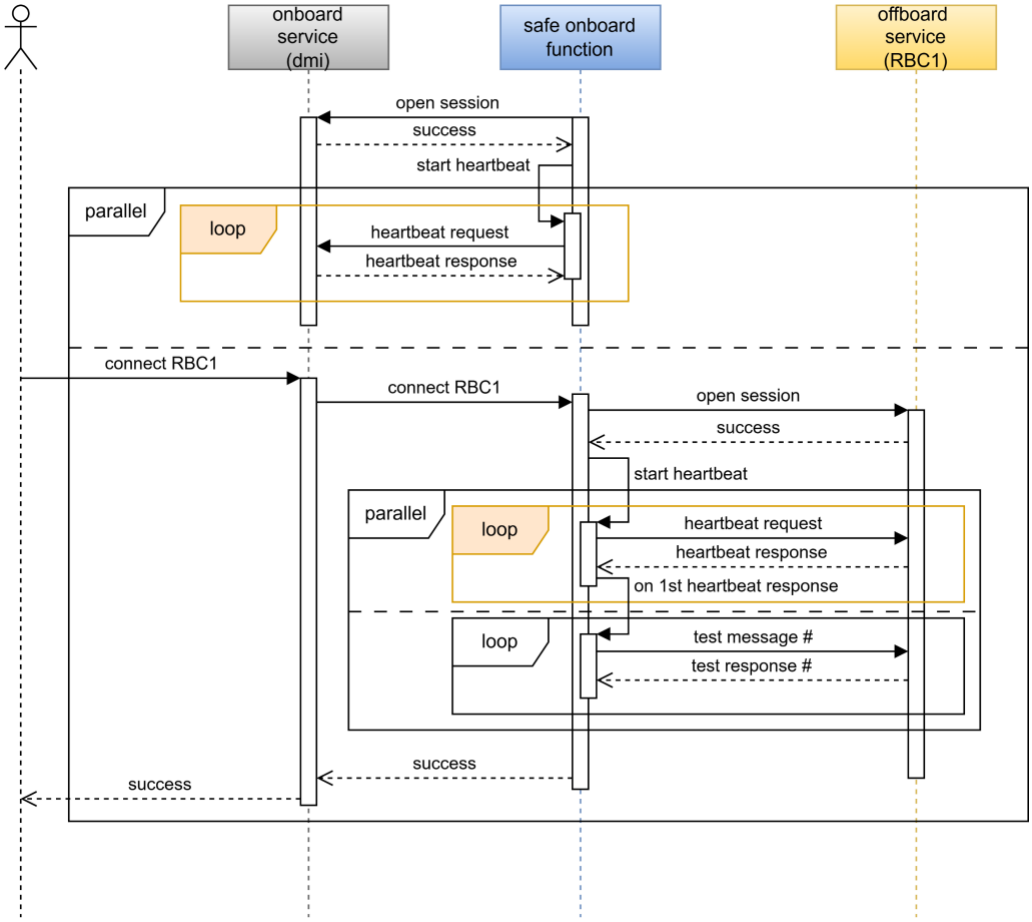


Figure 21: Heartbeat based health monitoring of active sessions

For the sake of simplicity, in the event of a heartbeat failure, the Communication Manager presumes that the sending computing element has encountered a failure and subsequently redirects external communication to the next available computing element. It is important to emphasize that a heartbeat failure does not necessarily indicate a failure of the computing element responsible for external communication; it may arise from various other issues, such as network disruptions, external service failures, or other anomalies.

Initially, the Communication Manager (CM) designates CE0 as the dedicated computing element for external communication. In the event of a failure of CE0, the CM will attempt to transfer external communication to CE1. If CE1 subsequently fails, the CM will then route

external communication to CE2. Should CE2 also fail, the external communication will revert to CE0, which is presumed to have recovered at that point.

Even after a computing element recovers from failure, the management of external communication does not automatically revert to the recovered computing element. Instead, the handling remains with the currently active computing element, ensuring continuity in communication operations.

Upon startup, the Safe Application Logic automatically establishes a connection to the Onboard DMI Service. Once this connection is successfully initiated, the session health status is visualized, displaying heartbeat signals and information regarding the currently active computing element responsible for external communication.

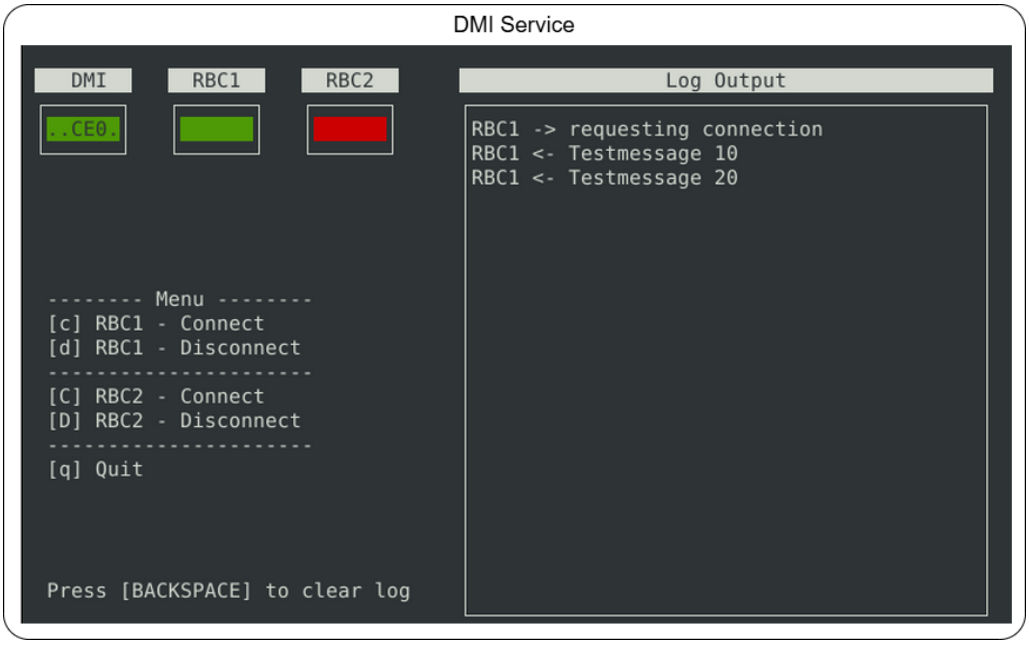


Figure 22: Heartbeat based health monitoring of active sessions

The menu provides options to connect to and disconnect from two offboard services, RBC1 and RBC2. The connection status, as reported by the Safe Application Logic, is represented by a coloured icon: green indicates a successful session establishment, red signifies that the session has been terminated, and orange denotes that the DMI lacks information from the Safe Application Logic.

On the right-hand side, the DMI displays log outputs pertaining to ongoing activities. This includes details of connection and disconnection requests, as well as updates on the number of test messages transmitted from the Safe Application Logic to the corresponding RBC. These updates are generated after every 10 test messages have been sent.

3.1.2 FRMCS Agent

The FRMCS Agent is a fundamental component that enables communication and control between onboard (OB) and trackside (TS) applications within the railway system. It is responsible for setting up FRMCS services via MCx, ensuring transparent integration for safe applications without the need to have a direct connection to the FRMCS specific OB_{APP} interface.. This is including service management and Quality of Service (QoS) enforcement with regards to the FRMCS system. By abstracting the complexities of FRMCS

communication, the FRMCS Agent enables onboard applications to seamlessly leverage FRMCS capabilities while maintaining compliance with railway safety and operational standards

The FRMCS Agent is fully compliant with the FRMCS FFFIS v1.2 specification [17], implementing communication based on HTTP/2 and Server-Sent Events (SSE). It provides a simple interface for interaction with onboard applications running on the TAS platform.

3.1.2.1 System communication data flows:

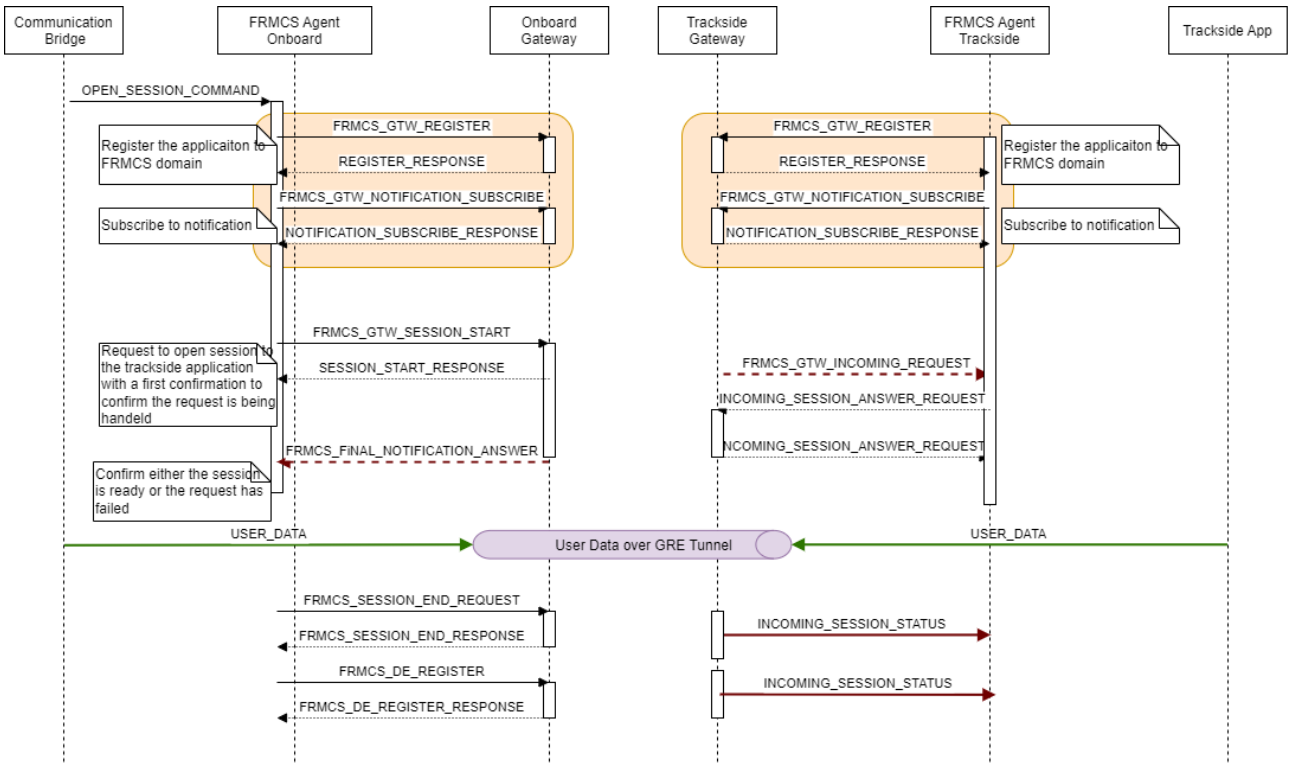


Figure 23: System communication data flows

3.1.2.2 Application Interface Description

In principle four relevant communication interfaces are covered:

1. Communication Bridge and FRMCS Agent Interaction
2. FRMCS Agent and Onboard Gateway Communication (FRMCS Services)
3. Trackside Agent and Trackside Gateway Communication (FRMCS Services)
4. User Data Exchange Between Communication Bridge and Trackside Application

Each segment plays a critical role in establishing, managing, and terminating communication sessions while ensuring data integrity, redundancy, and compliance with railway safety standards.

3.1.2.2.1 Communication Bridge and FRMCS Agent Interaction

After initial start from safe application the Communication Bridge initiates communication by sending an OPEN_SESSION_COMMAND to the FRMCS Agent.

The FRMCS Agent is responsible for managing the FRMCS service setup transparently.

3.1.2.2.2 FRMCS Agent and Onboard Gateway Communication (FRMCS Services)

The FRMCS Agent registers the onboard system to the FRMCS domain via the Onboard Gateway by calling:

- FRMCS_GTW_REGISTER (Registration Request)
- REGISTER_RESPONSE (Confirmation from Onboard Gateway)
- Subscription to event notifications is handled through:
- FRMCS_GTW_NOTIFICATION_SUBSCRIBE
- NOTIFICATION_SUBSCRIBE_RESPONSE

Once registered, the FRMCS Agent initiates a session request with the trackside application by invoking:

- FRMCS_GTW_SESSION_START
- SESSION_START_RESPONSE (Indicates session acceptance and processing)
- The session state is monitored, and final confirmation (FRMCS_FINAL_NOTIFICATION_ANSWER) ensures a successful setup.
- The FRMCS Agent operates independently from the onboard application but is triggered by control signals from the Communication Bridge via the Control Handler.

3.1.2.2.3 Trackside FRMCS Agent and Trackside Gateway Communication (FRMCS Services)

The Trackside FRMCS Agent initializes FRMCS services upon startup, ensuring readiness for incoming connections.

Similar to the onboard counterpart, the trackside system registers itself via the Trackside FRMCS Agent:

- FRMCS_GTW_REGISTER
- REGISTER_RESPONSE

It subscribes to necessary notifications via:

- FRMCS_GTW_NOTIFICATION_SUBSCRIBE
- NOTIFICATION_SUBSCRIBE_RESPONSE

When an onboard session request is received, the trackside system processes:

- FRMCS_GTW_INCOMING_REQUEST
- INCOMING_SESSION_ANSWER_REQUEST (Validation of session request)
- INCOMING_REQUEST_RESPONSE (Final session approval)

The Trackside FRMCS Agent runs as a persistent service, ensuring the system is always prepared for incoming communication sessions.

3.1.2.2.4 User Data Exchange Between Safe Application and Trackside Application

Once the session is established, User Data Exchange occurs between the Safe Application and Trackside Application, where the user data is encapsulated into a GRE Tunnel between the FRMCS onboard gateway and the FRMCS trackside gateway.

Data flow is bidirectional and handled at the IP level for efficiency and security.

The session remains active until explicitly terminated by either the Safe Application or Trackside Application.

3.1.2.3 Language and Framework

The FRMCS Agent is built using C and C++, leveraging key external libraries:

- **libcurl**: Manages HTTP/2 communication.
- **nghttp2**: Provides HTTP/2 and Server-Sent Events (SSE) support.
- **cJSON**: Used for JSON parsing and configuration handling.
- **POSIX APIs**: Utilized for networking, threading, and system interactions.

3.1.2.3.1 Services and Major Components

The FRMCS Agent is responsible for managing FRMCS services between:

- **Onboard applications and the onboard gateway**
- **Trackside gateway and the trackside application**
- **Handling HTTP/2-based communication** for session control and notifications
- **Managing service registration, notifications, and session handling**

Both FRMCS Agents does not handle user data transmission, which is managed via a separate GRE tunnel between the onboard and trackside applications.

3.1.2.3.2 Major Components in the Codebase

The FRMCS Agent is structured into models, services, commands, and utilities:

- **Services (`services/`)**: Manages HTTP/2 communication and FRMCS protocol interactions.
  - `agent_service.cpp`: Core logic for communication setup.
  - `registration_service.cpp`: Handles FRMCS domain registration.
  - `session_service.cpp`: Manages session establishment and termination.
  - `notification_service.cpp`: Implements SSE-based event handling.
- **Models (`models/`)**: Defines data structures for FRMCS communication.
  - `incoming_session.cpp`: Handles session requests.
  - `session_status_data.cpp`: Stores session state.
  - `notif_channel.cpp`: Defines notification subscriptions.
- **Commands (`commands/`)**: CLI interface for debugging and control.
- **Utilities (`utils/`)**: Logging, JSON parsing (`cJSON.c`), and helper functions.

3.1.2.3.3 Configuration Details

The FRMCS Agent relies on a structured configuration, which defines communication parameters.

[BASE] Section

- `url`: Specifies the endpoint of the OB_APP server at the onboard gateway.

[OB_APP_1] (Onboard Application Parameters)

- `app_category`: Defines the application type (e.g., ATO, ETCS).
- `static_id`: Unique identifier of the onboard application.
- `comm_type`: Specifies the communication mode (data, video, etc.).
- `comm_level`: Defines the communication priority (basic, critical, etc.).
- `app_local_ip`: IP address of the onboard application.
- `app_interface`: Network interface used for communication.
- `is_sender`: Defines whether this FRMCS Agent instance sends or receives data.

[TS_APP_1] (Trackside Application Parameters)

- Similar structure as `OB_APP_1`, but for the trackside counterpart.

3.1.3 FRMCS Onboard Gateway Redundancy Options

Redundancy is an important functionality in mission critical networks. Currently in the FRMCS specification redundancy is not specified in detail. In the course of this Onboard Platform Demonstrator project, the analysis of FRMCS Onboard Gateway redundancy was identified as an open research topic as per User Story 2.4.4. Note that this analysis was done based only on a theoretical study.

This specific analysis focuses on the FRMCS Onboard Gateway redundancy as such and its interfacing to the application. The redundancy towards the FRMCS network, which covers the FRMCS radio interface, was not part of the analysis.

The interface of the FRMCS Onboard Gateway towards the FRMCS Agent and communication bridge is defined via the OB_{APP} interface as already described in previous deliverables and chapters.

The structure is shown again in the following Figure 24

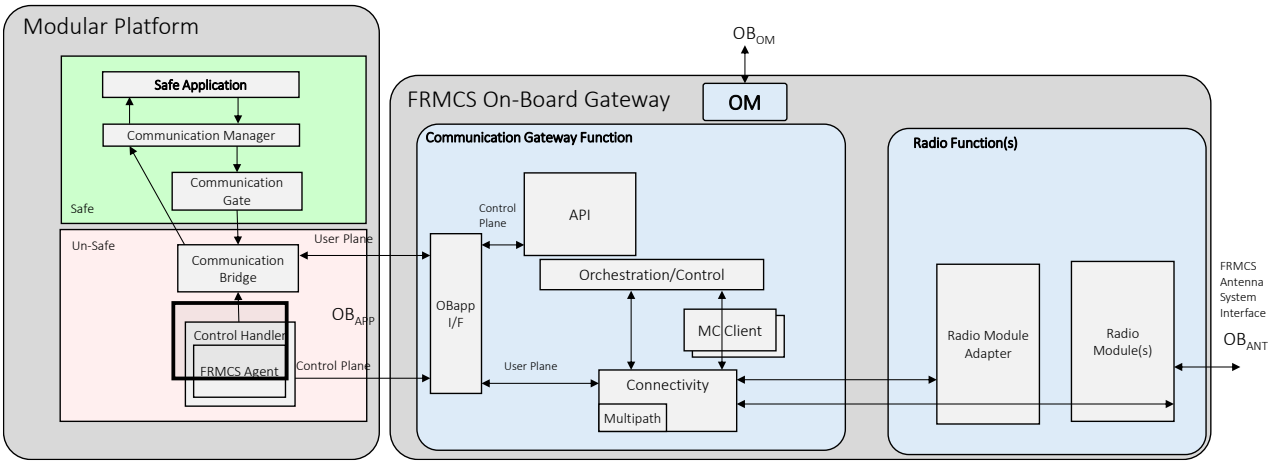


Figure 24: Connection of FRMCS Onboard Gateway

The defined interface between the Application (in this case acted via the FRMCS Agent and the Communication bridge) and the FRMCS Onboard Gateway is OB_{APP}. OB_{APP} is based on HTTP/2 protocol for the control plane and using plain IP for the user plane.

