D2.4

Report on TCMS framework instantiation

Project number:	730830
Project acronym:	Safe4RAIL
Project title:	Safe4RAIL: SAFE architecture for Robust distributed Application Integration in roLling stock
Start date of the project:	1st of October, 2016
Duration:	24 months
Programme:	H2020-S2RJU-OC-2016-01-2

Deliverable type:	Report
Deliverable reference number:	ICT-730830 / D2.4 / 1.1
Work package	WP2
Due date:	March 2018 – M18
Actual submission date:	30th of March, 2018

Responsible organisation:	SIE
Editor:	Hongjie Fang
Dissemination level:	Public
Revision:	1.1

Abstract:	This report describes the design instantiation of the proposed Functional Distribution Framework based on AUTOSAR, PikeOS and INTEGRITY.
Keywords:	TCMS Functional Distribution Framework, Design instantiation, AUTOSAR, INTEGRITY, PikeOS,

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730830.

Editor

Hongjie Fang (SIE)

Contributors (ordered according to beneficiary numbers)

Hongjie Fang (SIE)

Iñigo Odriozola, Ekain Azketa, Asier Larrucea (IKL)

Jan Löwe (IAV)

Executive Summary

This deliverable summarizes the TCMS Functional Distribution Framework (FDF) design of Safe4RAIL project in conceptual and structural view and aims to provide a feasibility report to bridge the FDF concept design and the FDF proof-of-concept implementation.

This document provides a detailed report of three design instantiations (AUTOSAR Adaptive Platform (in this text denoted by AUTOSAR), PikeOS and INTEGRITY), based on the TCMS FDF concept design in D2.3 and based on the safety and security concepts described in Chapters 3 and 4, respectively of this deliverable. This report describes the design instantiation in AUTOSAR of the FDF design concept, in order to check the feasibility of mapping the defined elements and components to the building blocks of AUTOSAR. The documentation also characterizes the design instantiation in mixed-criticality PikeOS hypervisor as well as in RTOS INTEGRITY. In these two design instantiations, the conceptual elements of the FDF design are mapped to the paradigms in PikeOS and INTEGRITY. The services of the structural design are also analyzed to cope with the APIs provided by PikeOS and INTEGRITY.

Finally, the deliverable defines the unique set of APIs for the hosted applications of the framework to ensure the portability of the applications which will standardize interfaces of the next generation TCMS framework, in line with Shift2Rail objectives.These three approaches are viable to provide a basis for next generation TCMSs. PikeOS and INTEGRITY are well proven and tested in lots of ECUs in various safety-related applications. AUTOSAR, on the other hand, is still in development and aims specifically at the automotive domain, omitting some railway-relevant features and leaving open questions when it comes to certification, as both domains take different approaches and railway applications usually require a higher safety level.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 About this document

The main task of WP2 of Safe4RAIL is to provide the “Functional Distribution” architecture concept for a mixed criticality embedded platform, offering an execution environment for multiple Train Control and Monitoring System (TCMS) application functions with a virtual bus inside the end-system.

This document aims at providing a detailed report of three design instantiations (AUTOSAR, PikeOS and INTEGRITY), based on the TCMS framework concept design in D2.3. This report describes the design instantiation in AUTOSAR of the “TCMS framework” design concept and characterizes the design instantiation in mixed-criticality PikeOS hypervisor as well as in RTOS INTEGRITY. This instantiation report covers both the FDF conceptual design and the structural design, meanwhile taking the defined safety and security concepts into account described in D2.3, Chapters 3 and 4, respectively.

According to the comparative analysis of ARINC 653 in D2.2, the execution environment defined by ARINC 653 targets at static system configuration, which deviates from the dynamic system configuration requirement for next generation TCMS. Therefore, this report does not cover the instantiation of the TCMS framework concept based on ARINC 653.

This deliverable will be organized in this way: Chapter 1 summarizes the Functional Distribution Framework (FDF) design in D2.3. The Functional Distribution Services (FDS) design for the FDF will be characterized in Chapter 2. Chapter 3 describes the FDF design instantiation in AUTOSAR, taking the FDS defined in D2.3 into account. Chapter 4 and Chapter 5 provide the design instantiation based on INTEGRITY and PikeOS. In Chapter 6, the summary and conclusion are provided.

1.2 Functional Distribution Framework design

The Functional Distribution Framework (FDF), the application framework concept for modular integration of TCMS applications, aims to host distributed safety-critical and non-critical application side-by-side on the same hardware platform in distributed next-generation TCMS systems. The goal of this mixed-criticality application is to provide solutions to fulfil functional safety-critical and non-critical requirements and non-functional requirements (including security) that support functional distribution, interoperability, reconfiguration, deterministic inter-partition communication, hardware and communication abstraction and virtual coupling of services. For more information, please consult Safe4RAIL deliverable “D2.3 – Report on ‘TCMS framework concept’ design, security concepts, and assessment” [2].

1.2.1 Conceptual view

The Functional Distribution Framework uses a set of physical and logic elements that interact with each other. The physical ones are mainly:

Partition: Execution environment with an isolated memory address space and limited execution time. With it we achieve temporal separation.

Process: Thread or group of threads with an isolated memory address space, with which we achieve spatial separation.

Shared Memory: Memory that may be simultaneously accessed by multiple processes.

Apart from these, we would have the NIC, IO Card or the system clock.

Regarding the logic elements, three main ones are identified:

Function: Schedulable software that executes some logic. It can be of two types:

> Service Functions: These functions are provided by the FDF to offer its services. They have configurable logic and can be of as many types as services provided by the FDF: IO, Time, Message, Network, Monitoring and Log etc.
>
> Application Function: This type of function I provided by the user to offer the application logic. In order to achieve the abstraction we want, these functions are registered in the Framework using its API and can only have access to Variables Shared Memories.

Variable: Data structure to share information between Functions.

Message: Data structure to share Variables between remote Functions.

1.2.2 Structural view

The Functional Distribution Framework is structured in a set of Software Components which handle the management of a given service provided by the framework. Some components may have different versions for different SILs, which implement the safety mechanisms for that SIL. FDF components are grouped in three blocks, depending on the service they provide, in order to ease the portability that it is intended to achieve in this project:

- **Hardware Access Services**: These components have the same interface but different implementations for different sort of hardware. They provide IO, NIC or Watchdog management. This block can be completely implemented or, alternatively, wrapper functions can be developed to access services provided by the underlying Drivers.
- **Operating System Services:** These components have the same interface but different implementation for different Platforms/Operating Systems. This block can be completely implemented or, alternatively, wrapper functions can be developed to access services provided by the underlying Operating System.
- **Functional Distribution Services:** The code of these components is portable across different Platforms/Operating Systems because the Hardware Access Service and Operating System Service layers provide specified interfaces.

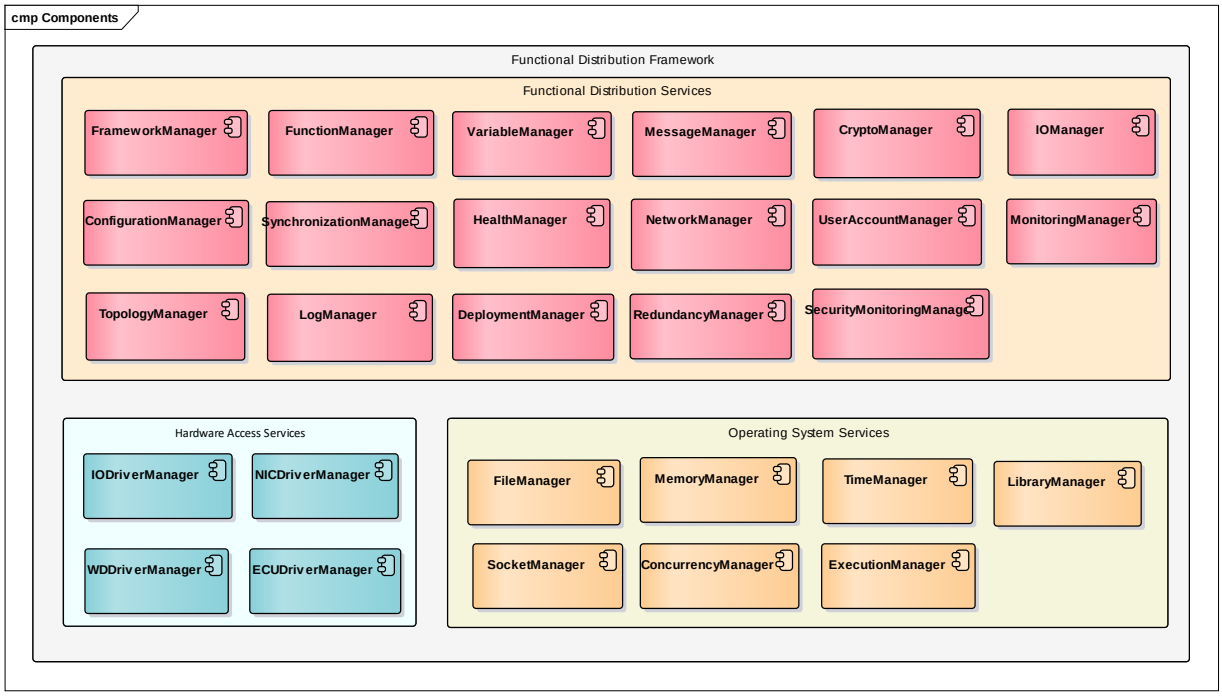

Figure 1 - Functional Distribution Framework

1.3 Functional Distribution Framework design instantiation

The FDF design instantiation report consists of three parts. The first candidate is the AUTOSAR that aims to create and establish a standardized software architecture for automotive electronic control units. The design instantiation on AUTOSAR provides a detail report, how the conceptual and structural design of the FDF could be mapped to the elements and components in AUTOSAR, meanwhile take the safety and security concepts into account. The other two options are based on the RTOS INTEGRITY and the hypervisor PikeOS. For these design instantiations, since INTEGRITY and PikeOS are both operating systems running under the FDF, the major point in these design instantiations will be how to map the designed components in the FDF to the paradigms of both operating systems.