First focus is thus the OB_{APP} interface.

Note: the provided redundancy options are not an exhaustive description of all potentially possible redundancy scenarios, however the main scenarios based on standard

Client/Server interaction in information technology are covered. Due to limitation as per project frame these two scenarios have been investigated.

3.1.3.1 OB$_{APP}$ Redundancy Option 1

The first option for OB$_{APP}$ redundancy is a local link binding between two FRMCS onboard gateways. A schematic graph is outlined in Figure 31.

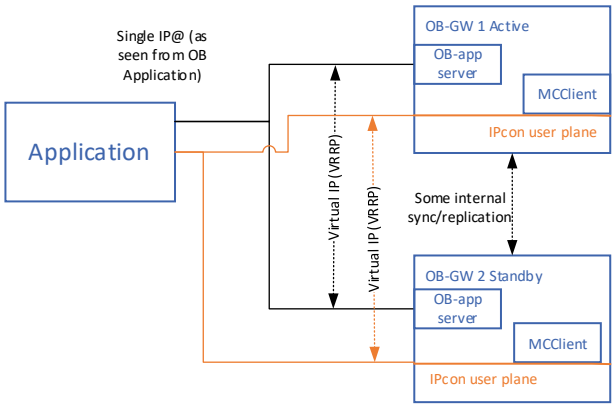


Figure 25: OB$_{APP}$ Redundancy Option 1

The main characteristics are the following:

- Local Link Binding between 2 FRMCS Onboard Gateways: The two FRMCS Onboard Gateways are linked via a standard local link protocol – e.g. VRRP. As such a virtual IP Address is provided to the Application (e.g. EVC or any other application)
- No load balancing is assumed, but active/standby configuration of the FRMCS Onboard Gateways. Optionally there could be some internal sync between the FRMCS Onboard Gateways for data and state replication.

3.1.3.2 OB$_{APP}$ Redundancy Option 2

The second option for OB$_{APP}$ redundancy is two have separate connections from the application side to each of the FRMCS Onboard Gateways. A schematic graph is provided in Figure 26.

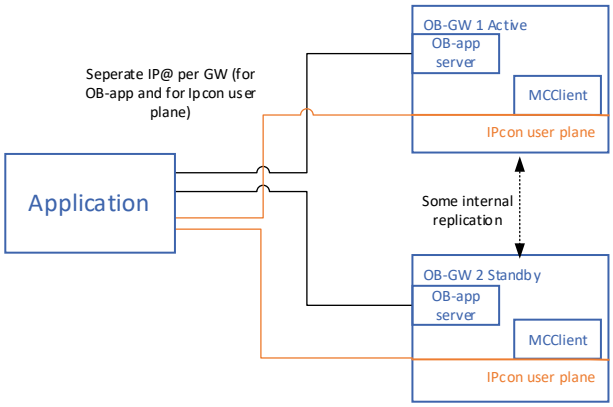


Figure 26: OB$_{APP}$ Redundancy Option 2

The main characteristics are the following:

- Separate address/interfaces locally at application. The application maintains separate links to each FRMCS Onboard Gateway. This means typically also separate IP addressing for each link.
- Application needs to decide which interface it shall use and does perform the selection between the two FRMCS Onboard Gateways. There could be optionally some internal sync between the FRMCS Onboard Gateways for data and state replication.

3.1.3.3 Specific Considerations for Modular Platform

Regarding the specific connection of the components on the Modular Platform within the present project an updated graph shows also the additional components, as example using redundancy Option 1 (Figure 27).

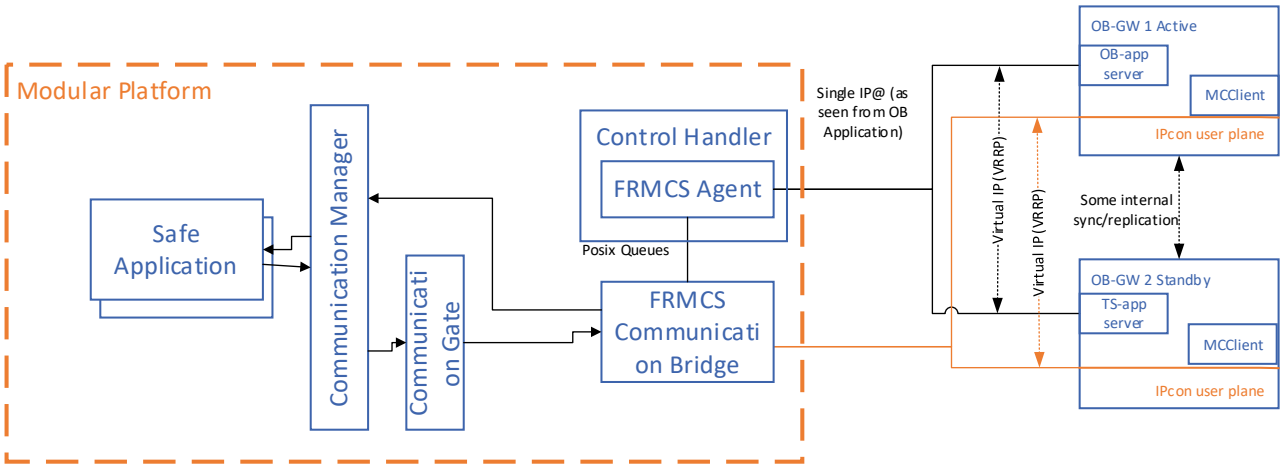


Figure 27: Redundancy Option 1 with the components of the Modular Platform Communication Architecture

Depending on the redundancy option the FRMCS Agent and the FRMCS Communication Bridge need to potentially support redundancy features.

In case of Redundancy Option 1, both, the FRMCS Agent and the Communication Bridge could rely on single interfaces towards the FRMCS Onboard Gateway, and the FRMCS Onboard Gateway takes care of all switchover tasks.

In case of Redundancy Option 2, the FRMCS Agent and the Communication Bridge need to handle and maintain separate connections to both FRMCS Onboard Gateways and thus must include some logic for redundancy and switchover handling.

3.1.3.4 FRMCS Specification Considerations

In addition to the redundancy options defined in previous chapters, a view on the current FRMCS specification for the OB_{APP} interface is beneficial:

The current version of the FRMCS specification stipulates that the OB_{APP} interface does not require session continuity if the connection between the FRMCS Onboard Gateway and OB_{APP} is interrupted for any reason.

This means a Cold-Standby mode is considered from FRMCS Onboard Gateway point of view.

As such the FRMCS Agent and the communication bridge need to do a re-instantiation and re-start from the beginning of the OB_{APP} communication flow in case of redundancy switchover, or any other failure. Therefore, no storage of states or internal session information is necessary – on both sides of the interface.

The same procedure can also be used in case there is an issue on the Modular Platform, e.g. one of the redundant computing elements fails or restarts with a switchover inside the Modular Platform to a redundant computing element. Then the SW components on the Modular Platform can do a clean restart initiating the session from the beginning.

Note that the actual implementation in the Onboard Platform Demonstrator regarding the redundancy in the Modular Platform was based on this setup according to current FRMCS specification.

3.1.4 FRMCS OB-GW Deployment Options

This section aims to provide the base for the theoretical investigation of potential deployment options of the required onboard components including FRMCS Onboard Gateway function in conjunction with the Modular Platform.

From the high-level architectural split provided in Figure 9, all the relevant onboard components can be identified which are needed at the onboard side to achieve safe data communication over FRMCS. This includes the:

- Information source and sink – namely the safe application,
- Components necessary inside the module platform for the communication fulfilling all Modular Platform related functions (Communication Manager, Communication Gate)
- The interface of the Modular Platform derived components to the FRMCS Onboard Gateway (Communication Bridge and FRMCS Agent)
- The actual FRMCS Onboard Gateway providing the FRMCS specification compliant implementation of the FRMCS onboard functions. The functions of the FRMCS Onboard Gateway are defined in [6].

A graphical representation of these onboard components and functions is shown in the following figure:

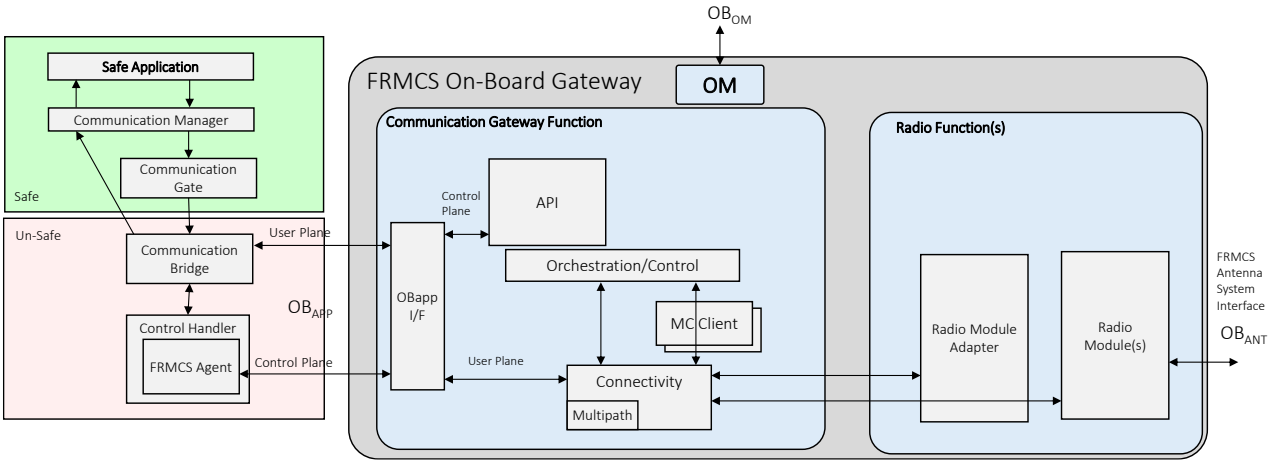


Figure 28: Functional view of all required onboard components to achieve data communication from a safe application via FRMCS

Starting from Figure 28, under the assumption that

- The Modular Platform as base element shall be used for the safe application
- Modular Platform specific functions for communication and redundancy shall be used

different deployment options can be derived. The deployment options have been chosen to split the functions in manner that as much as possible defined (or at least intended to be defined) interfaces are used for the split to come to somehow realistic scenarios for productive deployment also from standard compliancy and commercial perspective thus avoiding the need for any additional, potentially proprietary definition of interfaces.

The next chapters will describe three different deployment scenarios which will be analysed finally in Chapter 4.1.3 , respectively Chapter 4.1.3.2.

3.1.4.1 Deployment Scenario 1

The first deployment scenario assumes that the FRMCS Onboard Gateway is a fully separated entity and not deployed on the Modular Platform.

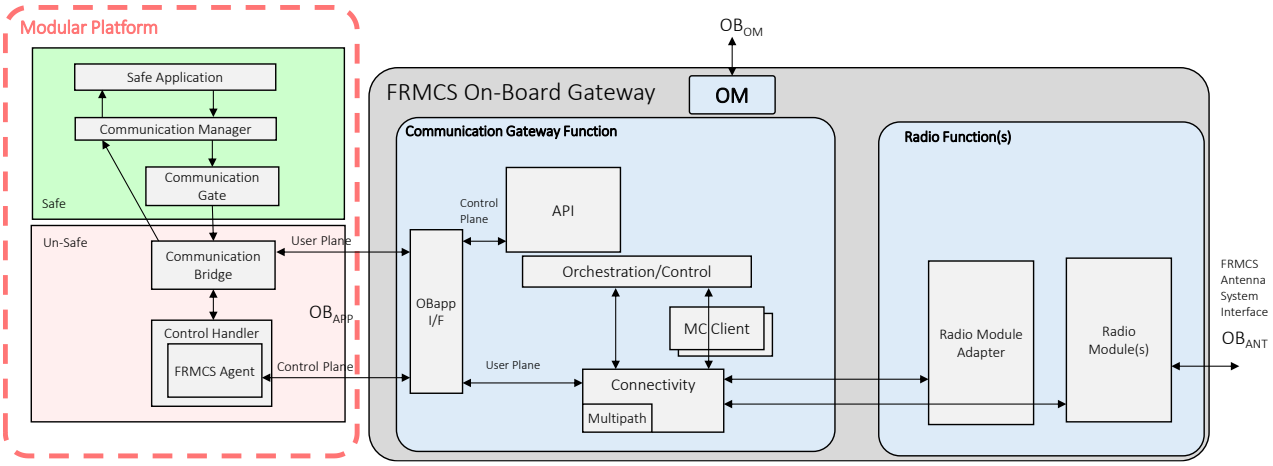


Figure 29: FRMCS OB-GW deployment scenario 1

The Modular Platform hosts “only” the functions which have been defined in the assumptions above (from Safe Application on to the Communication Bridge and FRMCS Agent). The FRMCS Onboard Gateway is connected via the FRMCS defined OB_{APP} interface. The physical interface between the Modular Platform and the FRMCS Onboard gateway is most probably an ethernet type of network interface (as per requirements in [6]).

The FRMCS Onboard Gateway is deployed on separate physical hardware module, including all the radio modules responsible for the FRMCS antenna system interface connection via OB_{ANT} (right hand side in Figure 29).

Note that internal FRMCS Onboard Gateway modularity is still very well possible and beneficial but not assessed here further as mainly the modularity and deployment with respect to the Modular Platform is investigated.

3.1.4.2 Deployment Scenario 2

The second deployment scenario enhances the scope of the Modular Platform. Functions from the FRMCS Onboard Gateway which cover the so called “Communication Gateway Function” are deployed on the Modular Platform.

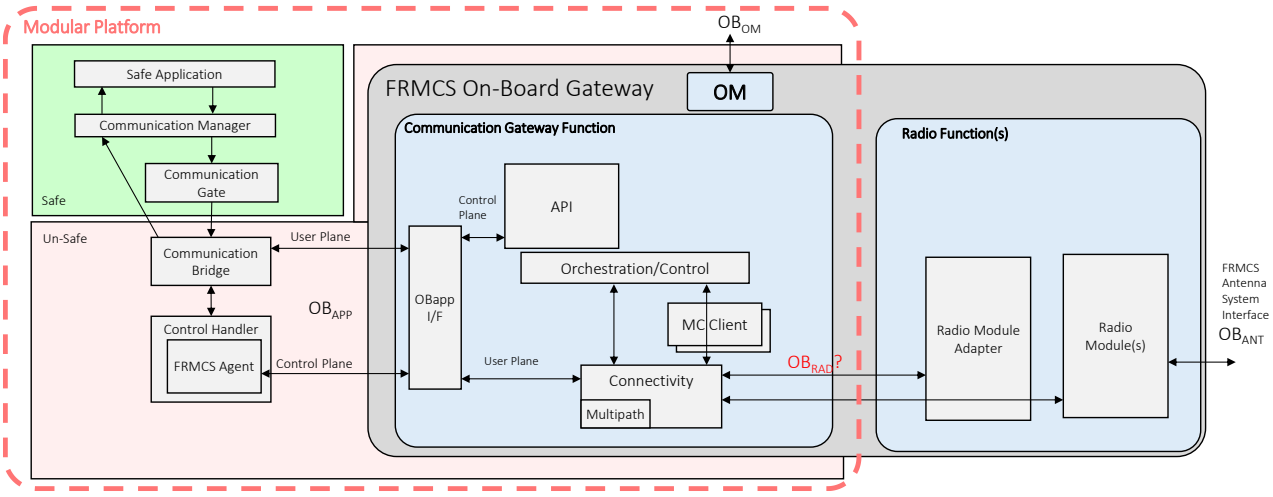


Figure 30: FRMCS Onboard Gateway deployment scenario 2

The functions inside the “Communication Gateway Function” of the FRMCS Onboard Gateway are functions without any hardware dependency.

The rest of the FRMCS Onboard Gateway covering the “Radio Function(s)” would remain on a separate entity.

The interface between the “Communication Gateway Function” and the “Radio Function(s)” is defined as OB_{RAD}. Interface specification is ongoing in ETSI RT. ETSI TR 104 006 provides initial input. Thus, the feasibility to implement this scenario is given due to

- the hardware independence of the “Communication Gateway Function” and
- the potential availability of a standardized interface to the “Radio Function(s)”

3.1.4.3 Deployment Scenario 3

The third deployment scenario even further extends the scope of the Modular Platform where the entire FRMCS Onboard Gateway is integrated into the Modular Platform.

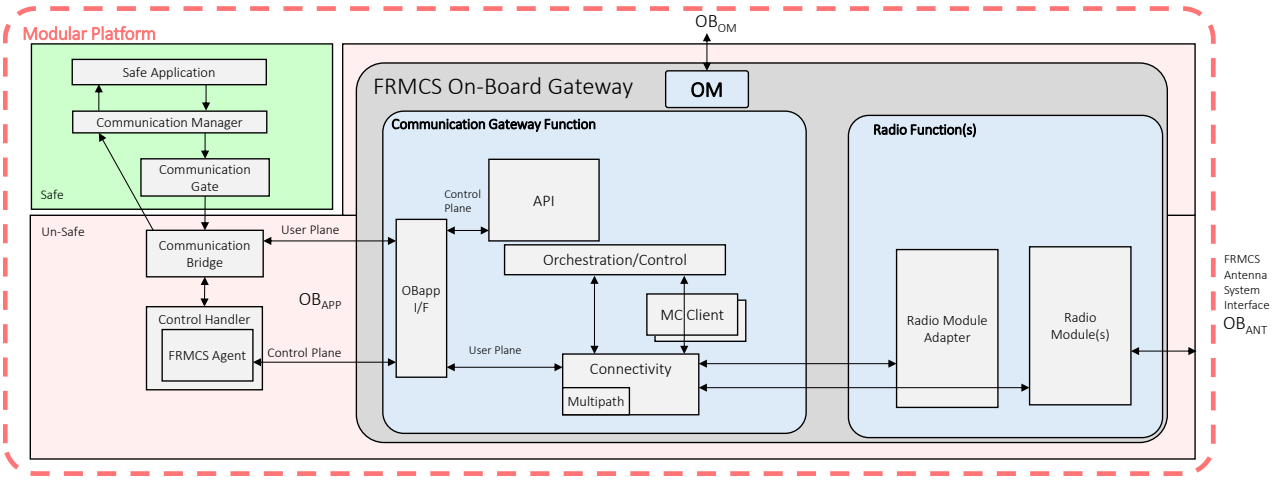


Figure 31: FRMCS Onboard Gateway deployment scenario 3

In addition to the deployment scenario 2, the scenario 3 covers all the Radio Functions of the FRMCS Onboard Gateway inside the Modular Platform.

As such all the interfaces inside the FRMCS Onboard Gateway are inside the Modular Platform and need not to be externalized.

However, in contrast, the OB_{ANT} Interface (FRMCS Antenna System Interface) is then the external interface to be implemented on the Modular Platform. This has the following implications:

- Radio Functions typically implemented on hardware radio modules with standardized low-level interfaces need to be supported on the Modular Platform. This means normally that the hardware interfaces as well as software interfaces of these radio modules need to be available on the Modular Platform (e.g. M.2 form factor with PCI or USB interface). The internal interface can also be provided by the radio module adapter – but still to be supported on the Modular Platform.
- External Radio interfaces (antenna connections) need to be supported to provide the radio connection from the external antenna to the internal radio module.

The other functions are also hosted on the Modular Platform in the same way as in the deployment scenario 2.

3.2 FUNCTIONAL CLUSTER DIAGNOSTICS

The implementation of MCP-DIA is foreseen to be realised in a stepwise approach, introducing its basic functionality and initial integration into the SBB-laboratory as part of this deliverable. Chapter 2.2 of this document covers most of the WP36 MCP-DIA architecture and features already but might be complemented in the final implementation task's upcoming deliverable D36.4 by design updates.

For details on MCP-DIA related User Stories please refer to D36.1 [2], chapter 2.5. Chapter 4.2 of this deliverable provides evidence and test results for the User Stories that could be realised so far.

3.2.1 MCP-DIA (A-Prototype Delivery)

The implementation of MCP-DIA in R2DATO is supposed to cover multiple aspects and needs of diagnostics for Modular Platforms. The realisation and demonstration of MCP-DIA is the key objective of this SOVD-based initial implementation. Though the context of a research project yields to limitations according to the realised feature-set and quality, DB as the contributor tried to find an adequate compromise between the spent effort and resulting outcome, represented by a targeted TRL of 5. The implementation has been specified und subcontracted as a three step A-, B- and C-Sample approach, in which the A-Prototype already provides a solid basis but still needs further extension and improvement. Anyhow we're happy to present here a first comprehensive feature-set and reliable implementation, driven by a reproducible and automated DevOps utilisation:

- Realisation of the SOVD Request-Response pattern and accompanying data endpoints, as introduced in chapter 2.2.2.1.
- Provision of endpoints for logging and bulk-data (please refer to chapter 2.2.2.6)
- SOVD server and SOVD Server API as pre-compiled artefacts for the defined target platform
  - artefacts are delivered including test coverage reports
  - Utilised APIs are configurable and extendable, provision of a separate toolchain for message format authoring (as explained later in chapter 3.2.2.3)
  - Build Integration for Protobuf, compatible and tested with the TAS toolchain
  - SOVD-Server accessible over the network, utilising bulk traffic network in the laboratory
- SystemHealthStatusApp (provided in source code) as a reference BI Application implementation, utilising the API and Protobuf to the SOVD-Server component (as explained in chapter 3.2.2.2)
  - Realised, configured and integrated as a Model 3 TaskSet
  - The Monitoring client can parse system information (Host-OS information and CPU usage) of the target HW as it is deployed on top of the TAS runtime environment
- MDCM client as a Mock-up application
  - The MDCM Mock-up utilises the SOVD REST API to visualise data acquired by the SOVD-Server and the Vehicle Health Monitoring BI Application

- Deployment of all software components to the laboratory using automized Azure DevOps Pipelines
  - The SystemHealthStatusApp client gets built in a Cloud VM by the Azure DevOps Pipeline, compiled with the provided TAS platform toolchain incorporating Protobuf as statically linked library
  - All software components are integrated and configured with appropriate scripts and configuration files (as cs.conf) and automatically started by appstart on one Computing Element of the target hardware (in the sense of User Story 2.1.5 “Execute Declarative Configuration”)
- Documentation is provided (as a design concept, API description), Integration and User Manual and appropriate Testing
- Delimitations:
  - Limited to the MCP-DIA feature-set as health- and performance data acquisition, a full SOVD implementation covering all SOVD endpoints is out of scope.
  - Software Configuration and Software Updates are not in scope
  - No security protocols (plain HTTP) between MDCM and MCP-DIA

3.2.2 Current Status of the Implementation

All features of the A-Prototype have been realised as specified above and the resulting artifacts are available. All corresponding software has been specified, developed, automatically build, deployed and tested successfully in the Azure DevOps environment and integrated onto the lab-server and the target hardware running into the SBB-laboratory as introduced in chapter 2.3.

The following diagram provides a high-level deployment overview.

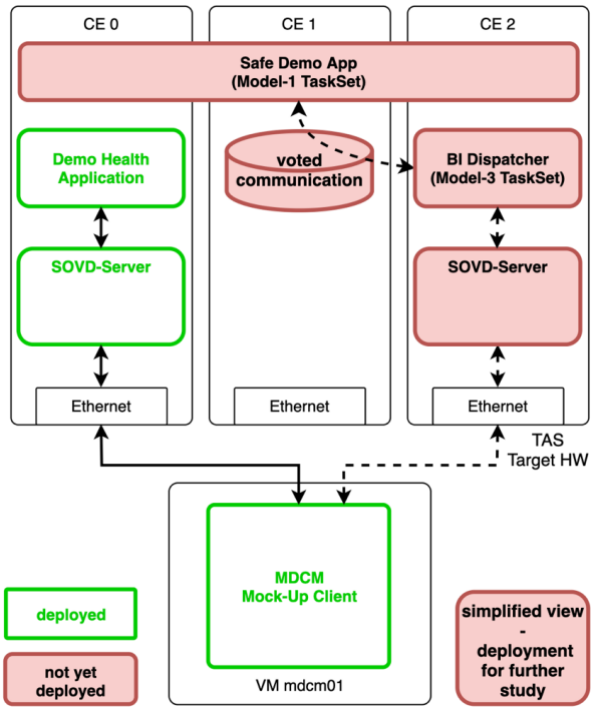


Figure 32: Deployed MCP-DIA Components in SBB-laboratory

The green components have already been integrated into the SBB-laboratory, red components are planned for the next deliverable and are subject of further change based on future design decisions. The Azure Cloud and the self-hosted Deployment Agent realised in the laboratory have been omitted in the diagram, including the utilised cloud-based repositories, the containerised build environment and pipelines that will be described in the next section.

3.2.2.1 Automated Build, Integration and Testing

The Azure build and deployment is organised in three stages that can individually be selected to be run. This provides multiple advantages: We can decide either just to build software, perform automated deployment and testing by finally leaving the target in a cleaned-up end state (all stages), or skipping the clean-up, keeping the hardware targets fully deployed with functional SOVD-Server and applications. This option enables us to use the MDCM Mock-Up e.g., for showcasing or manual testing obtaining and visualising diagnostics data as described in chapter 3.2.2.2.

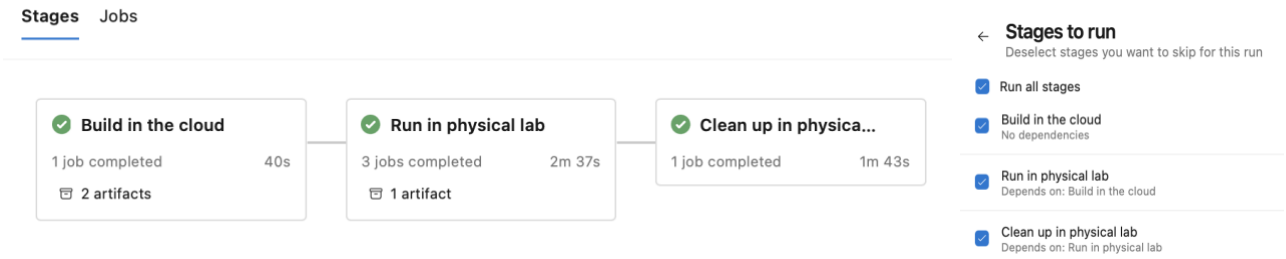


Figure 33: Azure DevOps Pipeline and Stages

A full DevOps pipeline run results currently in 5 pipeline jobs. The first job for building from source code is executed in the cloud while all other jobs trigger actions that get directly applied to the SSB-laboratory, utilising an Azure self-hosted agent deployed on a virtual machine on the SBB-laboratory server (see also chapter 2.3.1.8). The following diagram provides an overview of those 5 jobs, pointing out the main advantage of automisation, reproducibility and optimisation of deployment cycles time. Jobs get either manually triggered or as defined by shedule or trigger events as a change in a git repository.

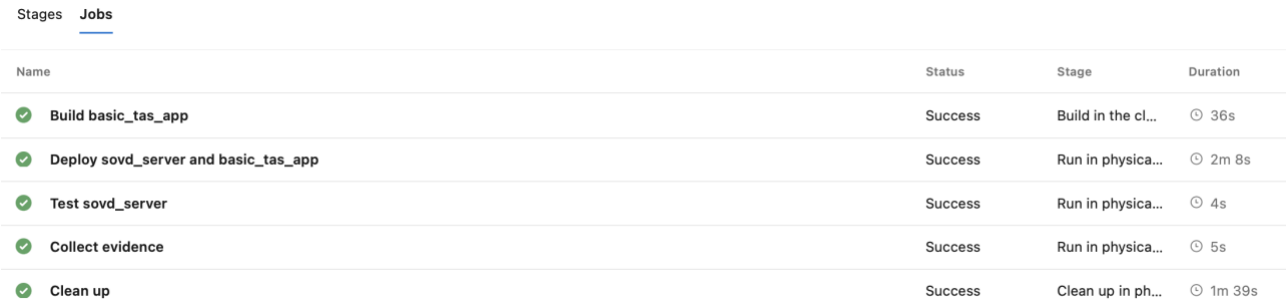


Figure 34: Jobs of the Azure DevOps Pipeline

In case there hasn't been a change on source code level and no clean build is needed, the pipeline will automatically utilise artefacts from its cache and reduces a full deployment to the laboratory, automated testing and cleanup currently to less than 5 minutes (including two restarts of the physical target system which is accounted for by a waiting time of 90

seconds each). A clean build gets triggered either automatically by a commit to the source repository or in case relevant binary artifacts have changed as API declarations or the sovd_server itself, that gets integrated as a binary.

The following diagram depics the advantage of the utilisation of caches, which reduces the duration of the build job from almost 20 minutes to less than one minute. Both provide all artifacts to be deployed onto the target hardware.

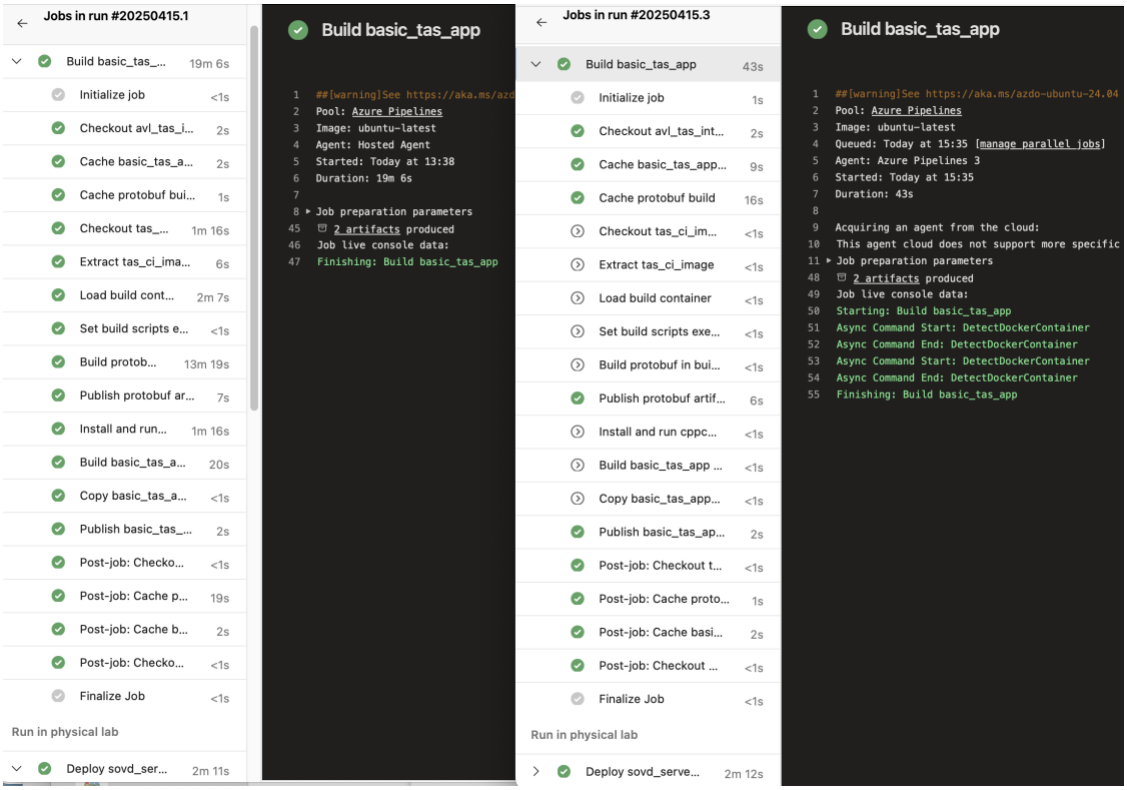


Figure 35: Clean build vs. utilisation of caching

The automated build process collects and provides artifacts (e.g., system and application logs from the target system) that can directly be accessed and downloaded over the Azure DevOps web-interface.

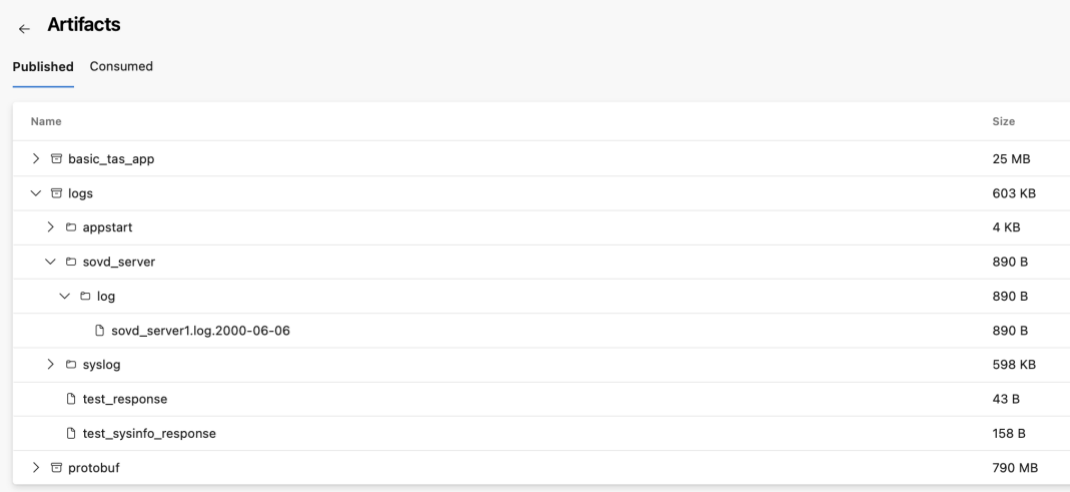


Figure 36: Azure DevOps collected and provided artifacts

3.2.2.2 SystemHealthStatusApp Example Client Implementation

As introduced in chapter 3.2.1 and represented in Figure 32: Deployed MCP-DIA Components in SBB-laboratory, the A-Prototype delivery comprises a demo health application that is provided in source code as a reference implementation, utilising the SOVD-Server API. This Model 3 TaskSet is serving as a blueprint how to realise a diagnosed application, running on top of the TAS platform runtime environment.

The SystemHealthStatusApp provides as an example some diagnostic faults and system information (Host-OS information and CPU usage) of the target HW.

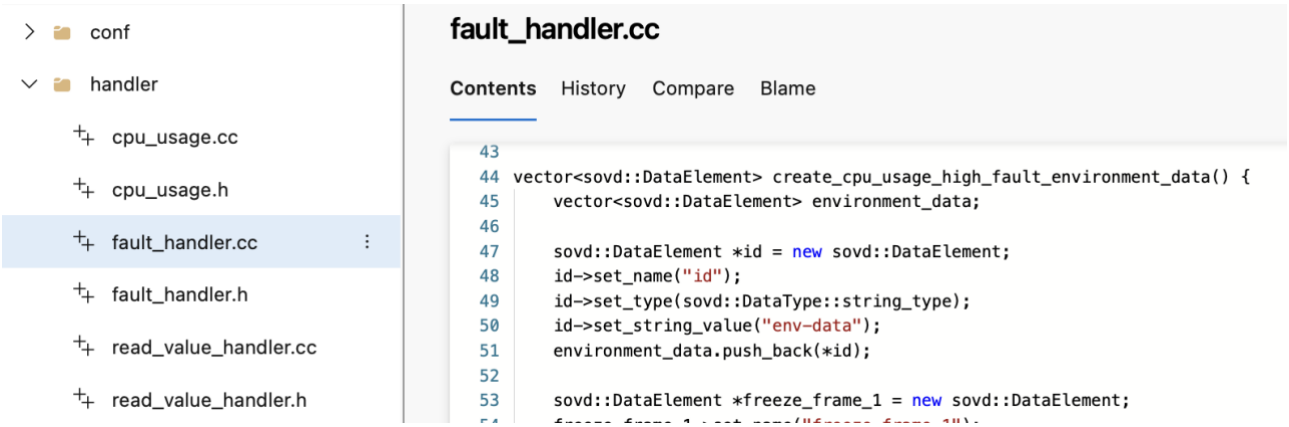


Figure 37: SOVD Fault Handler provided as source code example

3.2.2.3 SOVD Authoring Tool

The MCP-DIA SOVD-Server can provide generic and use-case specific functionality with a tailorable API-surface that can be adapted to specific needs of diagnostics clients. These dynamic and flexible structures and behaviours of MCP-DIA are achieved by the provisioning of binary definition files at compile and runtime.