According to the D2.3, the safety and security concepts for the FDF are proposed and the corresponding countermeasures are specified for the FDF components, hence, this report concentrates on the instantiation of the FDF design concept.

Chapter 2 Functional Distribution Services Design

Since the functional distribution services are defined to make use of the hardware access services and the operating system services to provide the defined functionalities, these services are portable across different platforms and operating systems. There are three proof-of-concept instantiations (INTEGRITY, PikeOS and Adaptive AUTOSAR) where the functional distribution services may be applied. In this instantiation report, a unique set of FDF APIs is defined, so that the applications hosted by the FDF will have the same view of the FDS

2.1 FrameworkManager

The FrameworkManager can be considered the brain of the Functional Distribution Framework. This component offers a complete set of functions to setup the different services of each instance with its only interface, IFrameworkManager. It commands the ConfigurationManager to load the configuration and initializes all the services, which means that once the configuration is read, all the required logic elements are created, logic and drivers are loaded and instances created. With register() and a set of getter calls it allows applications to be registered to be executed by it and the ability to access variables in the shared memories, respectively. Apart from the standard variables it may also provide the possibility to access and go through concrete complex structures with a given semantic such as the topology.

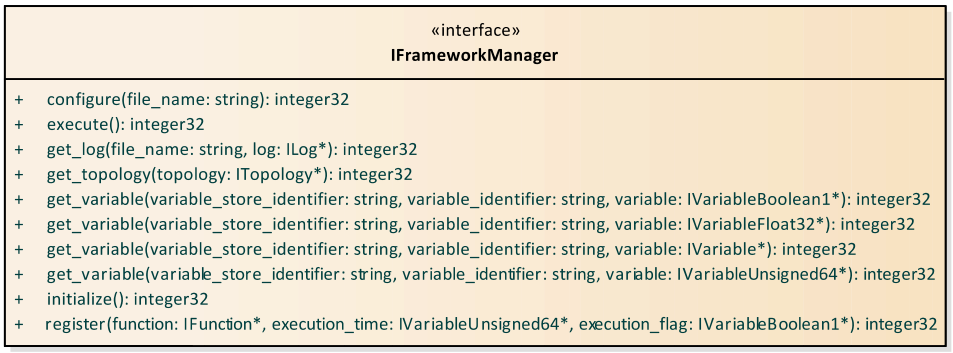

Figure 2 – FrameworkManager

Function	Description	Possible callers
Configure()	This function is used to command the ConfigurationManager to load the configuration file.	Applications
Execute()	This function starts the cyclic execution of the FDF.	Applications
Get_log()	An application can retrieve all the information of a log object.	Applications
Get_topology()	An application calls this function to get the topology-related information. It receives a reference to the topology object.	Applications.
Get_variable()	An application calls this function to access variables. It receives a reference to the Variable object.	Applications
Initialize()	It calls the initialize() function of all other managers. It creates instantiations of the different components. It loads the drivers; it creates	Applications

	the Variables and Messages Stores and Service Functions.	
Register()	An application calls this function to register its own application functions.	Applications

Table 1 – FrameworkManager

2.2 ConfigurationManager

This component provides first an initialize() function to load the selected configuration file, load it and parse it, apart from checking the loaded files CRC. The unique interface offers functions to retrieve configuration objects, once the selected configuration file is loaded. It iterates through the list of configuration items by the use of getter functions.

Figure 3 – ConfigurationManager

Function	Description	Possible callers
Initialize()	Load a configuration file, check CRC, parse a configuration file	FrameworkManager
Get_***_next()	Whole set of getters to retrieve all the information on the configuration by iterating one by one through the list of loaded objects.	FrameworkManager

Table 2 – ConfigurationManager

2.3 FunctionManager

This manager is in charge of reading through the list of functions that have been registered and executing each of them. It offers two interfaces. The first one, IFunctionManager loads the FunctionSchedule with the list of functions that needs to execute and offers the register() function by the use of which the FrameworkManager can register application functions from the outside. On the other hand, this FunctionManager implements the IFunction interface. The IFunctionManager executes this interface.

All functions provided by those interfaces have an identifier as one of the parameters, which is returned in the initialize() function.

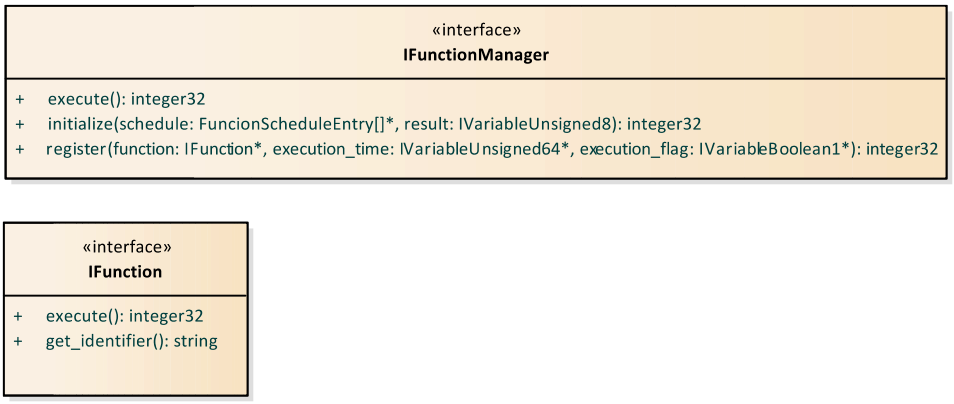

Figure 4 – FunctionManager

Function of IFunctionManager	Description	Possible callers
Execute()	Execute the registered functions	FrameworkManager.
Initialize()	Gives the configurable parameters as arguments. It gets the schedule of functions, i.e., the list of functions to be executed sequentially and the cycles in which they are executed.	FrameworkManager
Register()	Register a function, with its “execution time”, the Variable where it is stored the last execution time, and “execution_flag”, which activates or inhibits the execution of this particular function.	FrameworkManager

Table 3 – FunctionManager (part 1)

Function of IFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Get_identifier()	To retrieve the function ID	FunctionManager

Table 4 - FunctionManager (part 2)

2.4 HealthManager

It handles the health-monitoring functionalities. This component is among the most critical components since it not only makes internal checks such as CPU temperature check, but it is also responsible for refreshing the watchdog, checking the integrity of the memory or even to advise the ExecutionManager to stop or kill the faulty process.

In order to grant such health-monitoring functionalities it offers the following set of interfaces:

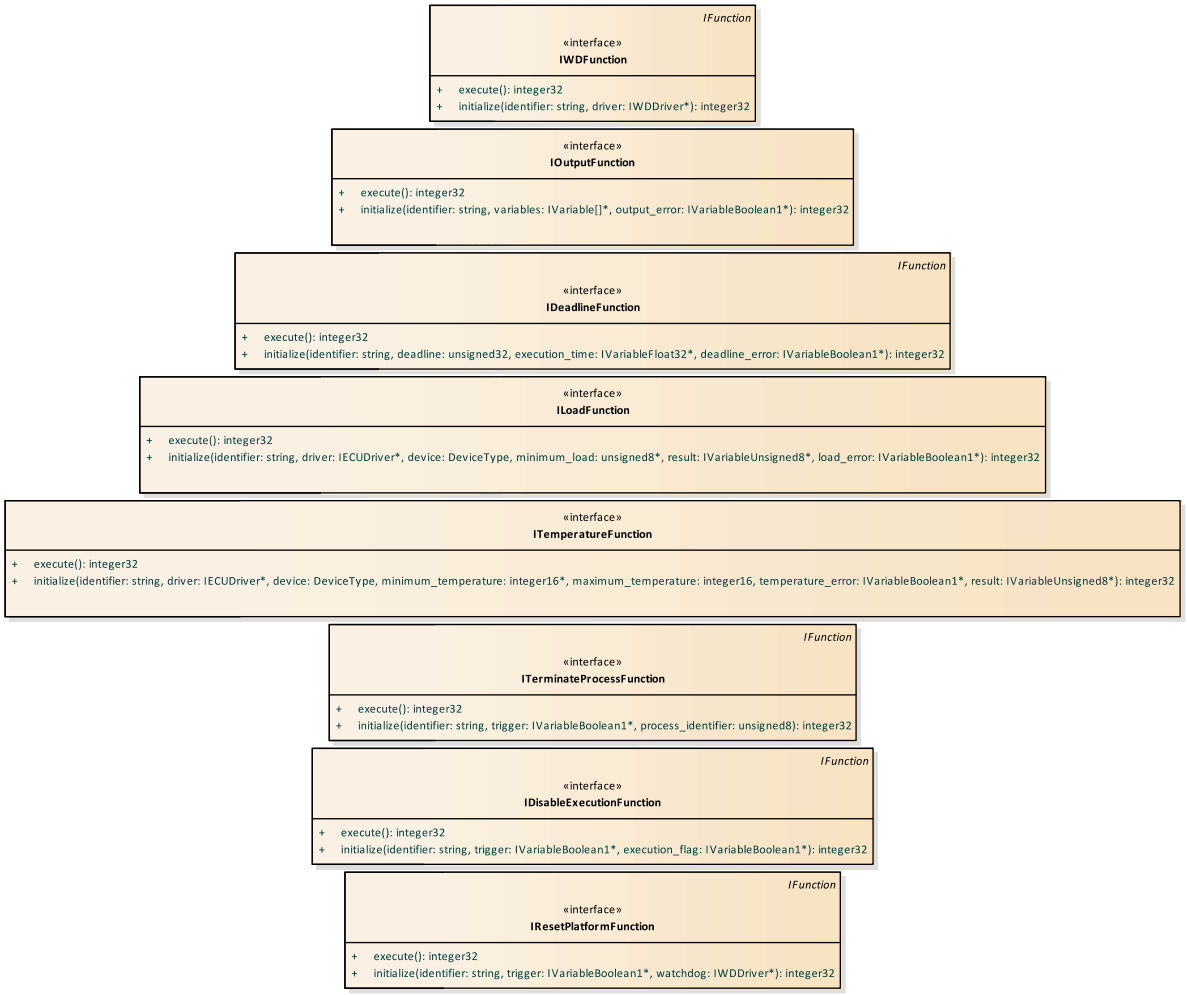

Figure 5 – HealthManager

Function of IWDFunction	Description	Possible callers
Execute()	To execute this WatchDog function	FunctionManager
Initialize()	It links the HealthFunction to the “driver” of the watchdog	FrameworkManager

Table 5 – HealthManager (part 1)

Function of IDeadlineFunction	Description	Possible callers

Execute()	To execute this function. It monitors the execution of a function. The FunctionManager measures the “execution_time” and stores it in the execution_time variable, then it compares this time to the “deadline” and acts accordingly, i.e., activate the “deadline_error” variable to indicate something is going wrong.	FunctionManager
Initialize()	When called the configurable logic is given. “Deadline”, “execution_time” and "deadline_error” are set.	FrameworkManager

Table 6 - HealthManager (part 2)

Function of ILoadFunction	Description	Possible callers
Execute()	Executes the load of a specific device through the ECUDriver.	FunctionManager
Initialize()	Initializes the load function.	FrameworkManager

Table 7 - HealthManager (part 3)

Function of IOutputFunction	Description	Possible callers
Execute()	It is read the timestamp of specified variables and it is checked if they have been updated in the current cycle or not.	FunctionManager
Initialize()	Initializes the output function.	FrameworkManager

Table 8 - HealthManager (part 4)

Function of ITemperatureFunction	Description	Possible callers
Execute()	Reads the temperature of the specified device through the ECUDriver and checks if the temperature is lower than the predefined maximum temperature and higher than the predefined minimum temperature.	FunctionManager
Initialize()	Initialized the temperature reading.	FrameworkManager

Table 9 - HealthManager (part 5)

Function of IDisableExecutionFunction	Description	Possible callers
Execute()	To execute this function. The execution flag of a given function can be set to false to deactivate this function.	FunctionManager
Initialize()	When called the configurable logic is given. The variable that triggers the command to disable is set by “trigger”	FrameworkManager

	parameter and the execution flag to be set to false in “execution_flag”.	

Table 10 - HealthManager (part 6)

Function of ITerminateProcessFunction	Description	Possible callers
Execute()	To execute this function. The health manager alerts the application logic and provides the means to kill a process when it takes too long or is constantly producing wrong variable values or variables with the bad quality flag.	FunctionManager
Initialize()	When called the configurable logic is given. “Trigger” is the variable which provokes to terminate the process “process_identifier”.	FrameworkManager

Table 11 - HealthManager (part 7)

Function of IResetPlatformFunction	Description	Possible callers
Execute()	To execute this function. It sets a variable as a trigger to stop refreshing the watchdog and with it force the reset of the platform when a hazardous situation is given.	FunctionManager
Initialize()	When called the configurable logic is given. “Trigger” is the variable which provokes to stop refreshing the watchdog “watchdog”.	FrameworkManager

Table 12 - HealthManager (part 6)

2.5 IOManager

This software component provides an interface for each of the different IO device types. It links a given variable to a channel of a concrete driver so that the value of the pin that will be read is stored in this location on the shared memory and those variables that need to be written are taken from there to provide an output.

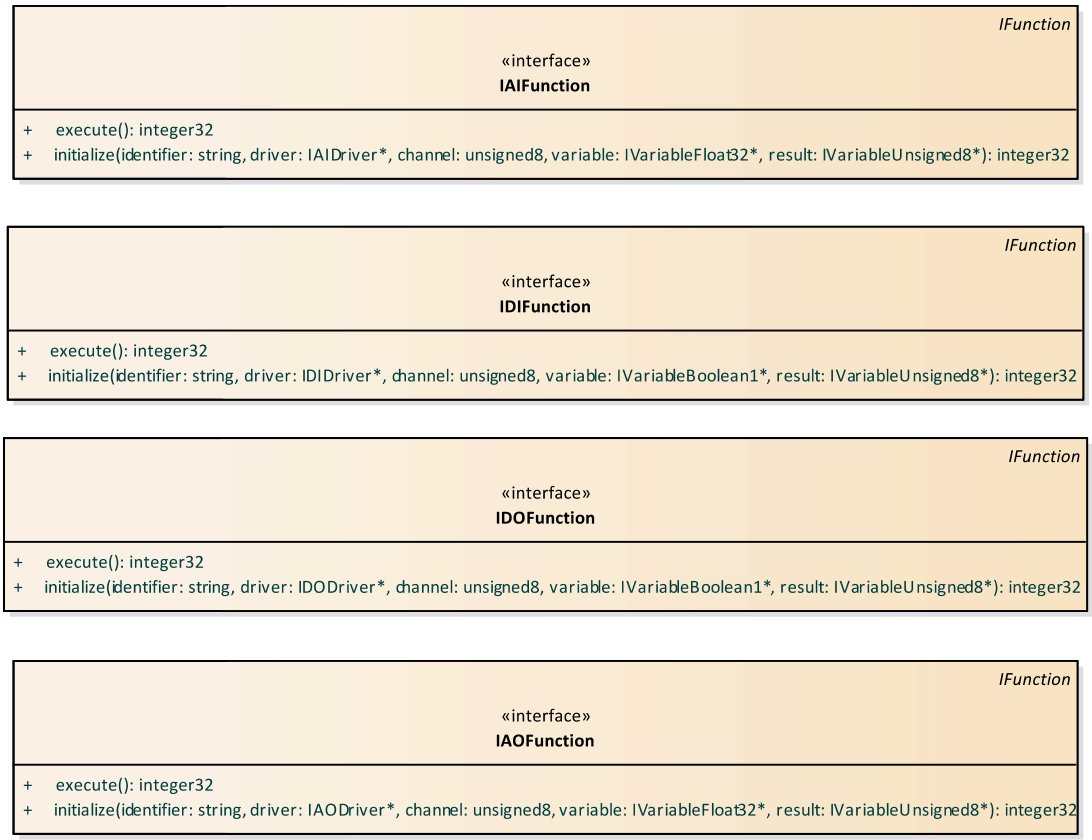

Figure 6 – IOManager

Function of IAIFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Link a “channel” of the “driver” to the “variable” in which the input will be stored.	FrameworkManager

Table 13 – IOManager (part 1)

Function of IDIFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Link a “channel” of the “driver” to the “variable” in which the input will be stored.	FrameworkManager

Table 14 - IOManager (part 2)

Function of IDOFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Link a “channel” of the “driver” to the “variable” with which the Output will be written. Indicate the Variable in which it will be indicated if the writing went wrong: “error”.	FrameworkManager

Table 15 - IOManager (part 3)

Function of IAOFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Link a “channel” of the “driver” to the “variable” with which the Output will be written. Indicate the Variable in which it will be indicated if the writing went wrong: “error”.	FrameworkManager

Table 16 - IOManager (part 4)

2.6 LogManager

This service gives the ability to write logs in log files. ILog encapsulates the file and is responsible for writing the message and introducing the timestamp without altering the format. The second interface, ILogFunction, logs a set of variables together with the message and timestamp in a log object when the trigger is set to true.

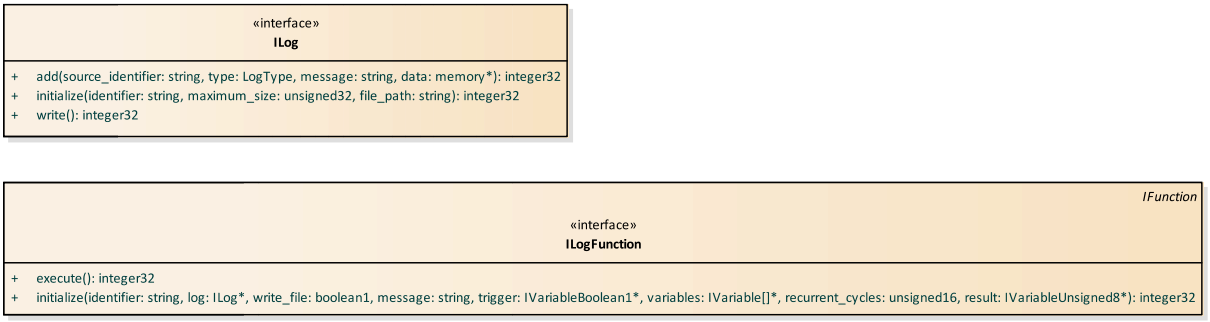

Figure 7 – LogManager

Function of ILog	Description	Possible callers
add()	Adds functions to record Log entries.	ILogFunction
Initialize()	Link the function to the file in which the log will be stored. “file_name” is a path to the file.	FrameworkManager
write()	Write a message in the log file.	ILogFunction and applications.

Table 17 – LogManager (part 1)

Function of ILogFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Link this concrete logFunction to a “log” object. When the “trigger” is true, the value of a set of “variables” is stored along with a “message” in the log.	FrameworkManager

Table 18 - LogManager (part 2)

2.7 VariableManager

This component handles the management of the variables in the shared memories. It contains a generic interface IVariable to create variables by giving the set of parameters needed to form a variable, i.e., the type of variable and location in both regular shared memory and the mirrored one, the size it takes and whether concurrent access is permitted or not. The interface also provides functions to get the timestamp, the size and variable type, but also the quality and whether it is forced or not.