A definition file contains the structure of the server with detailed information about the SOVD entities to be utilised in clients, such as the demo SystemHealthStatusApp, the BI Dispatcher (Model-3 TaskSet) and Safe Demo App (Model-1 TaskSet).

This includes for instance possible faults and fault-details, readable measurements like Host-OS information, CPU and memory usage for health monitoring, log storage, etc.

The SOVD Authoring Tool provides comprehensive support for creating those binary definition files. It facilitates a straightforward definition process using the domain-specific language "SOVD Definition Language" (SDL).

3.2.2.3.1 Example Workflow for the "SystemHealthStatusApp"

The structure of the SOVD Entity "SystemHealthStatusApp" is specified as SOVD Application in an SDL file. Possible faults, such as "cpu_load_high" and "free_memory_percent_low" are defined along with their fault codes and names. If a specific fault occurs during runtime, the "environment-data" will be populated with concrete information about each fault occurrence when queried through the REST-API.

Measurements like System Information, Memory Usage and CPU Load are defined as Read Values and can be queried using the GET /{entity-path}/data endpoint afterwards.

Additionally, data types, their units and ranges can be specified in SDL in a dedicated SOVD Authoring Tool, that is provided as a Visual Studio Code Editor Plugin.

SDL offers many more language features which allow the definition of an entire full-fledged SOVD-Server and its entities.

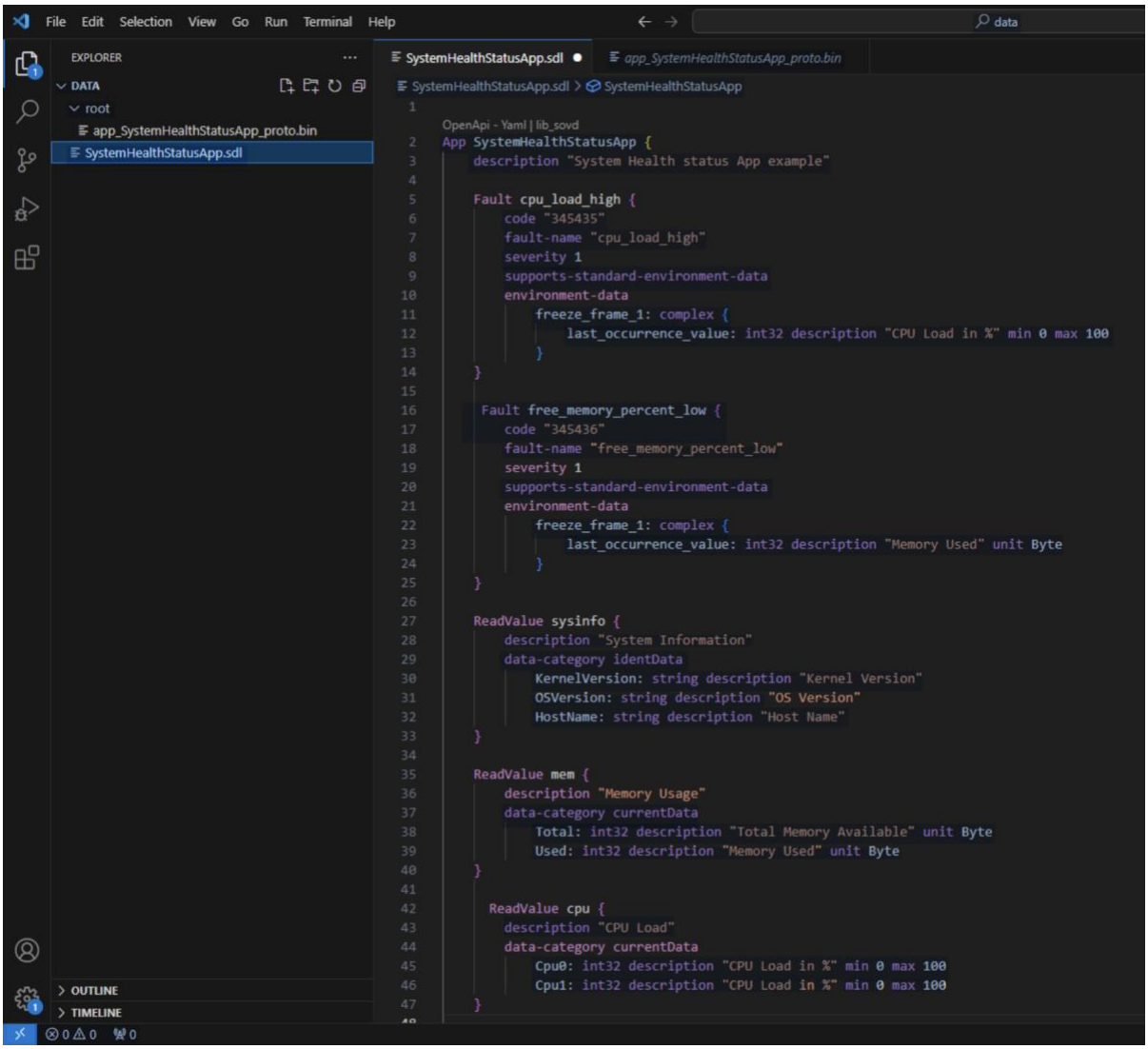


Figure 38: SDL example for the SystemHealthStatusApp

The generation of binary definition files can be triggered in the Authoring Tool based on an SDL file. It then uses the tooling interface of a running SOVD-Server instance to create the actual binary. The created binary can be found in the directory next to the SDL file after the creation process is completed (see app_SystemHealthStatusApp_proto.bin in the screenshot above).

3.2.2.4 Usage of MDCM Mock-Up Client for Show-casing and Manual Testing

The A-sample deliverable contains a first version of a MDCM Mock-up. This SOVD client provides a graphical user interface and is mainly intended for show-casing easy access to diagnostics data. It is an essential feature, that the MDCM-Mockup is based on a generic implementation that dicovers and builds up its structure, content and views upon the available configuration of the SOVD-Server, defined with the SOVD Authoring Tool.

This approach provides a straightforward utilisation of the open REST-API of the SOVD-Server and therefore enables client implementations to be easily integrated in other MDCM realisations that typically would be train-borne headless data aggregators.

As shown in Figure 39 network traffic towards the SOVD-Server can be analysed with the Mock-up as well. Based on the provided configuration, the MDCM Mock-Up provides multiple views to visualise the acquired diagnostics data. We present here just a limited number of them to provide a first impression.

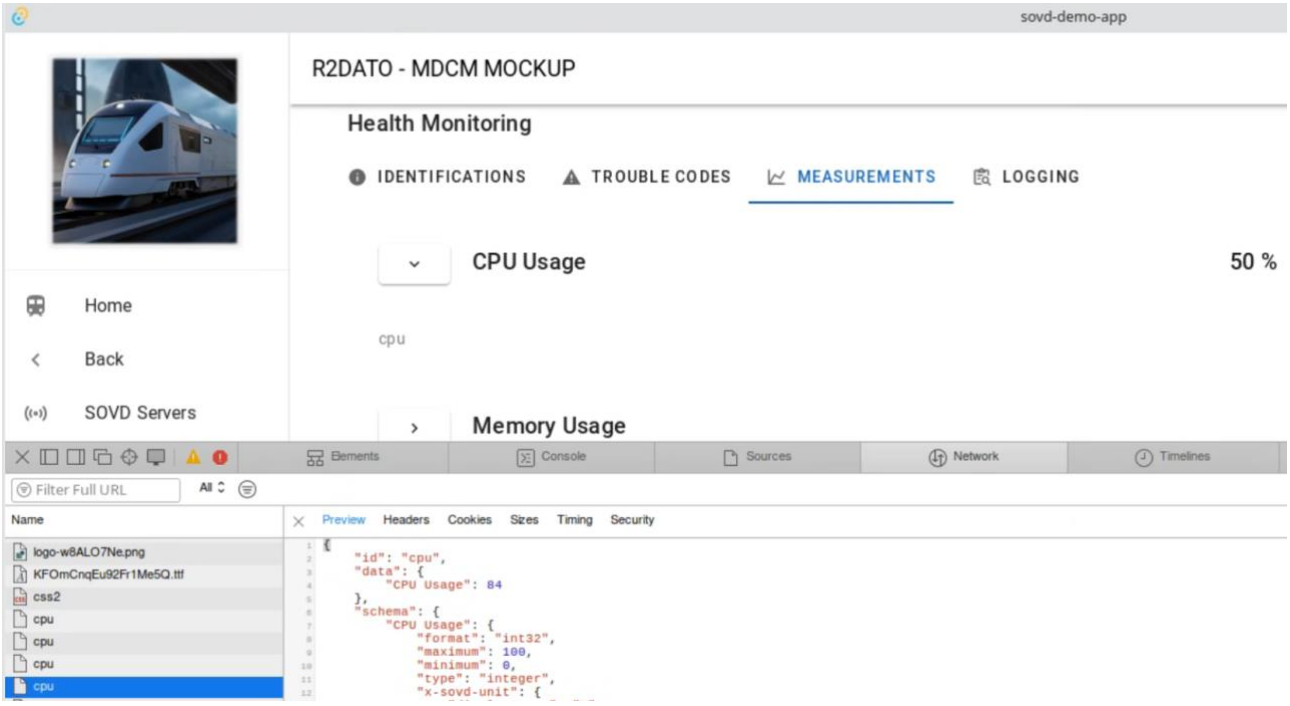


Figure 39: MDCM Mock-Up network analysis

The source of the visualised data is the integrated SystemHealthStatusApp demo application. So, these views are representing a full end-to-end data flow based on the Request-Response pattern.

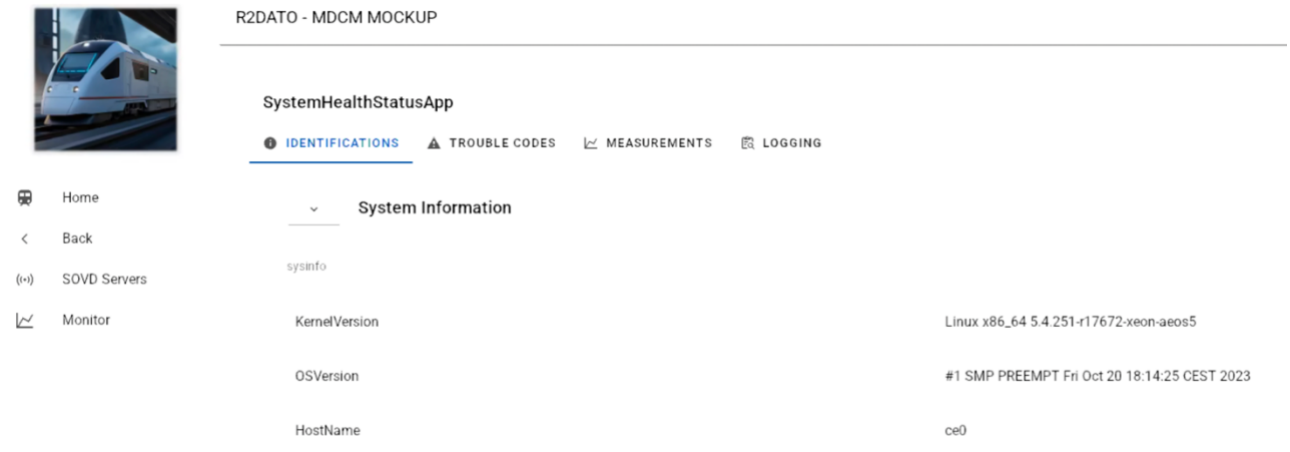


Figure 40: MDCM Mock-Up System Information CE0

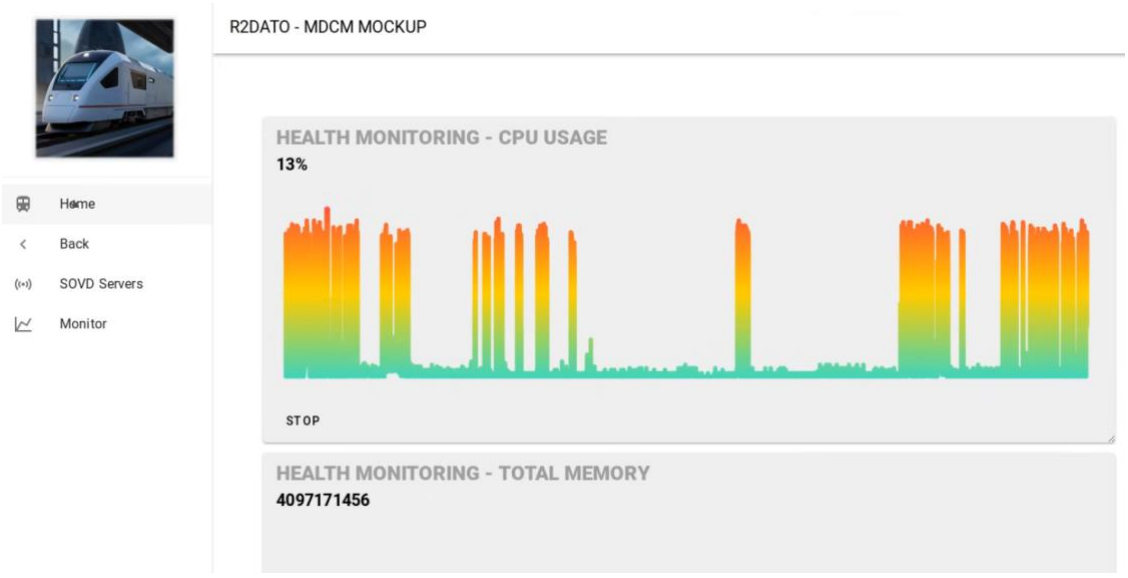


Figure 41: MDCM Mock-up CPU usage visualisation

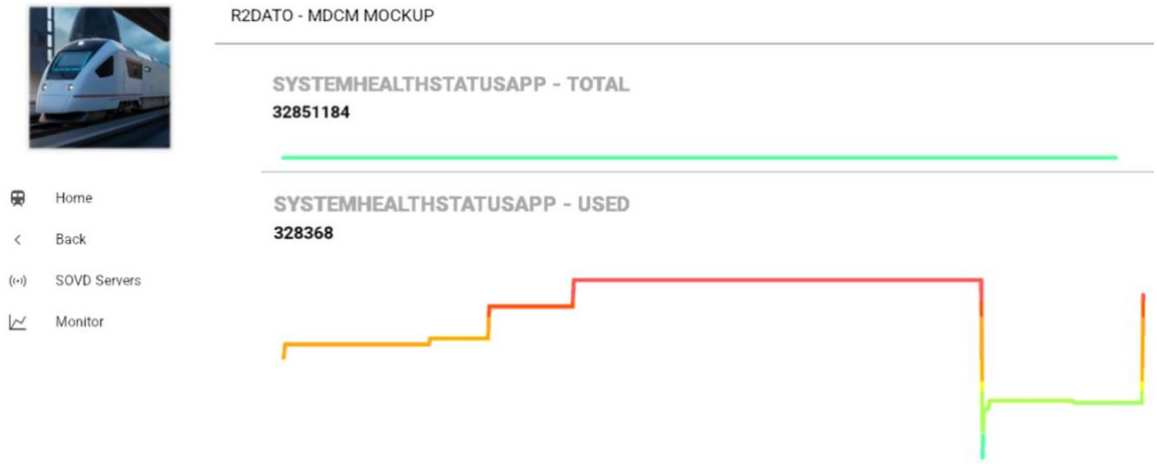


Figure 42: MDCM Mock-up memory usage

Diagnostic faults are shown in a table format, additional details can be obtained issuing Requests over the REST API for each stored fault.

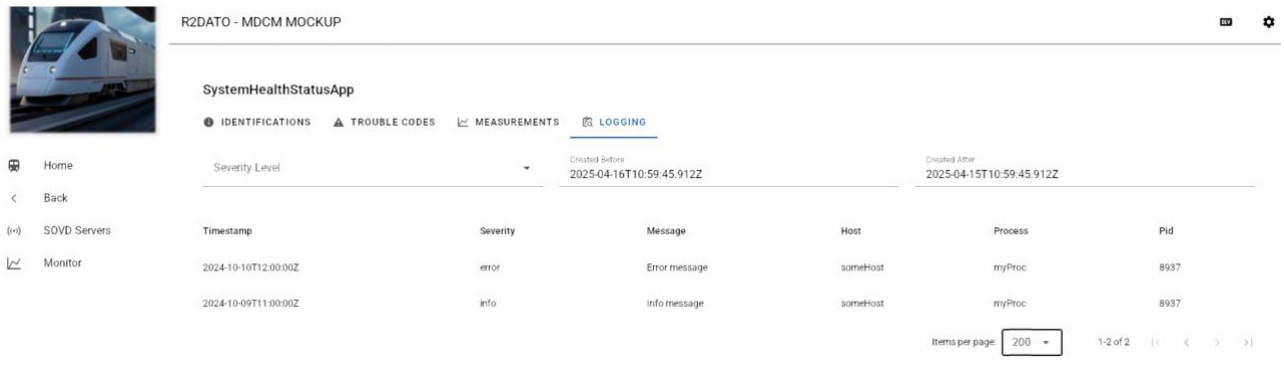


Figure 43: MDCM Mock-up diagnostics fault representation

3.2.3 Open Topics to Be Realised or Finalised

The following topics are not yet implemented or finalised and shall be provided later:

Number	Topic	Status
MCP-DIA OT-1	Cache component	Concept ready, implementation not finished yet
MCP-DIA OT-2	SOVD subscriptions and cyclic data	Needs further study and more detailed specification input from the SP Transversal domain
MCP-DIA OT-3	Syslog-ng utilisation for logging on local Computing Element	Concept ready, implementation not finished yet
MCP-DIA OT-4	Syslog-ng utilisation for logging on other CEs	Needs further evaluation. The idea is to be able to collect, distribute and merge syslog data from all three Computing Elements of the Computing Node
MCP-DIA OT-5	BI Dispatcher Application	First prototype available, will be integrated and tested in the laboratory in Task 36.4
MCP-DIA OT-6	Multiple MCP-DIA instances for redundancy (deployment on different CEs)	Needs further evaluation. The idea is to have SOVD-server instances always running on at least one CE, not missing the ability to record diagnostics data even if the CE where the active server is running needs a reboot.

Table 3: Open MCP-DIA topics

4 TESTS AND RESULTS

This chapter describes the test plans defined to validate the implemented user stories. The results are covering as such both, the selected user stories and the implemented functions.

The aim has been providing all sufficient information to be able to reproduce the tests results in similar and extended environments.

4.1 FUNCTIONAL CLUSTER FRMCS

In this chapter the test cases and results for the Functional Cluster FRMCS are included.

4.1.1 User Story 2.4.1: Comm. over FRMCS Transparent to the Application

4.1.1.1 Description of the User Story

4.1.1.1.1 References

EROPD-23, EROPD-45, EROPD-58

4.1.1.1.2 Description

As R2DATO WP36, we want an application on the Modular Platform (and an application on the safety layer / runtime environment) to communicate over FRMCS to a trackside entity, so that we can demonstrate an FRMCS communication that is transparent to the application.

4.1.1.1.3 Related Requirements

SUBSET-147-3.1.1.3, FRMCS-FRS-11, FRMCS-SRS-6.4.1, TOBA-FRS-7.1

4.1.1.1.4 Test Cases and Results

ID: FRMCS_AUTH_ONBOARD

Summary:
This test verifies that the **onboard MCx client** successfully authenticates to **FRMCS services**.

Purpose:
Ensure that the **FRMCS Wi-Fi network** functions correctly and allows authentication of the **onboard MCx client**, ensuring seamless communication.

Priority:
High

Test Environment:

- **Hardware: OB_GTW** interfaced with a **Wi-Fi Radio Module**
- **Network: FRMCS Wi-Fi** with connectivity to the **MCx server**
- **Tools:** Network tracing utilities (**Wireshark, tcpdump**)

Pre-requisites:

- **OB_GTW** must be connected to a **Wi-Fi Radio Module**.

- The **Wi-Fi network** must have access to the **MCx server**.

Test Steps:

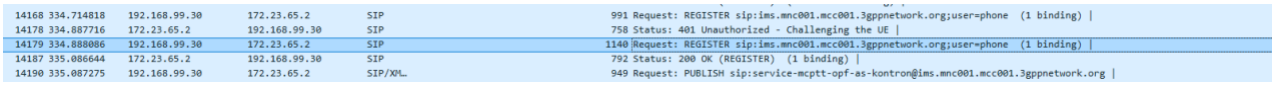
1. **Enable Wi-Fi coverage** in the onboard environment.
2. From **OB_GTW**, send a **ping request** to the **MCx server** to verify connectivity.
3. Configure the **MCx client** on the **onboard gateway**.
4. Start a network tracing tool (**Wireshark/tcpdump**) on the interface handling **MCx server communication**.
5. Start the **onboard super loose service**.
6. Capture network traces and **save the trace file** for analysis.

Expected Results:

- The **MCx client** must successfully authenticate with the **IdMS** and obtain a **JWT token**.
- The **MCx client** must correctly authenticate with the **SIP server**.
- **SIP registration** should return **200 OK**.
- **SIP publish** should return **200 OK**.

Pass/Fail Criteria:

- **Pass**



ID: FRMCS_AUTH_TRACKSIDE

Summary:

This test verifies that the **trackside MCx client** successfully authenticates to **FRMCS services**.

Purpose:

Ensure that the **FRMCS core network** functions correctly and allows authentication of the **trackside MCx client**, ensuring reliable network communication.

Priority:

High

Test Environment:

- **Hardware: TS_GTW** interfaced with the **core network**

- **Network: TS_GTW** with connectivity to the **MCx server** via the core network interface
- **Tools:** Network tracing utilities (**Wireshark, tcpdump**)

Pre-requisites:

- **TS_GTW** must be connected to the **core network**.
- The **core network interface** must have access to the **MCx server**.

Test Steps:

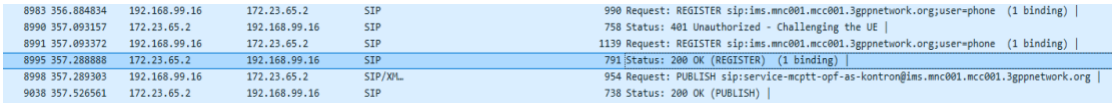
1. From **TS_GTW**, send a **ping request** to the **MCx server** to verify connectivity.
2. Configure the **MCx client** on the **trackside gateway**.
3. Start a network tracing tool (**Wireshark/tcpdump**) on the interface handling **MCx server communication**.
4. Start the **trackside super loose service**.
5. Capture network traces and save the trace file for analysis.

Expected Results:

- The **MCx client** must successfully authenticate with the **IdMS** and obtain a **JWT token**.
- The **MCx client** must correctly authenticate with the **SIP server**.
- **SIP registration** should return **200 OK**.
- **SIP publish** should return **200 OK**.

Pass/Fail Criteria:

- **Pass**



ID: FRMCS_AGENT_AUTH_OB

Summary:
This test verifies that the onboard FRMCS Agent successfully authenticates to FRMCS services via the onboard gateway using OB$_{APP}$.

Purpose:
Ensure that the onboard **FRMCS Agent** functions correctly and supports authentication through **FRMCS OB$_{APP}$**, confirming end-to-end connectivity and protocol integrity.

Priority:
High

Test Environment:

- **Hardware:** Onboard Gateway (OB_GTW) interfaced with a **Wi-Fi Radio Module**
- **Network: FRMCS Wi-Fi** with connectivity to the **MCx server**
- **Tools:** Network tracing utilities (Wireshark, tcpdump)
- **TAS Platform:** Onboard FRMCS Agent running on **TAS**

Pre-requisites:

- The **OB_GTW** must be connected to a **Wi-Fi Radio Module**.
- The **Wi-Fi network** must have access to the **MCx server**.
- **TAS** must be connected to the **onboard gateway** via local or wide-area network.

Test Steps:

1. Enable **Wi-Fi coverage** on the onboard system.
2. From the **OB_GTW**, send a ping request to the **MCx server** to verify network connectivity.
3. From the **OB FRMCS Agent** on **TAS**, send a ping request to the **OB_GTW** to verify connectivity.
4. Configure the **MCx client** on the **onboard gateway**.
5. Start a network tracing tool (**Wireshark/tcpdump**) on the interface handling **MCx server communication**.
6. From the **TAS**, send a ping request to the **onboard gateway** to verify connectivity.
7. Start **FRMCS services** on the **onboard gateway**.
8. Start the **FRMCS local binding** on the **TAS platform**.
9. Capture network traces on both the **TAS platform** and the **onboard gateway**.

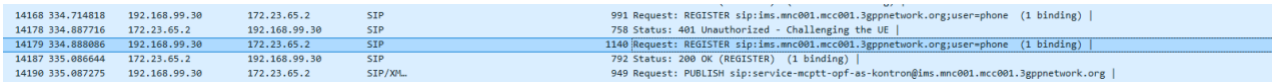
10. Save the trace files for detailed analysis.

Expected Results:

- The **MCx client** must successfully authenticate with the **IdMS** and obtain a **JWT token**.
- The **MCx client** must correctly authenticate with the **SIP server**.
- **SIP registration** should return **200 OK**.
- **SIP publish** should return **200 OK**.
- The onboard system must receive an **OB$_{APP}$ REGISTER 200 OK** response.

Pass/Fail Criteria:

- **Pass**



ID: FRMCS_AGENT_TRACKSIDE

Summary:
This test validates the successful authentication of the trackside agent to FRMCS services.

Purpose:
Ensure that the FRMCS core network is fully operational and supports the authentication of the trackside MCx client, verifying seamless connectivity and protocol adherence.

Priority:
High

Test Environment:

- **Hardware:** Trackside Gateway (TS_GTW) interfaced with the core network
- **Network:** TS_GTW with connectivity to the MCx server via the core network interface
- **Tools:** Network tracing utilities (Wireshark, tcpdump)

Pre-requisites:

- The TS_GTW must be connected to the core network.
- The interface to the core network must have access to the MCx server.

Test Steps:

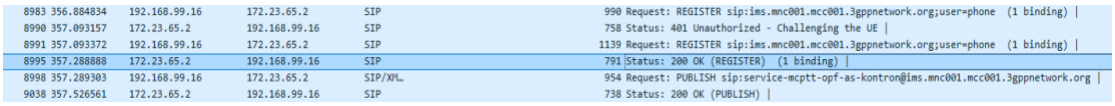
1. From the **TS FRMCS Agent**, send a ping request to the **TS_GTW** to verify basic network connectivity.
2. From the **TS_GTW**, send a ping request to the **MCx Server** via the core network to confirm accessibility.
3. Configure the **MCx client** on the **trackside gateway (TS_GTW)**.
4. Start a network tracing tool (Wireshark/tcpdump) on the interface handling MCx server communication.
5. Start the **FRMCS services** on the **TS_GTW**.
6. Start the **FRMCS local binding** on the **TS FRMCS Agent**.
7. Capture network traces on both **TS FRMCS Agent** and **TS_GTW**.
8. Save the trace files for in-depth analysis.

Expected Results:

- The **MCx client** should successfully authenticate with the **IdMS** and obtain a **JWT token**.
- The **MCx client** should correctly authenticate with the **SIP server**.
- **SIP registration** should return **200 OK**.
- **SIP publish** should return **200 OK**.

Pass/Fail Criteria:

- **Pass**



ID: FRMCS_APPS

Summary:

This test verifies that the onboard application (OB APP) running TAS can successfully communicate with the Trackside application (TS APP).

Purpose:

Ensure that an application on the Modular Platform (including an application on the safety layer/runtime environment) can communicate over FRMCS to a trackside

entity, demonstrating an FRMCS communication that remains transparent to the application.

Priority: High

Test Environment:

- **Hardware:** TS_GTW interfaced with core network, OB_GTW interfaced with a Wi-Fi Radio Module.
- **Network:** TS_GTW with connectivity to the MCx server via core network interface, FRMCS Wi-Fi with connectivity to the MCx server.
- **Tools:** Network tracing utilities (Wireshark, tcpdump)
- **TAS_PLATFORM:** Onboard agent running on TAS, SAFE APP.

Pre-requisites:

- TS_GTW must be connected to the core network.
- The interface to the core network must have access to the MCx server.

Steps:

1. From the TS FRMCS AGENT, send a ping request to the TS GTW to verify connectivity.
2. From the TS GTW, send a ping request to the TS MCx Server to verify connectivity.
3. Configure the MCx client on the trackside gateway.
4. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication and OB_{APP} interface.
5. Start the FRMS services on TS GTW.
6. Start the FRMS local binding on TS FRMCS AGENT.
7. Capture traces on both TS FRMCS AGENT and TS GTW.
8. Enable Wi-Fi coverage on onboard.
9. From the onboard gateway (OB_GTW), send a ping request to the MCx server to verify connectivity.
10. From the OB_FRMCS AGENT on TAS, send a ping request to the OB_GTW to verify connectivity.
11. Configure the MCx client on the onboard gateway.
12. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication.
13. From the TAS, send a ping request to the onboard gateway to verify connectivity.
14. Start FRMS services on onboard.

15. Start the trackside application.
16. Start the SAFE APP on the TAS PLATFORM.

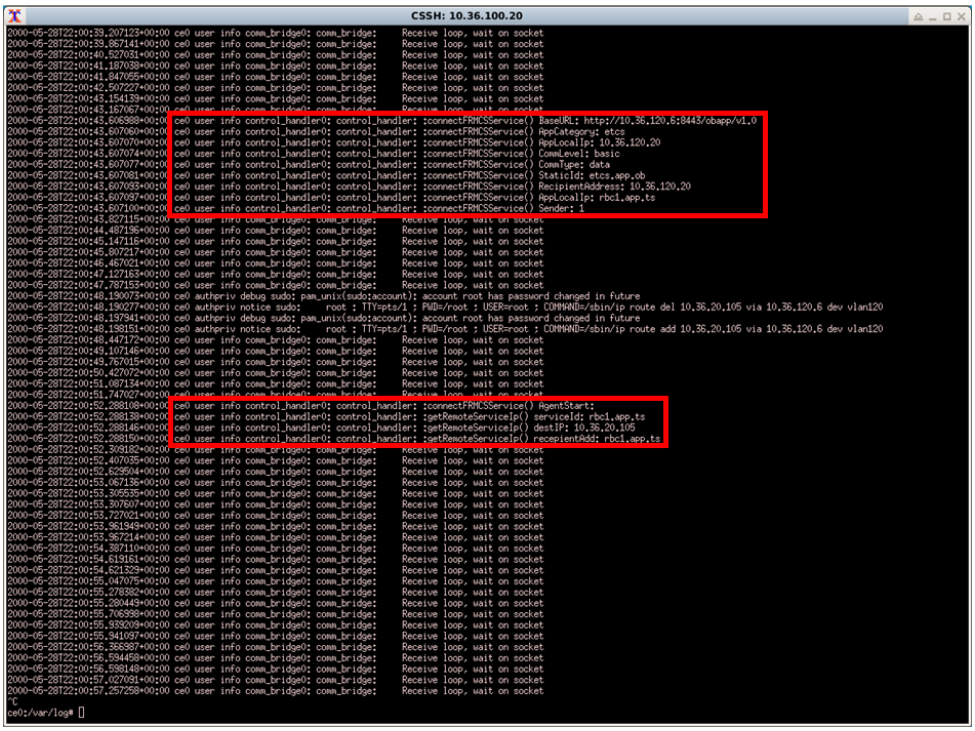
Expected Results:

- The MCx client must successfully authenticate with the IdMS and obtain a JWT token on both onboard and trackside.
- The MCx clients must correctly authenticate with the SIP server on both onboard and trackside.
- SIP registration should return **200 OK**.
- SIP publish should return **200 OK**.
- OB_{APP} protocol local binding must be successfully completed on both onboard and trackside.
- MCX client on onboard SIP INVITE must return **200 OK**.
- OB_{APP} open session must be successfully completed from onboard to trackside.
- Verify that the connection from SAFE APP on onboard is successfully opened to the Trackside App.
- Confirm that data transmission occurs via GRE in a UDP tunnel established between onboard and trackside.
- Confirm application data is going through GRE in UDP.

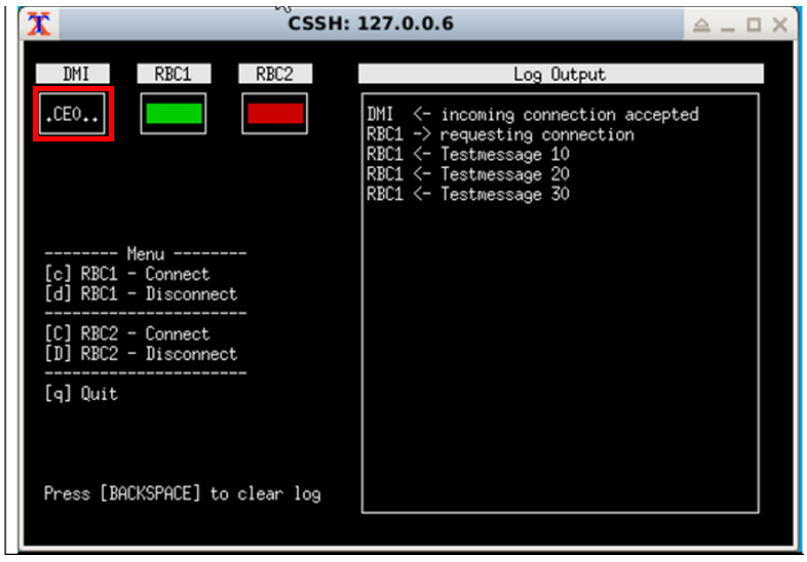
Pass/Fail Criteria:

- **Pass**

Connection to trackside App #1 (RBC1) through the control handler on CE0:



Visualization of the onboard application indicates that the active bridge is running on CE0 and the onboard application is connected to RBC1:



ID: FRMCS_APPS_HA

Summary:

This test verifies that the onboard application (OB APP) running on TAS can successfully communicate with the Trackside application (TS APP) in failover scenario

Purpose:

Ensure that an application on the Modular Platform (including an application on the

safety layer/runtime environment) can communicate over FRMCS to a trackside entity even if one of the Modular Platform CEs failed, demonstrating an FRMCS communication that remains transparent to the application.

Priority: High

Test Environment:

- **Hardware:** TS_GTW interfaced with core network, OB_GTW interfaced with a Wi-Fi Radio Module.
- **Network:** TS_GTW with connectivity to the MCx server via core network interface, FRMCS Wi-Fi with connectivity to the MCx server.
- **Tools:** Network tracing utilities (Wireshark, tcpdump)
- **TAS_PLATFORM:** Onboard FRMCS Agent running on TAS, SAFE APP replicated on multiple CEs.

Pre-requisites:

- TS_GTW must be connected to the core network.
- The interface to the core network must have access to the MCx server.

Steps:

1. From the TS FRMCS AGENT, send a ping request to the TS GTW to verify connectivity.
2. From the TS GTW, send a ping request to the TS MCx Server to verify connectivity.
3. Configure the MCx client on the trackside gateway.
4. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication and OB_{APP} interface.
5. Start the FRMS services on TS GTW.
6. Start the FRMS local binding on TS FRMCS AGENT.
7. Capture traces on both TS FRMCS AGENT and TS GTW.
8. Enable Wi-Fi coverage on onboard.
9. From the onboard gateway (OB_GTW), send a ping request to the MCx server to verify connectivity.
10. From the OB_FRMCS AGENT on TAS, send a ping request to the OB_GTW to verify connectivity.
11. Configure the MCx client on the onboard gateway.
12. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication.

13. From the TAS, send a ping request to the onboard gateway to verify connectivity.
14. Start FRMCS services on onboard.
15. Start the trackside application.
16. Start the SAFE APP on the TAS PLATFORM.
17. Let the app run for some time.
18. Put down the active CE.
19. Take trace on OB APP and Wifi interface.