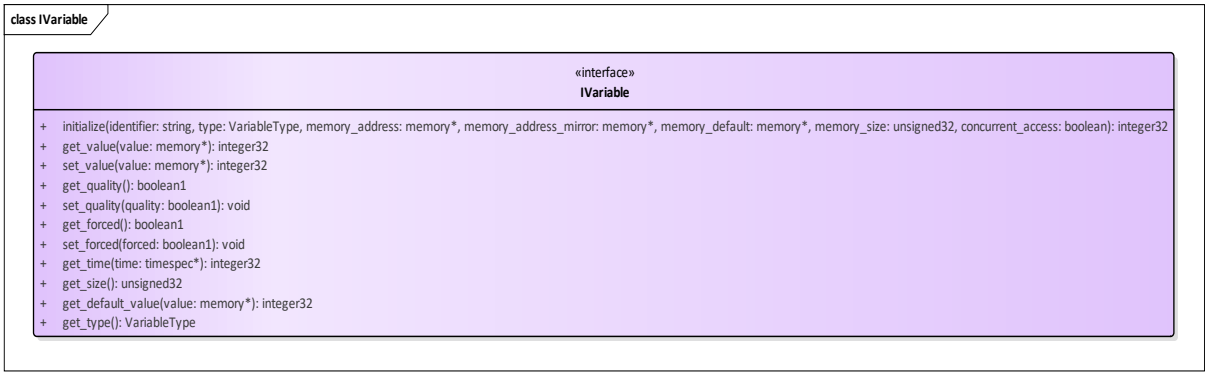

Figure 8 - General VariableManager

This generic interface is then inherited by that of every sort of different basic types such as String, Boolean, Float, integer or unsigned integer. The initialize() function of every one of these interfaces requires type-specific data such as minimum and maximum value in the case of the Float, for instance.

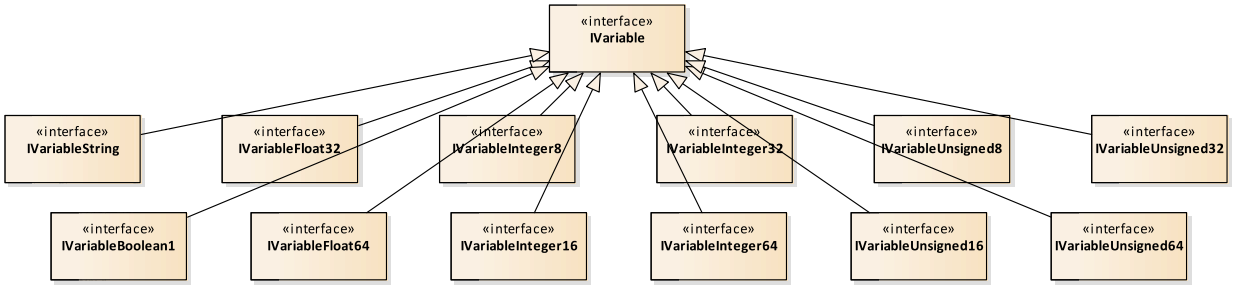

Figure 9 - Specific VariableManager

Function of IVariable	Description	Possible callers
Initialize()	Create an object for the Variable which is linked to a particular position in memory. Give the type, size, default value, whether it is possible to access it concurrently or not.	FrameworkManager
Get_/Set_***	A set of getters and setters to access and manipulate attributes of the Variable.	All

Table 19 – Specific VariableManager

2.8 MessageManager

MessageManager, by the use of IMessage, provides the ways to create Message type structures and get and set those in the corresponding shared memory. Besides, there is also the possibility to create Message parsing functions as well as functions to compose a given Message by packing together a set of Variables by the provided IParseFunction and IComposeFunction, respectively.

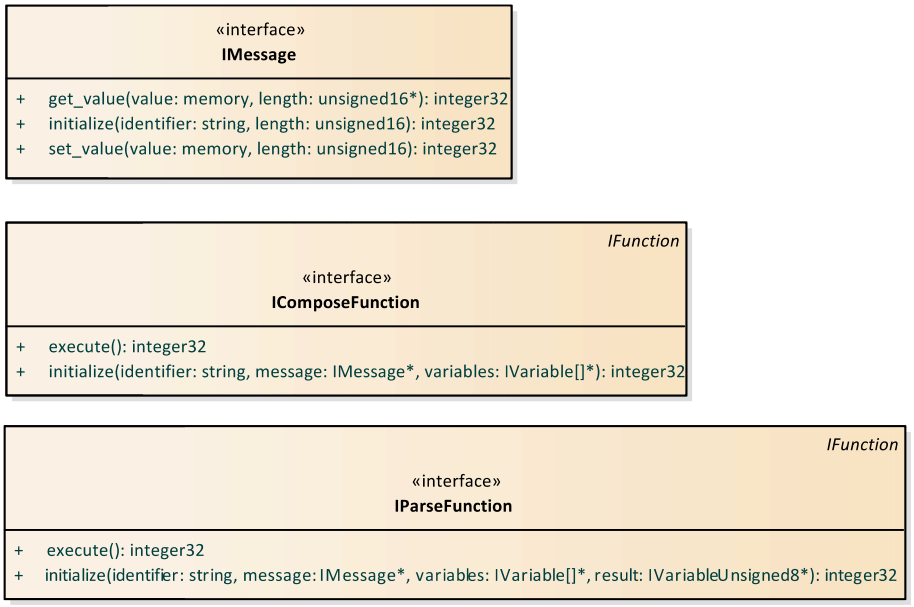

Figure 10 – MessageManager

Function of IMessage	Description	Possible callers
Get_value()	Get message data from specified memory_address and store it in a “buffer” of size “length”.	IParseFunction and ISendFunction
Initialize()	Create a Message object with a concrete “length”.	FrameworkManager
Set_value()	Set message data on position “memory_address”.	IComposeFunction and IReceiveFunction

Table 20 – MessageManager (part 1)

Function of IComposeFunction	Description	Possible callers
Execute()	Execute this function.	FunctionManager
Initialize()	Give configurable logic. Determine which “variables” compose the “message”.	FrameworkManager

Table 21 - MessageManager (part 2)

Function of IParseFunction	Description	Possible callers
Execute()	Execute this function.	FunctionManager

Initialize()	Give configurable logic. Determine in which “variables” the “message” is decomposed.	FrameworkManager

Table 22 - MessageManager (part 3)

2.9 MonitoringManager

Monitoring means that a given port is provided so that an external device can connect from the outside to check the values of a set of variables. These information needs to be given in the initialize() function of its interface so that the necessary permissions are given.

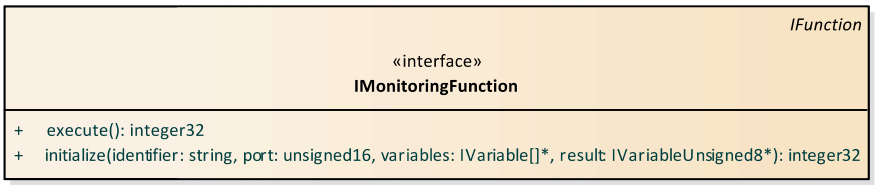

Figure 11 – MonitoringManager

Function of IMonitoringFunction	Description	Possible callers
Execute()	To execute this function	FunctionManager
Initialize()	Enables the monitoring of a set of “variables” through a given “port”.	FrameworkManager

Table 23 – MonitoringManagermo

2.10 NetworkManager

All the interactions with the NIC device going through the sockets of the operating System are handled by this Manager. It mainly adds and checks the application layer information on the Messages heading or coming from the network. Its only interface offers two functions, one for outgoing data and another one for incoming one, both of which need the name of the identifier of the message, the port to be used and an address.

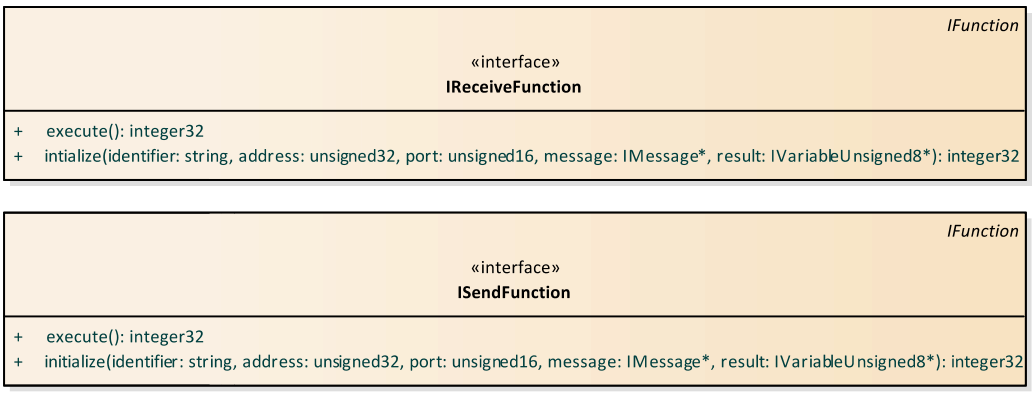

Figure 12 – NetworkManager

Function of IReceiveFunction	Description	Possible callers
Execute()	Execute this function to receive a message	FunctionManager
Initialize()	Gives the configurable logic. Determine in which “port” this function will receive the “message” from.	FrameworkManager

Table 24 – NetworkManager (part 1)

Function of ISendFunction	Description	Possible callers
Execute()	Execute this function to send a message	FunctionManager
Initialize()	Gives the configurable logic. Determine the use of which “port” this function will send the “message”.	FrameworkManager

Table 25 - NetworkManager (part 2)

2.11 RedundancyManager

The FDF makes all sort of redundancy handling in this component. It handles partition redundancy, output redundancy and provides the possibility to make cross-monitoring when voting is involved in a given topology for the IMP, for instance. In its initialize() function, concretely, it is indicated which variable it needs to listen to in the ECU which is in stand-by in order to determine whether the master ECU is alive or not. For that, a timeout needs to be provided together with the execution flags of the function to activate when a hot-swap needs to be carried out.