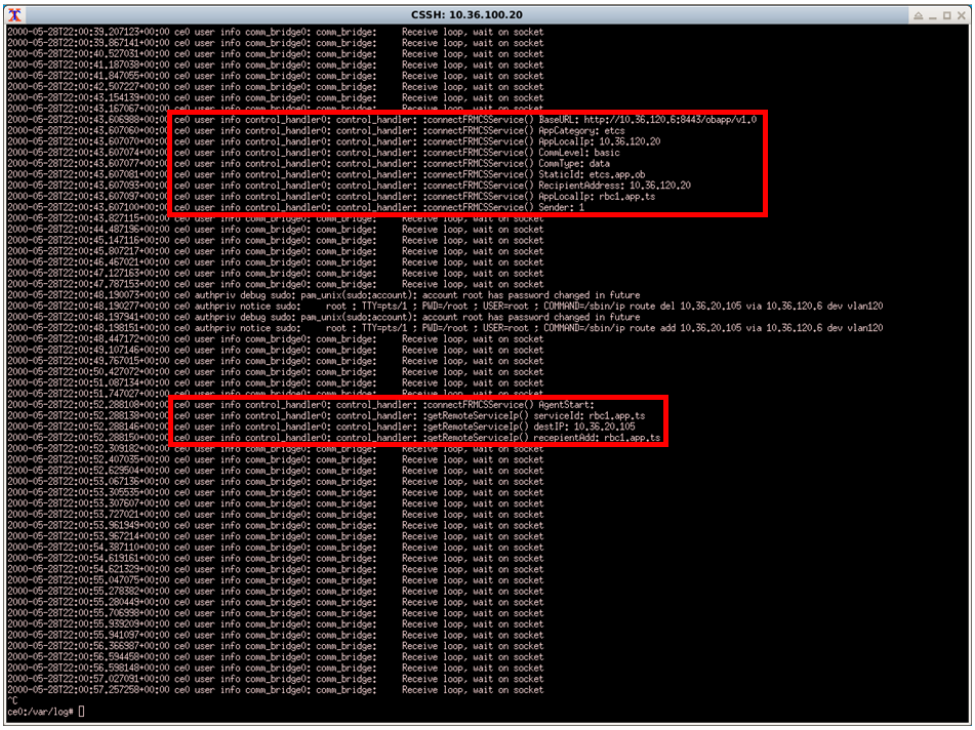
Expected Results:

- For the first connection:
  - The MCx client must successfully authenticate with the IdMS and obtain a JWT token on both onboard and trackside.
  - The MCx clients must correctly authenticate with the SIP server on both onboard and trackside.
  - SIP registration should return **200 OK**.
  - SIP publish should return **200 OK**.
  - OB_{APP} protocol local binding must be successfully completed on both onboard and trackside.
  - MCX client on onboard SIP INVITE must return **200 OK**.
  - OB_{APP} open session must be successfully completed from onboard to trackside.
  - Verify that the connection from SAFE APP on onboard is successfully opened to the Trackside App.
  - Confirm that data transmission occurs via GRE in a UDP tunnel established between onboard and trackside.
  - Confirm application data is going through GRE in UDP.
- After shutting down the active CE, for the second connection the above sequence repeats.

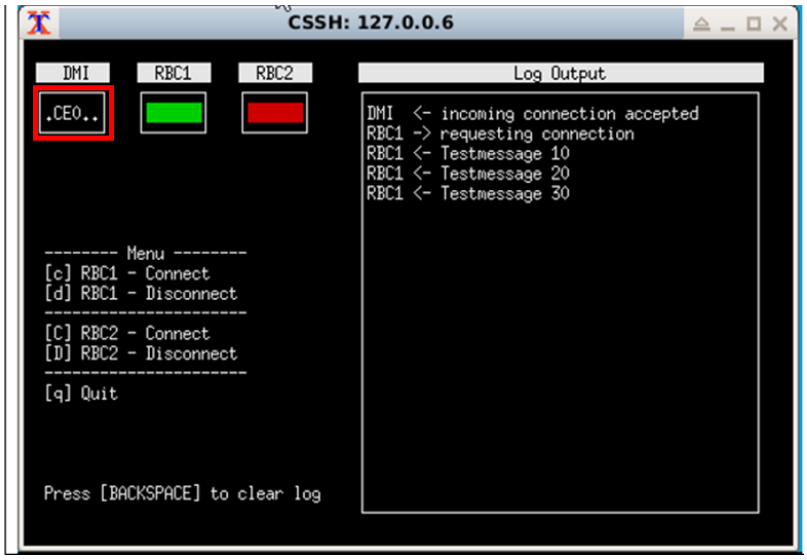
Pass/Fail Criteria:

- **Pass**
- **Finding:** During testing it turned out that the establishment of the second connection can take up to 5 seconds. Reasons for this need further study.

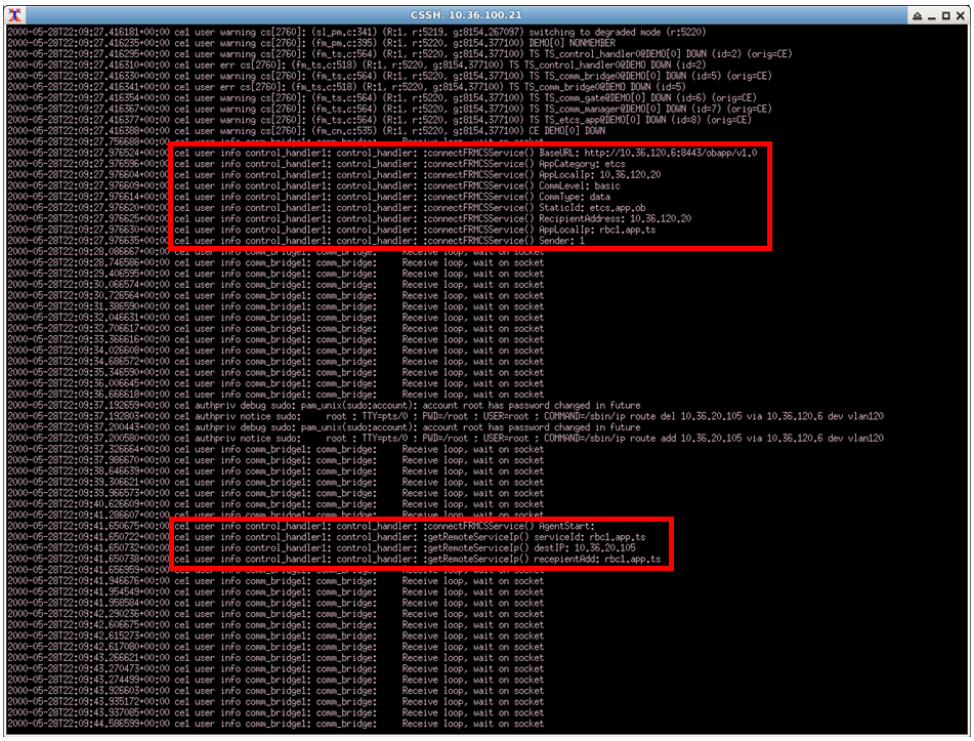
Initial connection to trackside App #1 (RBC1) through the control handler on CE0:



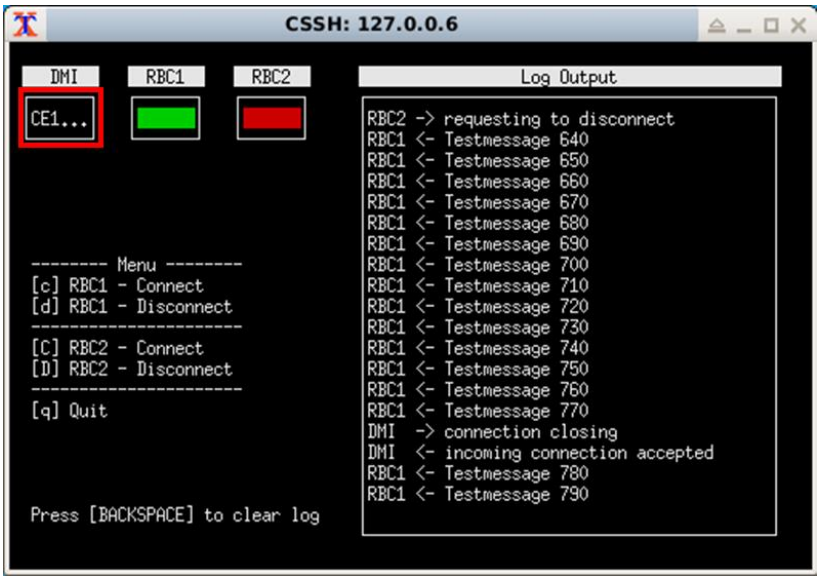
Visualization of the onboard application indicates that the active bridge is running on CE0 and the onboard application is connected to RBC1:



After schutting down CE0, the communitaion manager re-establishes the connection using the communication_bridge respectively the control handler running on CE1.



Visualization of the onboard application indicates that the active bridge is now running on CE1 and the onboard application is connected to RBC1:



ID: FRMCS_LOOSE_APP

Summary:

This test verifies that the onboard application (OB APP) running TAS can successfully communicate via the Trackside application (TS APP).

Purpose:

Ensure that an application on the Modular Platform (including an application on the safety layer/runtime environment) can communicate over FRMCS to a trackside entity, demonstrating an FRMCS communication that remains transparent to the application.

Priority: High

Test Environment:

- **Hardware:** TS_GTW interfaced with core network, OB_GTW interfaced with a Wi-Fi Radio Module.
- **Network:** TS_GTW with connectivity to the MCx server via core network interface, FRMCS Wi-Fi with connectivity to the MCx server.
- **Tools:** Network tracing utilities (Wireshark, tcpdump)
- **TAS_PLATFORM:** Onboard agent running on TAS, SAFE APP.

Pre-requisites:

- TS_GTW must be connected to the core network.
- The interface to the core network must have access to the MCx server.

Steps:

1. From the TS FRMCS AGENT, send a ping request to the TS GTW to verify connectivity.
2. From the TS GTW, send a ping request to the TS MCx Server to verify connectivity.
3. Configure the MCx client on the trackside gateway.
4. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication and OB_{APP} interface.
5. Start the FRMS services on TS GTW.
6. Start the FRMS local binding on TS FRMCS AGENT.
7. Capture traces on both TS FRMCS AGENT and TS GTW.
8. Enable Wi-Fi coverage on onboard.
9. From the onboard gateway (OB_GTW), send a ping request to the MCx server to verify connectivity.
10. From the OB_FRMCS AGENT on TAS, send a ping request to the OB_GTW to verify connectivity.
11. Configure the MCx client on the onboard gateway.
12. Start a network tracing tool (Wireshark/tcpdump) on the interface used for MCx server communication.
13. From the TAS, send a ping request to the onboard gateway to verify connectivity.
14. Start FRMS services on onboard.
15. Start the trackside application.

16. Start the SAFE APP on the TAS PLATFORM.

Expected Results:

- The MCx client must successfully authenticate with the IdMS and obtain a JWT token on both onboard and trackside.
- The MCx clients must correctly authenticate with the SIP server on both onboard and trackside.
- SIP registration should return **200 OK**.
- SIP publish should return **200 OK**.
- OBAPP protocol local binding must be successfully completed on both onboard and trackside.
- MCX client on onboard SIP INVITE must return **200 OK**.

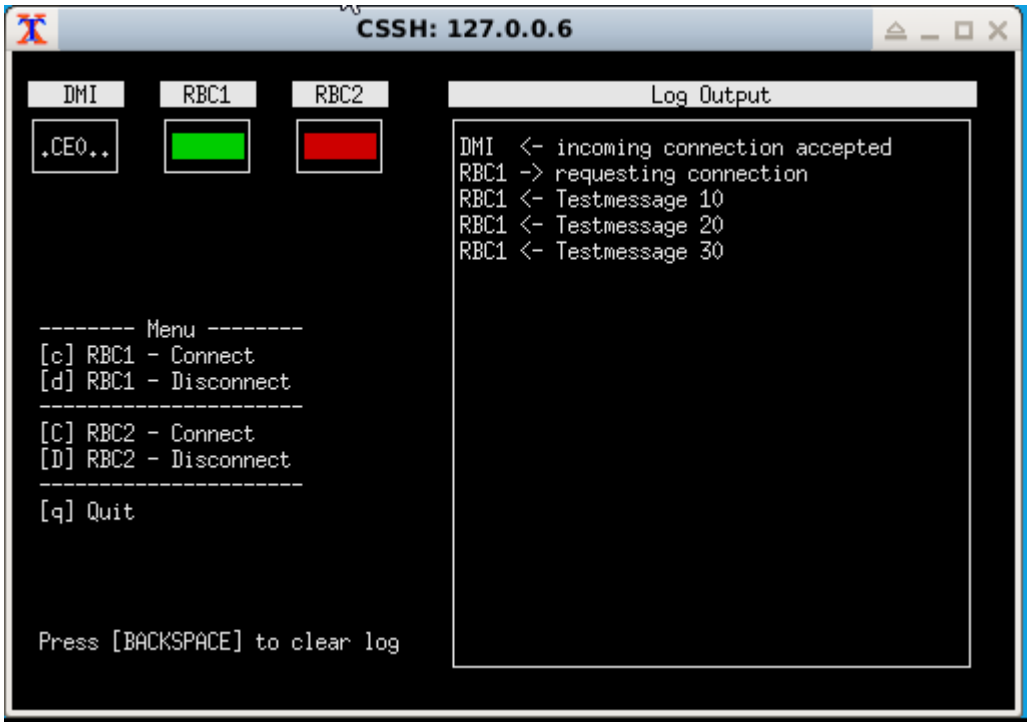
- OBAPP open session must be successfully completed from onboard to trackside.

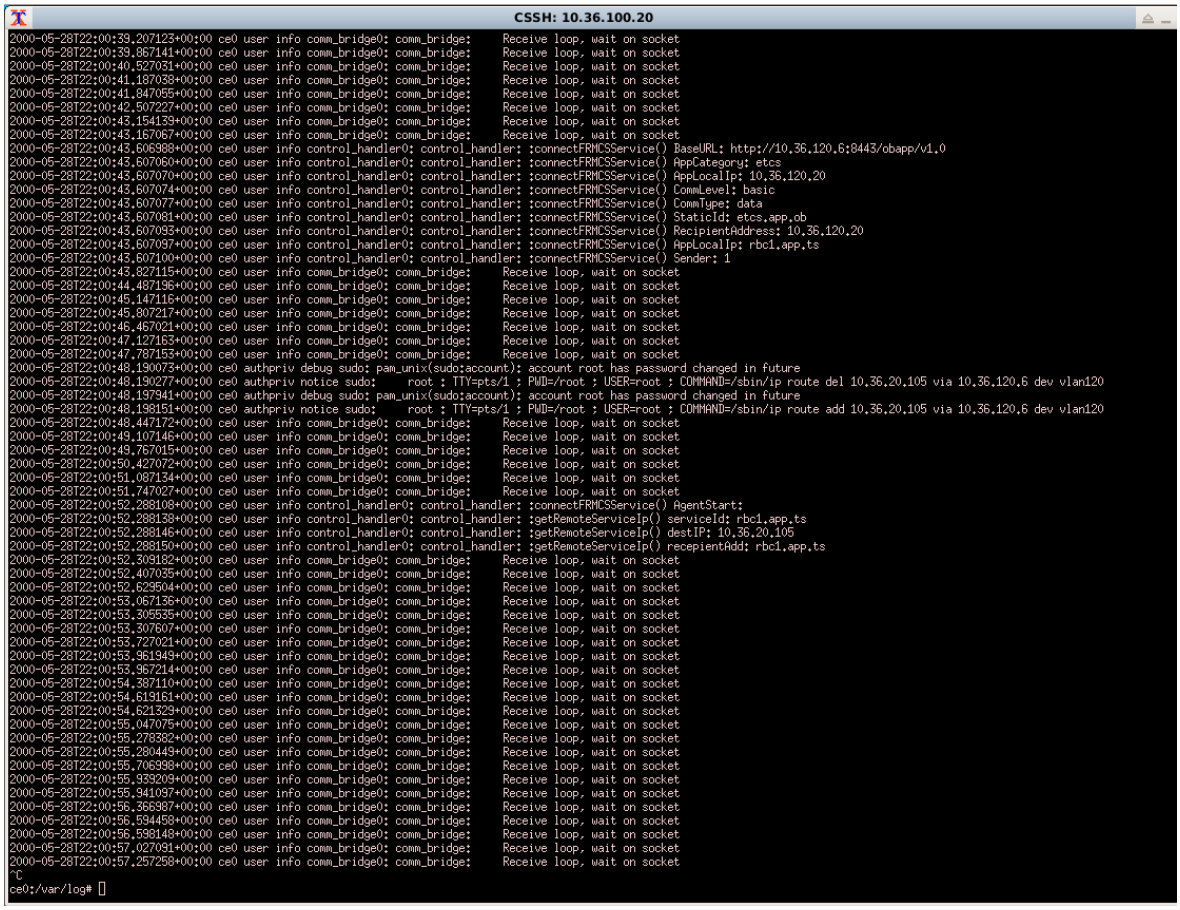
- Verify that the connection from SAFE APP on onboard is successfully opened to the Trackside App.

- Confirm that data transmission occurs via GRE in a UDP tunnel established between onboard and trackside.
- Confirm application data is going through GRE in UDP.

Pass/Fail Criteria:

- **Pass**


CSSH: 127.0.0.6
DMI
RBC1
RBC2
.CE0..
Log Output
DMI <- incoming connection accepted
RBC1 -> requesting connection
RBC1 <- Testmessage 10
RBC1 <- Testmessage 20
RBC1 <- Testmessage 30
-------- Menu --------
[c] RBC1 - Connect
[d] RBC1 - Disconnect
[C] RBC2 - Connect
[D] RBC2 - Disconnect
[q] Quit
Press [BACKSPACE] to clear log


CSSH: 10.36.100.20
2000-05-28T22:00:39.207123+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:39.867141+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:40.527031+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:41.187038+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:41.847055+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:42.507227+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:43.154139+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:43.167067+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:43.606988+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() BaseURL: http://10.36.120.6:8443/obapp/v1.0
2000-05-28T22:00:43.607060+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() AppCategory: etcs
2000-05-28T22:00:43.607070+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() AppLocalIp: 10.36.120.20
2000-05-28T22:00:43.607074+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() CommLevel: basic
2000-05-28T22:00:43.607077+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() CommType: data
2000-05-28T22:00:43.607081+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() StaticId: etcs.app.ob
2000-05-28T22:00:43.607093+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() RecipientAddress: 10.36.120.20
2000-05-28T22:00:43.607097+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() AppLocalIp: rbc1.app.ts
2000-05-28T22:00:43.607100+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() Sender: 1
2000-05-28T22:00:43.827115+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:44.487196+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:45.147116+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:45.807217+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:46.467021+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:47.127163+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:47.787153+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:48.190073+00:00 ce0 authpriv debug sudo: pam_unix(sudo:account): account root has password changed in future
2000-05-28T22:00:48.190277+00:00 ce0 authpriv notice sudo: root : TTY=pts/1 ; PWD=/root ; USER=root ; COMMAND=/sbin/ip route del 10.36.20.105 via 10.36.120.6 dev vlan120
2000-05-28T22:00:48.197941+00:00 ce0 authpriv debug sudo: pam_unix(sudo:account): account root has password changed in future
2000-05-28T22:00:48.198151+00:00 ce0 authpriv notice sudo: root : TTY=pts/1 ; PWD=/root ; USER=root ; COMMAND=/sbin/ip route add 10.36.20.105 via 10.36.120.6 dev vlan120
2000-05-28T22:00:48.447172+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:49.107146+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:49.767015+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:50.427072+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:51.087134+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:51.747027+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:52.288108+00:00 ce0 user info control_handler0: control_handler: :connectFRMCSService() AgentStart:
2000-05-28T22:00:52.288138+00:00 ce0 user info control_handler0: control_handler: :getRemoteServiceIp() serviceId: rbc1.app.ts
2000-05-28T22:00:52.288146+00:00 ce0 user info control_handler0: control_handler: :getRemoteServiceIp() destIP: 10.36.20.105
2000-05-28T22:00:52.288150+00:00 ce0 user info control_handler0: control_handler: :getRemoteServiceIp() recepientAdd: rbc1.app.ts
2000-05-28T22:00:52.309182+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:52.407035+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:52.629504+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.067136+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.305535+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.307607+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.727021+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.961949+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:53.967214+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:54.387110+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:54.619161+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:54.621329+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.047075+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.278382+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.280449+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.706998+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.939209+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:55.941097+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:56.366987+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:56.594458+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:56.598148+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:57.027091+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
2000-05-28T22:00:57.257258+00:00 ce0 user info comm_bridge0: comm_bridge: Receive loop, wait on socket
^C
ce0:/var/log#

4.1.1.2 Summary

With the test cases provided in the previous chapters we have verified the following functions:

- Successful Authentication of onboard MCx client to FRMCS services
- Successful Authentication of trackside MCx client to FRMCS services
- Successful Authentication to FRMCS services of onboard FRMCS Agent via OB-GW using OB_{APP}
- Successful Authentication of Trackside FRMCS Agent to FRMCS services
- Successful Communication of onboard entities running on Modular Platform (TAS) connected via FRMCS Agent to the FRMCS onboard gateway with the trackside application connected via trackside FRMCS Agent to the trackside FRMCS gateway

As such a full communication chain from onboard application to trackside application has been successfully verified.

In addition, a failover scenario inside the Modular Platform has been verified. So even if one CE of the Modular Platform fails, FRMCS communication remains active and transparent to the application. The recognised long re-establishment time of up to 5 seconds needs further investigation. The root cause might be the laboratory setup (including potentially the usage of WiFi connection instead of FRMCS 5G mobile communication), the implementation or even the FRMCS standard which we cannot judge at this time.

4.1.2 User Story 2.4.4: Failover to Redundant FRMCS Platform Functions

4.1.2.1 Description of the User Story

4.1.2.1.1 References

EROPD-28, EROPD-32

4.1.2.1.2 Description

As R2DATO WP36, we want an application on the Modular Platform (and an application on the safety layer / runtime environment) to communicate over redundant FRMCS gateways to a trackside entity, so that we can demonstrate the increase in availability in failover scenarios.

4.1.2.1.3 Related Requirements

TOBA-FRS-8.1

4.1.2.2 Theoretical analysis of the redundancy options

Building on the redundancy options identified and described in from Section 3.1.3, this section provides a comprehensive analysis of these options for production deployments.

The analysis of the redundancy options scenarios was done along qualitative criteria which are detailed below:

- Redundancy level: What level of redundancy can be achieved?
- Implementation feasibility: Is the implementation technically possible? What are the technical pros and cons?

- Implementation effort: What is the qualitative relative estimation of the implementation effort?
- Security: What are the security implications?
- Safety: What are the safety-related implications?

It should be noted that we concentrate in this analysis on the redundancy of the FRMCS Platform Functions. The redundancy of the components in the Modular Platform are complementary redundancy functions which work in addition to the redundancy functions of the FRMCS onboard components – these have been not analysed.

4.1.2.2.1 Redundancy Option 1

Redundancy level:

Redundancy of the onboard gateway is transparent to the application. This is achieved through a virtual single IP address managed by the VRRP shared between the two onboard gateways.

The actual level of redundancy depends on the internal sync and replication between the two redundant onboard gateways. If all relevant session data is replicated in theory a switchover can be done safely. In real world implementation there might be still some cases where a complete switchover cannot be guaranteed (e.g. because syncing is not instantaneous, race conditions might occur in various states of the session). These potential glitches need to be considered in application layer design.

Implementation feasibility:

The implementation of this redundancy option requires the following high-level functions:

- Synchronization of internal states between active and standby OB-GW
- Monitoring of OB_{APP} link between the FRMCS Agent and the OB-GW
- IP Address switchover via e.g. VRRP in case of loss of communication of link between the FRMCS Agent to the active OB-GW
- Re-establishment of communication between the Application / FRMCS Agent and the OB-GW

Preliminary assessment has shown that the implementation of these high-level functions is feasible and technically possible regarding application site interfaces.

Still to be assessed are the implications in terms of FRMCS network side. Re-establishment of an active MC-Client connection between OB-GW and FRMCS network with the same connection parameters on the redundant OB-GW, respectively, re-entering an ongoing connection, needs some more assessment which could not be done in the project timeframe.

Implementation effort:

The implementation effort to implement the functions stated in the section “Implementation feasibility” seems according to the preliminary assessment feasible in reasonable timeframe, as these are mainly functions which are used already in another context for similar functionality.

Security:

Regarding security, mainly the link security between application and OB-GW is affected.

TLS considerations: TLS is used on the OB_{APP} interface for securing the control plane. In the event of a switchover, the TCP connection will potentially have to be renewed on the new box. And as such also the TLS connection must be renewed. There are however some means in TLS which should make a re-establishment of the TLS connection for the new onboard gateway possible (TLS Session ID Resumption, TLS Session Ticket resumption, removing the need for session invalidation if TCP connection is closed in TLS versions from TLS1.1 onwards – see RFC 2246). A further detailed investigation into this topic could however be beneficial.

Safety:

As the FRMCS OB-GW together with the entire FRMCS communication network is non-safety relevant, also no impact on the safety consideration of the application is seen. The safety related application needs to take care of any loss of communication between the communication elements.

4.1.2.2.2 Redundancy Option 2

Redundancy level:

Redundancy of the onboard gateway is in in this scenario not transparent to the application.

The application needs to manage the switchover.

Thus, the actual level of redundancy depends on the implementation of the application switchover mechanism and the ability of the OB-GW to start up the standby OB-GW when a switchover is requested.

In addition, some synchronization logic between the application-level switchover and the OB-GW switchover needs to be considered to have both application and OB-GW using the same connection. This could be achieved by promoting either the application or the OB-GW as kind of “master”, responsible for detecting the switchover trigger and starting the switchover procedure. The initial perspective is that the application could serve as the master.

Internal sync and replication between the two redundant onboard gateways are also required in this scenario. The same considerations apply as in Scenario 1.

Implementation feasibility:

The implementation of this redundancy option requires the following high-level functions:

- Synchronization of internal states between Active and Standby OB-GW
- Monitoring of OB_{APP} link between Application, FRMCS Agent and OB_GW – preferable only by defined the “master” of the redundancy logic.
- TS-app switchover and re-establishment of communication between the Application and the OB-GW, preferably initiated by the “master” of the redundancy logic.

Preliminary assessment has shown that the implementation of these high-level functions is feasible and technically possible regarding application site interfaces.

Specifically, however in addition the Application is affected and needs to be capable of an implementation of two redundant links towards the OB-GW. Based on the experience of the project implementation on the Modular Platform, the implementation of the necessary functions at the application seems feasible as well.

Regarding the link towards the FRMCS network, same assessment as for Option 1 applies:

- Still to be assessed are the implications in terms of FRMCS network side. Re-establishment of an active MC-Client connection between OB-GW and FRMCS network with the same connection parameters on the redundant OB-GW, respectively, re-entering an ongoing connection, needs some more assessment which could not be done in the project timeframe.

Implementation effort:

The implementation effort to implement the functions stated in the section "Implementation feasibility" seems according to the preliminary assessment feasible in reasonable timeframe.

However, compared to implementation option 1 the implementation effort might be shifted. Some of the implementation efforts will be moved towards the application as the application has a significant role in the redundancy concept.

Regarding the link switchover the effort is potentially expected to be less compared to option 1, as both links are working independently.

Security:

Regarding security, mainly the link security between application and OB-GW is affected.

As the connections between application and OB-GW active and OB-GW standby are separate connections, also the security of each link is separate. As such there is no effort necessary to re-establish dropped connections, but just a new connection to be opened. Thus, no TLS re-establishment considerations do apply.

Safety:

As the FRMCS OB-GW together with the entire FRMCS communication network is non-safety relevant, also no impact on the safety consideration of the application is seen. The safety related application needs to take care of any loss of communication between the communication elements.

4.1.2.3 Summary

With the Testcase provide in previous chapter we verified the following functions:

- Successful verification that a loose coupled application can successfully communicate from onboard application part connected via OB_{APP} to the FRMCS System all the way down to the trackside application, which is itself connected via TS_{APP} to the FRMCS System.

In the theoretical analysis of the previous chapters, we analysed two potential redundancy options for the FRMCS onboard gateway.

Note that redundancy analysis of components on the Modular Platform is not part of this user story.

The main results are:

- Both analysed redundancy scenarios are in principle feasible for a high availability implementation of the onboard FRMCS gateway. Depending on the option, more implementation effort is seen either on the Modular Platform hosted SW components or on the FRMCS onboard gateway components.
- There is no explicit recommendation favouring either redundancy option; therefore, both options are considered equivalent in terms of performance.
- Some implications in terms of interfacing towards the FRMCS system side as well as further details on security topics regarding switchover could not be assessed in detail and thus cannot be concluded.

4.1.3 User Story 2.4.5: Host FRMCS Onboard Services on the Modular Platform

4.1.3.1 Description of the User Story

4.1.3.1.1 References

EROPD-24, EROPD-29, EROPD-30, EROPD-46,

4.1.3.1.2 Description

As R2DATO WP36, we want to split onboard FRMCS TOBA functions, with parts of the functions hosted on the same hardware pool as the Modular Platform and decuple the modem hardware (e.g., with OB_{RAD} if available) via the CCN (also analysing the hardware effort of the different hosting scenarios), so that we can demonstrate the modularity of the TOBA solution.

4.1.3.1.3 Related Requirements

FRMCS-SRS-7.1, TOBA-FRS-5.2.1.5, TOBA-FRS-7.9

4.1.3.1.4 Test Cases

Not applicable - only theoretical analysis was done in the project

4.1.3.2 Analysis of the scenarios

Based on the identified and described scenarios from Section 3.1.3, this section aims to give a qualified analysis of the scenarios for production deployments.

The analysis of the deployment scenarios was done along qualitative criteria which are detailed below:

- Implementation feasibility: Is the implementation technically possible? What are the technical pros and cons?
- Implementation effort: What is the qualitative relative estimation of the implementation effort?
- Security: What are the security implications?
- Safety: What are the safety-related implications?
- Modularity: What is the impact on the modularity of the system?

- Multi-Vendor deployment: What are the implications if different parts of the system are provided by different vendors?

Note that we focused on the relative difference for each of the scenarios. Items which are similar for all three deployment scenarios have not been described as they do not change the distinction between the deployment scenarios.

4.1.3.2.1 Deployment Scenario 1

Implementation feasibility:

Standard implementation utilizing interfaces OB_{APP} and OB_{ANT} as defined in FRMCS Specification. Modular Platform and FRMCS Onboard Gateway remains as separate implementation blocks – thus fully feasibly as per design of FRMCS specification.

Implementation effort:

As the FRMCS Onboard Gateway is deployed separately, we consider this as the standard implementation effort amongst the three scenarios

Security:

As the FRMCS Onboard Gateway stays completely separated, the defined Security measures in FRMCS FFFIS Specification for OB_{APP} [7] Chapter 6.3 apply and form a well secured interface between the Modular Platform and the FRMCS Onboard Gateway.

The security measures and guidelines for the Modular Platform apply naturally and are not affected by this deployment scenario.

Safety:

The Modular Platform is hosting the safety related applications. The link from the safety related software components to the non-safety related components on the Modular Platform is done inside this platform already (between Communication Gate and Communication Bridge). The FRMCS OB-Gateway is not safety relevant. As such this deployment model is not influencing any safety related implementation or concept at all.

Modularity:

The FRMCS Onboard Gateway has defined interfaces on both sides with OB_{APP} and OB_{ANT}. Therefore, modularity within these defined interfaces is a given. The Modular Platform itself provides its own modularity.

Multi-Vendor deployment:

Due to the usage of the OB_{APP} interface to interconnect Modular Platform SW modules with the OB-Gateway, a well-defined interface is used. Thus, interoperability testing on this interface should be sufficient to support well a multi-vendor deployment between Modular Platform and FRMCS Onboard Gateway.

4.1.3.2.2 Deployment Scenario 2

Implementation feasibility:

The intended functions of the FRMCS Onboard Gateway for deployment on Modular Platform are software related functions only (Communication Gateway Function).

Thus, the implementation of the functions on Modular Platform (part of un-safe SW layers) is fully feasible.

However, the formally internal interface between Communication Gateway Function and Radio Function(s) is getting external – which is currently not defined in standardization.

In case the FRMCS Onboard Gateway is already implemented in a modular way, where the Radio Functions are deployed separated/remotely, additional HW could be saved for deployment of the "Communication Gateway Function"

Implementation effort:

Due to moving FRMCS Onboard Function "Communication Gateway Function" to the Modular Platform development alignment to the distinct SW architecture is required which increases implementation effort due to SW porting and testing.

Security:

In this scenario the OB_{APP} security measures from [7] will move inside the Modular Platform, but some of the requirements probably need to be reconsidered as they cover physical ethernet connections (inherited from Subset-147).

In addition, it is expected that the "Communication Gateway Function" will has to respect potential additional security and or hardening guidelines from the Modular Platform and its development environment. The current evaluation assumes that such guidelines should be nevertheless in the range of standard guidelines for security and will not complicate the implementation at the Modular Platform excessively.

Safety:

When adding the "Communication Gateway Function" of the FRMCS Onboard Gateway to the Modular Platform, only non-safety related components are added. Thus, the safety related statement for Deployment Scenario 1 does not change and the transition from safety related SW modules to non-safety related SW module remains unchanged on the Modular Platform. As such, this deployment model is transparent to safety related implementation.

Modularity:

Moving the FRMCS "Communication Gateway Function" to the Modular Platform can increase the modularity of the system as an additional interface inside the FRMCS Onboard Gateway will be externalized. Note: This interface is supposed to be defined in Future Versions of the FRMCS Specification.

Due to this a better modularity e.g. for deployment of the Radio Function(s) can be expected.

Multi-Vendor deployment:

As the goal of the Modular Platform is the deployment of different SW modules also from different vendors, deployment of the "Communication Gateway Function" from different vendors as additional SW module is definitely feasible. However, SW adaptations to comply to specific SW architectures interfaces, or development environment will most probably increase complexity and leads likely to adaptations on existing SW or development workflows.

4.1.3.2.3 Deployment Scenario 3

Implementation feasibility:

Implementing the radio function requires HW interfaces at the Modular Platform to attach some kind of standard radio modules and specifically also FRMCS radio module HW. This defines essentially the possible compute platform hardware.

Potential redundancy functions with multiple Radio Functions need to be mapped on the Modular Platform architecture (software and hardware)

Implementation effort:

Due to moving, in addition to SW functions, also hardware functions to the Modular Platform, specific HW needs to be selected for the Modular Platform to allow hosting of Radio Functions, specifically the Radio Modules. Thus, for existing Modular Platform Hardware, new HW selection process is required. For new product introductions this might not be that relevant, however still additional HW interfaces are required, which usually increases the cost level.

Testing effort, especially and compatibility with Radio Modules needs to be factored for the Modular Platform hardware.

Security:

The security considerations for Scenario 2 fully apply for Scenario 3 as well. In addition, as new Hardware elements do apply for deployment on Modular Platform, physical security measures.

Safety:

When adding the complete FRMCS Onboard Gateway to the Modular Platform only non-safety related components are added. Thus, the same statement as for Deployment Scenario 2 applies here as well and the scenario is transparent to safety related aspects.

Modularity:

Putting complete FRMCS Onboard Gateway into the Modular Platform removes the strict need to invent the Interface between the two main FRMCS Onboard Gateway functions “Communication Gateway Function” and “Radio Function(s)”. Thus, deployments in the same level of modularity as scenario 2 are unlikely to happen as the strict need for this modularity does not apply any more. As such internal interfaces might be used between the two aforementioned modules.

Multi-Vendor deployment:

Regarding the aspect of moving the SW Functions of the FRMCS Onboard Gateway towards the Modular Platform, the same interface is still applying to interconnect to the Application also at the Modular Platform, namely OB_{APP}. As such the IoT Testing of the OB_{APP} interface remains vital as in the other scenarios, but thus also supports fully a multi-vendor deployment. Of course, the same arguments for multi-vendor deployment as stated for Deployment Scenario 2 apply as well. Regarding the aspect of moving HW Functions (Radio Modules) from a standalone deployment onto the Modular Platform HW, the HW interface for the Radio Modules must be considered as multi-vendor interface in addition to all the SW interfaces. As such an additional constraint occurs which increases the number of interfaces to be considered for Design, Implementation and Interoperability Testing.