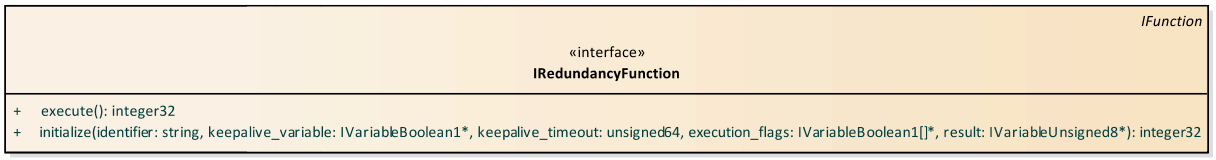

Figure 13 – RedundancyManager

Function of IRedundancyFunction	Description	Possible callers
Execute()	Execute the function.	FunctionManager
Initialize()	Gives the configurable logic to the function. The “keepalive_variable” is updated every certain time. The master node sends it to indicate that it is alive. If this variable has not changed in a “keepalive_timeout” time, then the master ECU can be considered down. “execution_flags” indicates which flags need to be set to true, i.e., which functions need to start publishing some outputs now that the master ECU is down in order to perform the hot-swap.	FrameworkManager

Table 26 - RedundancyManager

2.12 SynchronizationManager

This manager is responsible for updating the system clock when the global tick arrives from the network. For this duty, it offers an initialize() function to link the instance of the TimeFunction service function which will be created to the driver of the NIC which will provide this information.

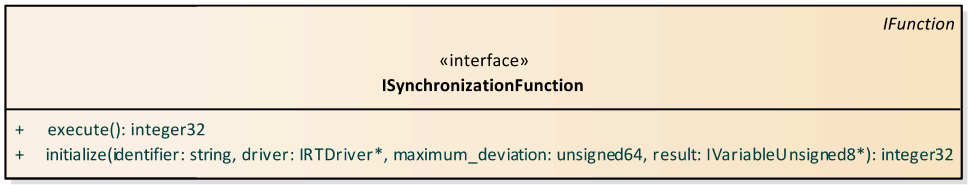

Figure 14 – SynchronizationManager

Function of ISynchronizationFunction	Description	Possible callers
Execute()	Execute this function.	FunctionManager
Initialize()	Indicate the “driver” from which the function will get the global time in order to update the system clock time.	FrameworkManager

Table 27 – SynchronizationManager

2.13 TopologyManager

In this manager the changes on the topology are handled. It offers an interface to create topology objects and another one to link a given port and address to the topology that is retrieved when making a TTDB request through this concrete port.

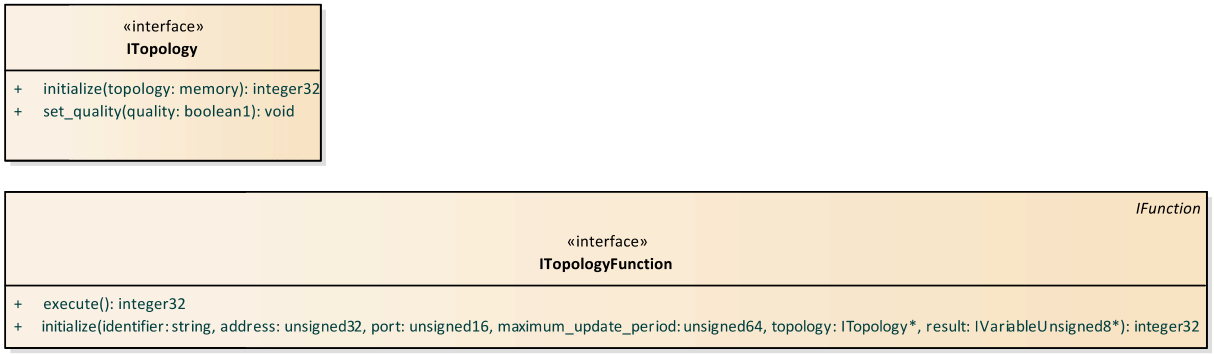

Figure 15 – TopologyManager

Function of ITopology	Description	Possible callers
Initialize()	Map a given slot “memory_address” in the shared memory to a concrete topology object.	FrameworkManager
Set_Quality()	Sets the quality of the topology to ITopologyFunction	FrameworkManager

Table 28 – TopologyManager (part 1)

Function of ITopologyFunction	Description	Possible callers
Execute()	Execute this function.	FunctionManager
Initialize()	Provide the “port” which the function will make a request to update the topology information in the “topology” object.	FrameworkManager

Table 29 - TopologyManager (part 2)

2.14 DeploymentManager

The deployment manager provides service to update configuration files and executables remotely.

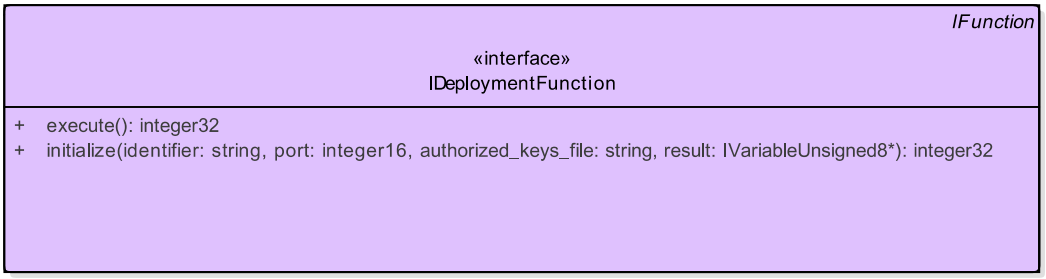

Figure 16: DeploymentManager.

Function of IDeploymentFunction	Description	Possible callers
Execute()	Execute this function.	FunctionManager
Initialize()	The “port” from which clients will be making requests to the secure file transfer protocol is given together with the “authorized_keys_file”, which contains the list of authorized clients on the server. Finally, the “result” is indicated as configurable logic, where the result of the transaction will show whether everything went alright or an error occurred.	FrameworkManager

Table 30 – DeploymentManager

2.15 CryptoManager

This software component is the result of the Security Concept and it takes care of the cryptographic services in the FDF. It provides the interfaces listed below, each of which will consist in an initialize() function, in which all necessary parameters to perform the corresponding functionality are given, and an execute() function to actually executing it. They access Variables and Messages:

- **IPublicKeyGenerationFunction:** This function generates the corresponding public key.

- **IPublicKeyVerificationFunction:** This function performs an embedded public key validation.
- **IKeyManagementFunction:** This function deals with storage, use and deletion.
- **ISignatureGenerationFunction:** This function generates the corresponding signature for a hashed message.
- **ISignatureVerificationFunction:** This function verifies the corresponding signature for a hashed message.
- **IHashGenerationFunction:** This function computes the hashing for a block of data, based on MAC key, key length, data and data_length.
- **IHashVerificationFunction:** This function verifies hash.
- **IEncryptFunction:** This function encrypts data based on cipher algorithm, user key, key length, and so on to generate the cipher text from a plaintext.
- **IDecryptFunction:** This function decrypts data based on cipher algorithm, user key length, and so on to generate the plaintext from a cipher text.
- **IBase64EncodeFunction:** This function encodes data in base 64. This will be a way to protect data from human-readable format, and when
- **IBase64DecodeFunction:** This function decodes data in base 64.

2.16 UserAccountManager

This software component is the result of the Security Concept and is responsible for managing user accounts. It provides four interfaces, each of which will contain an initialize() function, in which all necessary parameters to perform the corresponding functionality are given. IUserManagementFunction contains concrete functions to manage user accounts whereas the rest of the interfaces have an execute() function to run the corresponding action:

- **IUserManagementFunction:**
    - **create():**This function generates a unique identifier for the user account being created, and adds all parameters required for user profile, such as, first name, surname, email, user manager, role, current password, previous password etc.. A certain number of old passwords will be associated with a user account to verify passwords are changed properly.
    - **delete():** it deletes a user account, if user is authorized.
    - **update():** it modifies user parameters, if user is authorized. For example, password shall be able to be changed.
- **IPasswordCheckFunction:** This function checks if password satisfies password policy, for example, alpha-numeric characters, long, and so on, during the creation of the password.

- **IPrivilegesSettingFunction:** This function based on user role assigns certain privileges or permissions, applying least privileges philosophy.
- **IKeyManagementFunction:** This function deals with assigning a key to the created user, modifying or deleting it.

2.17 SecurityMonitoringManager

This software component is the result of the Security Concept and it monitors the behaviour of the FDF in terms of service availability, session list and user and application aspects, while checking authentication issues, execution of untrusted code or notification of attacks. It provides the following interfaces, each of which will consist in an initialize() function, in which all necessary parameters to perform the corresponding functionality are given, and an execute() function to actually executing it:

- **IApplicationIdentificationFunction**: This function will assigned a unique ID to an application on the FDF. This ID will be used for tracing application behaviour, that is for monitoring correct operation of access to CPU and network, data modification and for notifying to a higher system or administrator when abnormal behaviour is discovered. This information can be stored in the TPM.
- **IApplicationProfileFunction**: This function will create an application profile based on configuration files to trace application permissions.
- **IAuthenticationVerificationFunction:** This function authenticates user based on user and password, and on the USB or smartcard containing credentials, device for example by serial number and applications based on unique identifier.
- **ISessionManagementFunction:** This function is in charge of creating a session, locking a session if timeout and closing it.
- **ILoginManagementFunction:** This function checks logins.
- **IAuditEventsConfigurationFunction:** This function enables the configuration of audit events like login, timestamps, audit trail, information for non-repudation, modification, deletion, user, location, etc.
- **IAuditReportingConfigurationFunction:** This function enables the configuration of the audit events defined for reporting. All reports will provide timestamps based on system time, and additional information considered relevant, user location etc. This information shall be encrypted and store in a secure way but means of the CrytoManager.
- **IUserLoginReportingFunction:** This function enables to list all user accounts and login history.
- **IGetReportFunction:** this function enables to get a report, only authorized users shall get this information. This information will be protected by encryption, digital signature, digital message reports and timestamps.