4.1.3.2.4 Summary

The following table provides a concise summary of the analysis done in previous chapters. In addition to the textual summary, a qualitative rating is added, with the following legend:

+ overall positive impact

- overall negative impact

-- overall higher negative impact

(-) conditional negative impact

~ neutral or mean level (no overall positive or negative impact)

Scenario → Criteria↓	Scenario 1 / separate FRMCS Onboard Gateway	Scenario 2 / FRMCS Onboard Gateway partly on Modular Platform	Scenario 3 / FRMCS Onboard Gateway fully on Modular Platform
Implementation feasibility	+ Good	+ Good	- Challenging (depending on HW Platform
Implementation effort	~ Standard	- Slightly higher compared to standard	-- Significantly higher compared to standard
Security	~ Based on FRMCS Security requirements, including OB_{APP} security Unchanged Platform security.	- Additional interface security to be defined. FRMCS components need respect Platform security.	-- Additional interface security to be defined. Physical & HW security to be considered for FRMCS components FRMCS components need respect Platform security.
Safety	Neutral	Neutral	Neutral
Modularity	~ As per FRMCS definition of FRMCS Onboard Gateway borders OB_{APP} and OB_{ANT}	+ Increased Modularity due to need of potential standardized OB_{RAD} interface between Modular Platform and Radio function(s).	~ No strict need for increased modularity as all on Compute Platform
Multi-Vendor deployment	~ Well Feasible	~ (-) Feasible with additional standard/open SW Interface	~ (-) Feasible with additional standard/open HW Interface

Table 4: Summary of analysis of FRMCS OB-GW deployment scenarios

In summary we can thus mark the Scenario 1 one as the straightforward standard scenario. For some additional effort tighter integration with the Modular Platform moving to Scenario 2 can be provided, depending on the standardization and future wide-spread adoption of

the interface towards the Radio Function. The Scenario 3 is due to its HW dependency the least feasible variant and the most expensive for a deployment with questionable gain.

4.2 Functional Cluster Diagnostics

As elaborated in chapter 3.2.2.1 an automated testing approach was taken to reliably verify the tested User Stories. During our work, the use of automated testing has proven to provide the following advantages:

- **Exhaustive Documentation of Test Scenarios**

  Scripts that are automatically executed by computers must naturally contain all information that is necessary to complete the tasks. There can be no implicit knowledge or imprecision. This facilitates that tests can be repeated reliably by anyone.

- **Reproducible Results**

  As mentioned above the precision of the scripted test execution makes test results well reproduceable. This is not only a relevant quality of professional research but also allows for easy regression testing, whenever a change is introduced to the subject under test or the test environment. Combined with small and incremental changes (as suggested by the principle of Continuous Integration), a breaking change can be pinpointed to only a minimal changeset. This allows for easy and efficient failure discovery.

- **Short Feedback Cycles**

  To make the principle of Continuous Integration practical, feedback cycles must be short and complete in predictable times. This can be achieved by automatic testing, since the completion time does not rely on manual interaction and the risk of human error during the execution is eliminated. The context information from the configuration management additionally allows for smart caching of artifacts as already described in chapter 3.2.2.1 and Figure 35: Clean build vs. utilisation of caching.

To leverage those advantages, Azure DevOps Pipelines have been used to define the steps to be executed for build, deploy and test (see also Figure 34: Jobs of the Azure DevOps Pipeline). Since the two User Stories that have been implemented in this task both utilise the same test environment and tested components (see Figure 32: Deployed MCP-DIA Components in SBB-laboratory), their individual verification only affects the test step of the pipeline.

An outline of the steps in the common Build, Run and Clean-up stages is given in the listing below:

```
trigger:
  - main

stages:
- stage: build
  pool:
    vmImage: ubuntu-latest
  displayName: Build in the cloud
  jobs:
  - job:
    pool:
      vmImage: ubuntu-latest
    displayName: Build basic_tas_app
    steps:
      - checkout: self
        displayName: 'Checkout avl_tas_integration'
      - task: Cache@2
        inputs:
          key: '"basic_tas_app" |[…]'
          path: 'basic_tas_app'
          cacheHitVar: 'BASIC_TAS_APP_RESTORED'
        displayName: 'Cache basic_tas_app build'
      - task: Cache@2
        inputs:
          key: '"protobuf" | […]'
          path: 'avl_tas_integration/protobuf'
          cacheHitVar: 'PROTOBUF_RESTORED'
        displayName: 'Cache protobuf build'
      # setup build environment
      […]
      # build protobuf
      - script: |
          […]
        displayName: 'Build protobuf in build container'
        condition: ne(variables.PROTOBUF_RESTORED, 'true')
      - publish: 'avl_tas_integration/protobuf'
        displayName: 'Publish protobuf artifact'
        artifact: protobuf
      # build basic_tas_app
      - script: |
          […]
        displayName: 'Install and run cppcheck'
```

```
43          continueOnError: 'false'
44          condition: ne(variables.BASIC_TAS_APP_RESTORED, 'true')
45        - script: |
46            […]
47            make \
48            TARGET_SYS=aeos5 \
49            TARGET_ARCH=x86_64 \
50          displayName: 'Build basic_tas_app in build container'
51          condition: ne(variables.BASIC_TAS_APP_RESTORED, 'true')
52        - publish: 'basic_tas_app'
53          displayName: 'Publish basic_tas_app artifacts'
54          artifact: basic_tas_app
55  - stage: run
56    dependsOn: build
57    pool: ERJU-R2DATO-WP36
58    displayName: Run in physical lab
59    jobs:
60    - job: deploy
61      displayName: Deploy sovd_server and basic_tas_app
62      steps:
63        - checkout: self
64          displayName: 'Checkout avl_tas_integration'
65        - checkout: git://avl_mcp_dia_bin
66          lfs: true
67          displayName: Checkout avl_mcp_dia_bin (sovd_server)
68        - download: current
69          artifact: protobuf
70          displayName: 'Download protobuf artifact'
71        - download: current
72          artifact: basic_tas_app
73          displayName: 'Download basic_tas_app artifacts'
74        - script: |
75            […]
76          displayName: 'Copy binaries to NFS share'
77        - script: |
78            […]
79          displayName: 'Backup and set start script'
80        - task: SSH@0
81          inputs:
82            sshEndpoint: 'ssh-con-ce0'
83            runOptions: 'commands'
84            commands: |
```

```
85              reboot
86          displayName: 'Reboot ce0'
87        - script: |
88            sleep 90
89          displayName: 'Wait for restart'
90    - job: test
91      dependsOn: deploy
92      displayName: Test sovd_server
93      steps:
94        <individual test steps>
95    - job: collect
96      dependsOn: test
97      displayName: Collect evidence
98      steps:
99        […]
100       - publish: '/mnt/tas-physical-deploy/erju/mcp-dia/logs'
101         displayName: 'Publish logs as artifact'
102         artifact: logs
103         condition: always()
104 - stage: clean
105   dependsOn: run
106   pool: ERJU-R2DATO-WP36
107   displayName: Clean up in physical lab
108   jobs:
109   - job: clean
110     displayName: Clean up
111     steps:
112       […]
```

The three dots […] indicate that some parts have been omitted in the listing. Those are either implementation details that are confidential and cannot be disclosed or boiler plate that is not necessary to reproduce the general approach.

The placeholder <individual test steps> is where the actual test commands are placed.

Regarding the diagnostics communication paradigms explained in chapter 2.2.2.2 the tested user stories 2.5.6: “Provide Diagnostics Events to Basic Integrity App on RTE” and 2.5.2: “Collect Diagnostics Data from Basic Integrity App on RTE” are actually the two directions of the same end-to-end test scenario. Thus, both user stories can be verified by triggering the same communication loop from the SOVD-Client over the SOVD-Server to the diagnosed application and back.

In our example implementation as described in chapter 3.2.2.2 this is the request for the system information (diagnostics event) answered by the appropriate JSON string (diagnostics data). In the Azure DevOps Pipeline script this can be done by a simple curl command on the respective SOVD end point:

```
      - task: SSH@0
        inputs:
          sshEndpoint: 'ssh-con-mdcm'
          runOptions: 'inline'
          inline: |
            […]
            curl -s --connect-timeout 10
              http://10.36.130.20:7690/apps/SystemHealthStatusApp/data/sysinfo
              > ~/erju/mcp-dia/logs/test_sysinfo_response
          readyTimeout: '20000'
        displayName: 'Curl sysinfo of SystemHealthStatusApp from mdcm'
```

The successful execution of the pipeline containing this step has been recorded multiple times, e.g., with run #20250328.7.

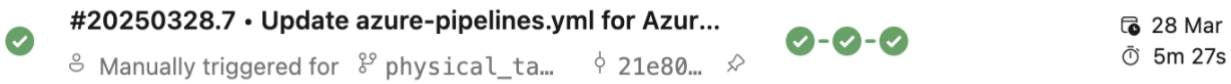


Figure 44: Successful Pipeline Execution for MCP-DIA Test

Before any diagnostics events were sent to the application, it registered itself at the SOVD-Server on startup. This can be observed in the syslog:

```
ce0 user info PING:basic_tas_app: Parsing configuration file path
ce0 user info PING:basic_tas_app: Using configuration file: basic_tas_app.conf
ce0 user info PING:basic_tas_app: Connecting to SOVD-Server: 127.0.0.1:28383
ce0 user info PING:basic_tas_app: Creating Socket
ce0 user info PING:basic_tas_app: Finish TAS platform startup by calling
  cs_sched_yield
ce0 user info PING:basic_tas_app: Provisioned SOVD-Server with app from definition
  file
```

(Time stamps have been omitted for better readability.)

The captured diagnostics data that has been received by the curl command can be found in the file test_sysinfo_response of the logs artifact:

```
{
    "id":"sysinfo",
    "data": {
        "KernelVersion":"Linux x86_64 5.4.251-r17672-xeon-aeos5",
        "OSVersion":"#1 SMP PREEMPT […]",
        "HostName":"ce0"
    }
}
```

Observing this data in the pipeline artifact concludes a successful execution of the test scenario.

4.2.1 User Story 2.5.2: Collect Diagnostics Data from Basic Integrity App on RTE

4.2.1.1 Description of the User Story

4.2.1.1.1 References

EROPD-50

4.2.1.1.2 Description

As R2DATO WP36, we want to collect diagnostics data from a basic integrity application on the safety layer / runtime environment through a standardized interface, so that we can demonstrate a harmonised diagnostics service on the Modular Platform.

4.2.1.1.3 Related Requirements

MSCP-59, MSCP-73, MSCP-84, MSCP-86, MSCP-87, MSCP-111, MSCP-124, MDCM-SRSOCORA-8216

4.2.1.1.4 Test Cases

The general test scenario is described in the overarching chapter 4.2. As outlined in theory in chapter 2.2.2.2 the Model-3 TaskSet SystemHealthStatusApp responds to the triggered request by sending diagnostics data in a standardised and well defined format to the SOVD-Server which collects it and provides it to the original SOVD-Client as payload of the HTTP response.

Even though this already satisfies this User Story, it will also be revisited in the next implementation task when we implement the periodic provisioning of diagnostics data (especially measurements) as already described in theory in chapter 2.2.2.4.3 as well as logging outlined in chapter 2.2.2.6.

4.2.2 User Story 2.5.6: Provide Diagnostics Events to Basic Integrity App on RTE

4.2.2.1 Description of the User Story

4.2.2.1.1 References

EROPD-73

4.2.2.1.2 Description

As R2DATO WP36, we want to provide diagnostics events to a basic integrity application on the safety layer / runtime environment through a standardized interface, so that we can demonstrate a harmonised diagnostics service on the Modular Platform.

4.2.2.1.3 Related Requirements

MSCP-111, MDCM-SRS-OCORA-7425

4.2.2.1.4 Test Cases

The general test scenario is described in the overarching chapter 4.2. As described in chapter 2.2.2.2 the HTTP request towards the SOVD-Server triggers the delivery of a diagnostics event to the application. In this case the application is the Model-3 TaskSet SystemHealthStatusApp with the process name basic_tas_app that has introduced itself to the SOVD-Server on startup according to the prepared configuration (see also chapter 3.2.2.3.1).

The successful provision of this diagnostics event to the application can be proven by the reception of the respective diagnostics data as described in the general end-to-end test scenario.

5 CONCLUSION

This chapter summarizes the achievements, learnings and key takeaways of the project's second implementation and testing phase. Key accomplishments include:

- Successful collaboration among project partners to achieve results integrated into a connected and distributed physical lab environment
- Comprehensive architecture updates reflecting design decisions for realized functions and their integration into the laboratories hosted by SBB and Kontron
- Those designs reach a much more sophisticated level of detail and concrete deployment proposals that have been realised and driven by a sustainable infrastructure for the realisation
- Development and setup of a mature infrastructure, resulting in a ready-to-use physical architecture for deployments in the laboratories
- Implementation and test of User Stories focused on the two main Functional Clusters FRMCS and Diagnostics

Achievements in the Functional Cluster FRMCS:

- Integration of FRMCS into the Modular Platform. This includes a newly developed software component, the FRMCS Agent, that is handling the control plane towards the FRMCS specified OB_{APP} interface, enabling a standardised and application transparent end-to-end FRMCS signalling from onboard to trackside.
- Implementation, documentation and testing of the blueprint architecture, providing a high availability setup of the functions running on the Modular Platform. This enables continuous end-to-end communication onboard to the trackside even in cases of failures when computing elements need to reboot. The implementation is supposed to serve as a blueprint for further FRMCS utilisation by additional applications that will be integrated in the third implementation task of the project. It can be considered as major step leveraging the final setup of the demonstrator. Findings on reconnection performance collected during the test execution need further study.
- The analysis of redundancy options for the FRMCS Onboard Gateway complements the high-availability tests done for the Modular Platform. The theoretical analysis helps to bring forward the OB_{APP} redundancy discussion. Several feasible cases have been identified which can facilitate future research and product development.
- Furthermore, the realisation of FRMCS communication has been complemented by a theoretical analysis of potential deployment scenarios of FRMCS communication gateway functions on the Modular Platform.

Achievements in the Functional Cluster Diagnostics:

- Integration of the initial MCP-DIA (SOVD-Server) A-Prototype delivery into the Modular Platform as pre-compiled artefacts for the defined target platform
- Tool-driven definition and build of the diagnostics API, enabling demanded communication paradigms and flexible data structure declaration, integrated to a demo application as Protobuf library

- SystemHealthStatusApp (provided in source code) as a reference Basic Integrity Application implementation, utilising the API and Protobuf library as an example Model-3 TaskSet for diagnostics faults and measurements
- Provision of a MDCM-Mock-up client to test the SOVD-Server REST API and to visualise acquired diagnostics data with the ability to query adaptable data structures defined with the authoring tool of the SOVD interface
- Execution of end-to-end diagnostics workflows in the laboratory between the target platform and the test setup, utilising an automated cloud-based build and deployment environment using Azure DevOps pipelines combined with a customisable laboratory setup

The results shall serve stakeholders to derive concrete further in-depth judgement for final product feasibility decisions in a multi-vendor setup. Of course, it must be acknowledged that further analysis and potential interoperability measures have to be performed for final CCS-onboard solutions.

Overall, a substantial step has been achieved with regards to the Onboard Platform Demonstrator, where apart from the achievements in the functional point of view as described above, also major steps towards deployment efficiency have been taken. The demonstrator has increased to the aimed technical readiness level, utilising scalable solutions as a well-structured and maintained implementation, supporting the envisaged project tasks and team integration. The presented results constitute a major achievement and represent a solid ready to use basis for the upcoming activities, ease potential for further evolution towards the final target setup of the work package and paving the way for further evolution in R2DATO.

REFERENCES

[1] Europe's Rail. (2023). R2DATO, D36.1 – Demonstrator Specification Deliverable. Retrieved from: https://projects.rail-research.europa.eu/eurail-fp2/deliverables/

[2] Europe's Rail. (2023). R2DATO, D36.1 – Demonstrator Specification, User Stories & Test Cases. Retrieved from: https://projects.rail-research.europa.eu/eurail-fp2/deliverables/

[3] Europe's Rail. (2023). R2DATO, D36.1 – Demonstrator Specification, Statement of Work. Retrieved from: https://projects.rail-research.europa.eu/eurail-fp2/deliverables/

[4] Europe's Rail. (2023). R2DATO, D36.1 – Demonstrator Specification, Architecture. Retrieved from:https://projects.rail-research.europa.eu/eurail-fp2/deliverables/

[5] Europe's Rail. (2023). R2DATO, D36.2 – Demonstrator implementation phase 1 concluded and test results consolidated. Retrieved from: https://projects.rail-research.europa.eu/eurail-fp2/deliverables/

[6] Opinion ERA/OPI/2024-10 - Future Railway Mobile Communication System - System Requirements Specification - AT-7800. Retrieved from: https://www.era.europa.eu/content/opinion-eraopi2024-10-european-union-agency-railways-regarding-version-2-future-railway

[7] Opinion ERA/OPI/2024-10 - Future Railway Mobile Communication System – Form Fit Functional Interface Specification – FFFIS-7950. Retrieved from: https://www.era.europa.eu/content/opinion-eraopi2024-10-european-union-agency-railways-regarding-version-2-future-railway

[8] ASAM. (2022). SOVD, Service-Oriented Vehicle Diagnostics, API Specification, version 1.0.0. ASAM. Retrieved from https://www.asam.net/standards/detail/sovd/

[9] RUST (2025) programming language, version 1.86.0. Retrieved from https://www.rust-lang.org/ and https://doc.rust-lang.org/book/

[10] RUST (2025) safety critical consortium. Retrieved from https://foundation.rust-lang.org/news/announcing-the-safety-critical-rust-consortium/

[11] Google Protocol Buffers (2025), version v30.2. Retrieved from https://protobuf.dev/programming-guides/proto3/

[12] SPT2-Transversal Systems SD2 System Analysis FRS Generic Diagnostic Concept. Available in R2DATO, to be retrieved from https://rail-research.europa.eu/task-2-ccs/

[13] OCORA (2025) SRS Monitoring, Diagnostics, Configuration & Maintenance subsystem. Retrieved from: https://github.com/OCORA-Public/Publications/blob/master/00_OCORA%20Latest%20Publications/Latest%20Release/OCORA-TWS08-030_MDCM-SRS.pdf

[14] Multipurpose Internet Mail Extensions (1996) Format of Internet Message Bodies. Retrieved from https://www.rfc-editor.org/rfc/rfc2045.html
Multipurpose Internet Mail Extensions (1996) Media Types. Retrieved from https://www.rfc-editor.org/rfc/rfc2046.html

[15] Security Multiparts for MIME (1995) Multipart/Signed and Multipart/Encrypted. Retrieved from https://www.rfc-editor.org/rfc/rfc1847.html

[16] Curl (since 1998) Command line tool and library for transferring data with URLs. Retrieved from https://curl.se

[17] UIC (2023) FRMCS FFFIS Specification 7950-1.2.0 Retrieved from https://uic.org/railsystem/telecoms-signalling/frmcs