- **INetworkMonitoringFunction:** This function shall check all related network issues: insertion of packages, data flooding, loss of communication, replay of messages, messages to provoke a DoS attack.
- **IMonitoringunction:** This function will monitor deletion or insertion of configuration data or detection of insertion of malicious code, very critical.
- **IUseNotificationunction:** This function will inform administrator about user access and actions performed.
- **IAttackNotificationFunction:** This function shall be used in case of determination of a possible attack: access to CPU, modification of configuration files during execution or not. This communication can be done by means of email, text messages or any other means.
- **IIncidentSupportConfigurationFunction:** This function will enable the configuration of automated incident notification services to whom corresponds (user or system).
- **IIncidentNotificationFunction:** This function will notify to an authorized user or system about an incident. This can be made by e-mails, text messages, or any other means configured before.
- **IPasswordExpirationNotificationFunction:** this function shall notify user to modify password after a period of time defined by an administrator.
- **IPasswordStrenghEnforcementFunction:** This function shall guarantee that criteria defined for strength: minimum length, use of upper/lower cases, non-alpha characters etc. In FSA-AC-2.18 it is set a minimum of 6 characters for passwords.
- **IAdministratorAccessVerificationFunction:** This function will notify administrator for getting approval of a user access. This is needed to fulfil FSA-AC-1.2 Dual Approval Access. The result will be encrypted.
- **IMulticastTransmissionVerificationFunction:** This function will verify the source and integrity of the transmissions.
- **IMulticastTransmissionHandlingFunction:** This function will register authorized applications to subscribe to multicast transmission, and authorized applications enabled to send multicast transmissions.
- **IErrorHandlingFunction:** This function will handle error conditions without providing information that could be exploited by adversaries.
- **IBlacklistingCreationFunction:** This function will be used for creating blacklists to protect FDF against executable code. Administrator can use either black lists or white lists.
- **IWhitelistingCreationFunction:** This function will be used for creating whitelists to protect FDF against executable code.

- **ICommunicationVerificationFunction:** This function will check a loss in the communication for inputs/outputs or any other transmitted message to be applied upon loss of communications.
- **IBackupCreationFunction:** This function will create a backup for recovering the system either as a result of an attack or for any other reason like a failure. This backup will be at the user level and system level. Only authorized entities will be able to create it and it will be saved in the TPM.
- **IFDFRecoveryFunction:** This function will restore the system by means of a secure backup after a disruption or failure in the system.

Chapter 3 FDF design instantiation based on AUTOSAR

The AUTOSAR standard has been developed by major automotive OEMs together with soft- and hardware suppliers to simplify and accelerate the development of ECUs. While the so-called Classic Platform (CP) targets the needs of self-contained, monolithic systems, the newly developed Adaptive Platform (AP) addresses mainly high-performance systems that are able to run multiple applications in parallel. The AP bases on POSIX operating systems and pursues a service-oriented approach thus allowing high flexibility in deployment and update of applications as well as system configuration. Furthermore, it enables modular and independent development, scalability and partial updates even over the air.

3.1 Mapping conceptual design

The AP follows a service-oriented-architecture. This approach offers many advantages. For example, it supports the independence of the application software components. It allows establishing communication paths both at design- and run-time, so it is possible to build up both static communication with known numbers of participants and dynamic communication with unknown number of participants. Figure 17 shows the basic operation principle of communication in service-oriented-architectures.

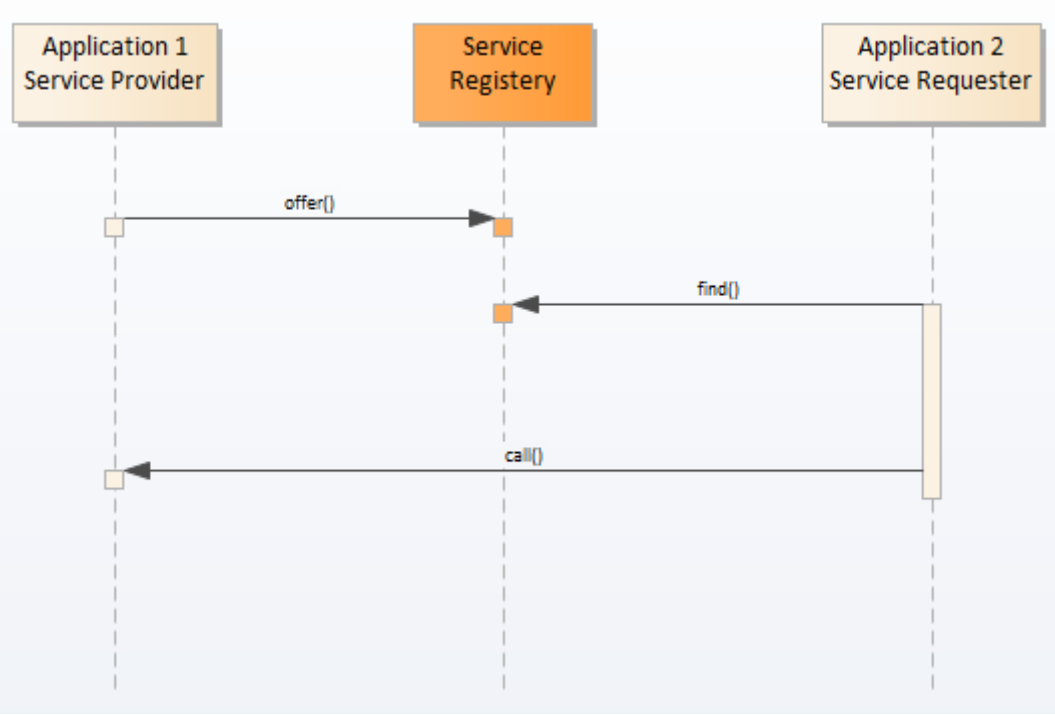

Figure 17 – Communication in service-oriented-architecture

Service Discovery decides whether external and internal service-oriented communication is established. The discovery strategy shall allow either returning a specific service instance or all available instances providing the requested service at the time of the request, no matter if they are available locally or remotely. The service definitions include port and the communication direction.
AP uses SOME/IP[1], which is a service oriented middleware specification on top of IP and has been developed with the automotive domain in focus.
Communication management software in AP provides an optimized implementation for both the service discovery and the communication connection, depending on the location where the service provider resides. The communication manager abstracts all aspects of the communication including element types, local and remote data handlings. In summary, the communication manager is responsible for all aspects of communication including the E2E[2] security protection.

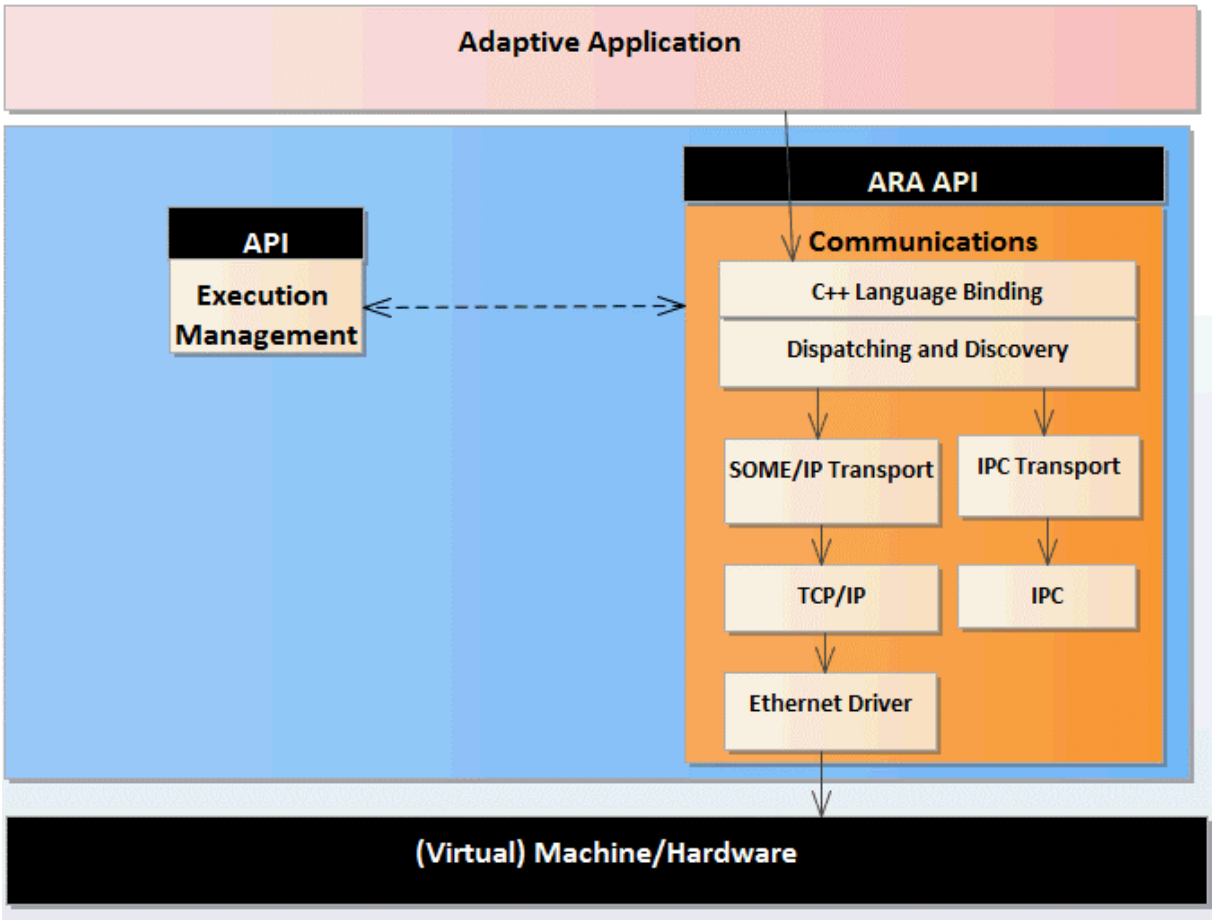

Figure 18 - Service management via the communication manager

As discussed above, the adaptive application interacts with other applications and the underlying AP framework through the services. The service based communications get routed through the communication manager. Figure 18 describes the basic AP

[1] Scalable service-Oriented Middleware over IP

[2] End-to-End

communication framework. Please note that an application might act as a service, too by providing a certain service interface.

The communication manager receives the service request, decides based on the registered services and their service manifest configurations, whether the request should be sent via SOME/IP stack to other nodes through Ethernet, or whether it should be handled via IPC[3] in the same machine.

Furthermore, it should be mentioned that the AP resides upon a POSIX compatible OS. By definition the OS should provide at least PSE51 profile's API which is specified in IEEE1003.13. This API is also available to applications.

One major drawback is mentioned in the AP standard itself:

> The Communication Management software using Service-Oriented Communication will not achieve hard real time requirements, as the implementation will behave like a virtual ethernet including latencies of communication. This behaviour must be respected with the design of the overall ECU and SW system.
> (*AUTOSAR_SWS_CommunicationManagement)*

While this document is being written, the Adaptive AUTOSAR standard is not yet complete. Thus some modules are not yet fully specified, while others might be subject to change. Apart from that mature implementations are obviously not available. Therefore latencies are subject to evaluationas soon as those are released.

The following sub-sections describe the mechanism how the high-level FDF logical elements (e.g. variables, messages, and functions), FDF physical elements (Shared-memory, Processes and Partitions) and HW-access (NIC, IOC and Clock) can be realized through the Adaptive-AUTOSAR (AP) framework.

3.1.1 Variable

In the FDF, Variables are defined as data structure to share information between parts of the applications. This definition is in line with the element 'field' in the AP service interfaces. Those fields may have further attributes such as a type, which might include a valid range, initialization value, notification enable, setter and getter function enable.

The AP communication manager reads the field configurations from the application and service manifests, creates the variable objects and writes the default or initialization values. The E2E protection in the communication manager is responsible for deadline timeout monitoring of the variables. Further, the communication manager coordinates with the IAM (Identity and Access Manager) module regarding the access control decisions. The IAM module decides the read and write access permissions to the variable objects based on the service interface attributes in the manifest files.
The major difference between the FDF concepts and the AP is that in the AP fields are stored in the providing service. Memory protection, which is defined in FDF's VariableManager, is usually left to the OS implementation, Platform health management and the CPU's diagnostic functionality. However, it is possible to implement a new service, which in co-operation with the communication manager implements memory mirroring for instance. This service could as well provide access to variable's timestamp, which are also not part of AP.

[3] Inter Process Communication

3.1.2 Message

AP applications either provide or consume data in the form of fields as described in the previous section. There is no differentiation between local and remote variables or messages. This is because the AP communication manager abstracts and manages all aspects of the communication. As for the FDF variables, the message definitions can be mapped to the element ‘field’ in the service interfaces in the service manifest file.

3.1.3 Shared Memory

AP does not define the shared memory objects. As variables are stored within the providing service, there is no need for this concept.

3.1.4 Function

3.1.4.1 Application Function

The application function implementation is framework independent and can be realized easily. The variables for the input and output can be defined in the application/service manifest files.

3.1.4.2 Service Function

AP offers many, but not all the FDF service functionalities. . For example, AP does not offer the services for managing IO-driver operations. The missing services may be implemented as adaptive services.

3.1.4.2.1 IO Function

IO handling is not in the scope of AP. The reason for this is that the AUTOSAR classic platform contains a very mature specification of IO handling. The idea is that AP applications interface to CP applications using SOME/IP to achieve IO access.

As FDF aims at a full integration of platform services to a single framework, this might not be the way to go for the Safe4RAIL project. Nevertheless, it is possible to implement the IO functionality, as it will be specified in the FDF as an additional service in the AP.

3.1.4.2.2 Time Function

AP provides the Time functionalities through the AP ‘Time synchronization’ module.

3.1.4.2.3 Message Function

The communication manager is responsible for the composition/decomposition of the messages from/to variables.

3.1.4.2.4 Network Function

The service-manifest files describe the transport layer e.g. SOME/IP properties. Based on the manifest files, the communication manager, signal-2-service mapping and network

management modules know that the message must be sent over the network using a given transport layer and will take responsibility for the correct transmission.

3.1.4.2.5 Monitoring Function

Applications may provide notifications for variables, which other applications may subscribe to. In case the value of the variable changes, the SOME/IP stack calls the notification function.

3.1.4.2.6 Deployment Function

AP Execution Manager is responsible for the application deployment.

3.1.4.2.7 Log Function

AP implements the Log functionalities through the 'Log and trace' module. This module is intended to provide logging capabilities to applications, so applications may decide which information is to be stored in the logs.

3.1.5 Process

AP implements every framework module and the adaptive applications as processes. Execution manager starts the processes.

3.1.6 Partition

The 'Explanation of Adaptive Platform Design' reads:

> *In summary, from the OS point of view, the AP and Adaptive Applications (AP) forms just a set of processes, each containing one or multiple threads – there are no difference among these processes, though it is up to the implementation of AP to offer any sort of partitioning.*

Therefore, the concept of partitioning is not a part of the AP. For the purpose of the FDF instantiation, we propose the use of a hypervisor and virtual machines to provide spatial separation of different applications of different SILs. A similar idea is already standardised and used in safety-related-applications for avionics (ARINC 653) as well as for instance in in-car infotainment systems (non-safety-related). Figure 19 depicts this approach.

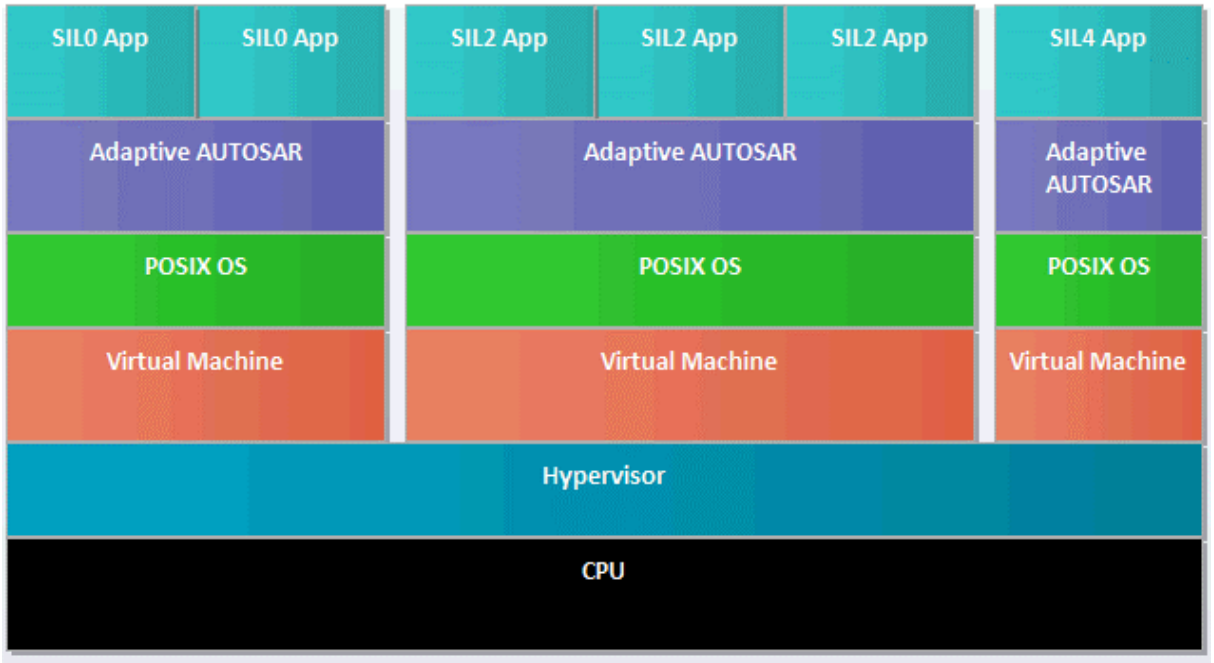

Figure 19 - Hypervisor architecture

This concept will complement the AP architecture with the concept of partitions, which are implemented by independent virtual machines. Furthermore, the usage of a hypervisor will provide an additional benefit: It enables deployment of other OSes on virtual machines on the same physical machine, which might be for instance Linux or even classic AUTOSAR applications.

3.1.7 Schedule

3.1.7.1 Partition Schedule

Partition scheduling is up to the hypervisor and thus implementation specific. Most available hypervisor implementation support partition scheduling.

3.1.7.2 Process Schedule

Execution manager is responsible for starting applications and managing the application lifecycle. Applications may use the OS interface to schedule their functions. The OS interface must at least provide the following scheduling policies defined in the IEEE1003.1 POSIX standard: SCHED_OTHER, SCHED_FIFO and SCHED_RR. Nevertheless, implementers may decide to add additional policies like SCHED_DEADLINE in order to achieve real-time requirements.

3.1.7.3 Function Schedule

A main application thread (runnable function) executes on every process execution. Application is responsible to properly configure secondary threads (runnable functions). OS can perform the thread level monitoring.

3.2 Mapping structural design

The following sub-sections describe the mechanism how the FDF software components can be realized through the AUTOSAR Adaptive Platform.

Figure 20 shows the architecture of AP. The Adaptive Applications (AA) run on top of ARA, the AUTOSAR Runtime for Adaptive applications. ARA consists of application interfaces provided by Functional Clusters, which belong to either Adaptive Platform Foundation or Adaptive Platform Services. Adaptive Platform Foundation provides fundamental functionalities of AP, and Adaptive Platform Services provide platform standard services of AP. Any AA can also provide Services to other AA, illustrated as Non-platform service in the figure.

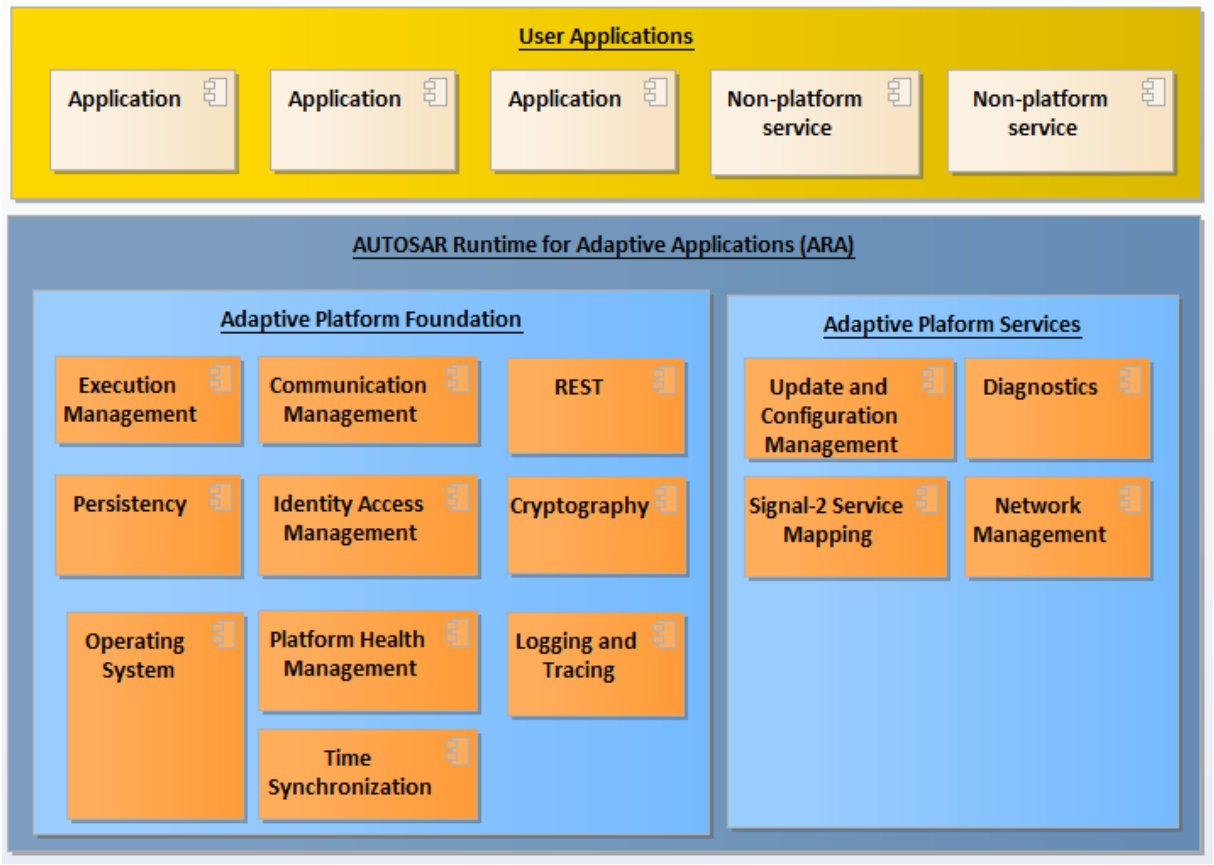

Figure 20 - AP architecture logical view

Figure 21 shows the broad service categories available in the adaptive autosar.

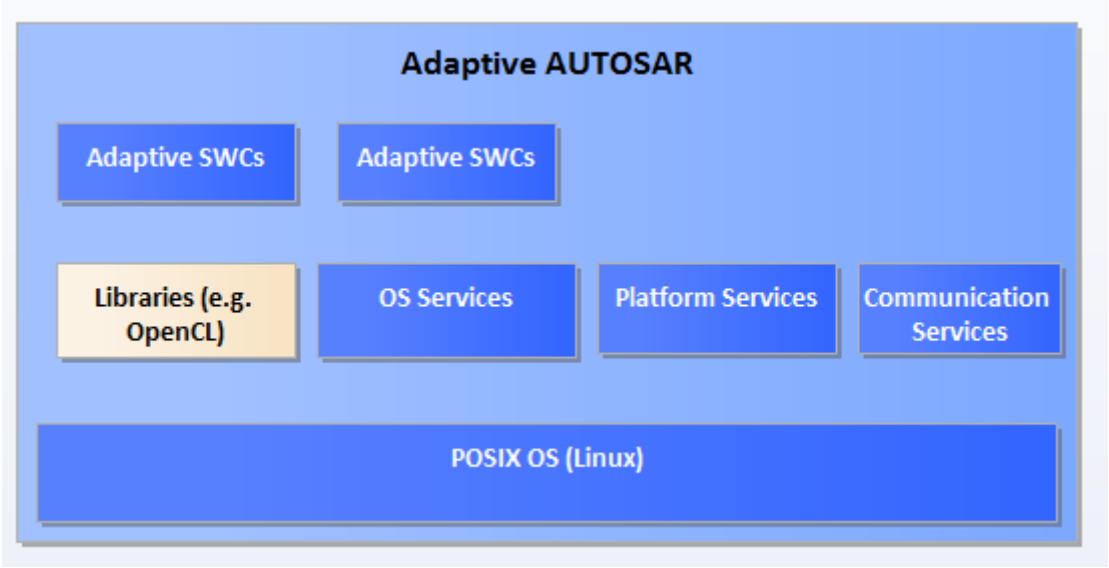

Figure 21 - AP service categories

OS-Services include Execution manager and POSIX interface (PSE51). Platform services include Diagnostics and Logging. Communication services include SOME/IP communication

3.2.1 Hardware Access Services

3.2.1.1 IODriverManager

The AP does not provide IO-driver functionalities. Additional adaptive services need to be implemented which provide the IO-driver functionalities. Note that the applications may work with the communication cluster service, which abstracts all kind of communications. The communication manager works with the newly added services to provide the IO-driver functionalities. As mentioned before it might be a possible solution to utilize classic AUTOSAR applications connected via SOME/IP.

3.2.1.2 NICDriverManager

Network Management module is responsible for implementing the access to the network interface.

3.2.1.3 WDGDriverManager

Platform Health Manager is responsible for implementing the WDG services.

3.2.1.4 ECUDriverManager

Platform Health Manager is responsible for monitoring ECU conditions.

3.2.2 Operating System Services

3.2.2.1 FileManager

This module's functionality is realized by AP's persistency module.

3.2.2.2 MemoryManager

As stated above this module is not needed in the AP instantiation.

3.2.2.3 ConcurrencyManager

OS is responsible for the implementation of the concurrency manager functionality, which includes management of threads, mutexes and semaphores.

3.2.2.4 TimeManager

Time synchronization module is responsible for the implementation of the clock manager functionality, which includes getting and setting the system clock time.

3.2.2.5 SocketManager

AP communication manager may include the socket manager functionality, which is used for the network message transmissions. Additionally, the socket manager may also be implemented as an adaptive service. The communication manager may then use this newly added service for the network message transmissions.

3.2.2.6 LibraryManager

Dynamic libraries are handled by the underlying operating system.

3.2.2.7 ExecutionManager

Execution management module is available in the AP. However its role is different from the FDF execution manager. For example in FDF, the execution manager is responsible for the partition and process schedule management. In AP, the execution manager is responsible for configuring the OS based on the information in application and machine manifest files. In AP, the hypervisor and OS are responsible for the partition and process schedule management.

3.2.3 Functional Distribution Services

3.2.3.1 VariableManager

As stated above this module is not needed in the AP instantiation, as variables are stored in the providing service (see 3.1.1).

3.2.3.2 MessageManager

In AP, the communication manager performs the activities, which are performed by the FDF message manager. For example, the communication manager, together with the signal-2-

service mapping module, composes and decomposes the messages. The E2E module in the communication manager is responsible for controlling the integrity of messages (e.g. CRC).

3.2.3.3 ConfigurationManager

Configuration manager cluster is available in AP. However, its role is rather to update the software components, configuration and calibration data. In AP, the application manifest files provide the configuration data for the clusters. The execution manager performs loading, parsing and checking the coherency of the configurations.

3.2.3.4 NetworkManager

Communication manager performs the activity of the sending/receiving the messages from/to message storage to/from network.

3.2.3.5 MonitoringManager

The E2E module in the communication manager performs the monitoring of the messages. Communication manager coordinates with the access-control module in security manager cluster for the access-control decisions. The access-control information in the application manifest files servers as the basis for the access-control decisions.

3.2.3.6 IOManager

IO access is not in the scope of AP. Possible solutions have been mentioned in 3.1.4.2.1 and 3.2.1.1.

3.2.3.7 SynchronizationManager

AP Time synchronization module provides the Time and Synchronization management functionalities

3.2.3.8 FunctionManager

The OS module handles scheduling in the AP. The Execution Manager is responsible for application lifecycle management.

3.2.3.9 FrameworkManager

As there is no VariableStore or MessageStore in the AP, this functionality is not needed. Functions are registered via Execution Management and OS modules.

3.2.3.10 HealthManager

The AP execution manager, in coordination with security cluster modules, performs the activities related to integrity checks before their deployment. The execution manager also reads the application and machine-manifest file information and applications configure the OS. Note that the OS is primarily responsible of CPU time, memory budgeting as well as partition and process monitoring.

AP Platform-health-manager cluster performs the WDG and IO-hardware monitoring. The cluster is also responsible for handling the reactions resulting from any anomaly.

3.2.3.11 LogManager

Log module in the AP manages the logging functionality. The applications create the logger instances, initialize loggers and later initiate data logging. However, in AP the idea is that applications actively provide messages to the log module. Therefore, to achieve the functionality designed in the FDF an additional service, which monitors Variables and creates corresponding log messages needs to be implemented.

3.2.3.12 TopologyManager

AP does not provide the service for topology management. The functionality however can be implemented as an adaptive service.

3.2.3.13 RedundancyManager

AP does not provide redundancy management functionality. However, the functionality can be implemented as an adaptive service.

3.2.3.14 DeploymentManager

Execution manager performs the deployment of the clusters and applications.

3.2.3.15 CryptoManager

Cryptographic functions are supplied by AP's cryptography module.

3.2.3.16 UserAccountManager

The AP module "Identity and Access Management" is designed to provide access management based on access control policies, which also include user/entitity management.

3.2.3.17 SecurityMonitoringManager

The Identity and Access management module together with the Platform Health Monitoring module provide this functionality.

3.3 Summary and Conclusion of Adaptive AUTOSAR Instantiation

Although there are some conceptual differences between Adaptive AUTOSAR and the FDF, the basic ideas and goals are similar. Both approaches try to simplify ECU development and ECU cost by providing mechanisms to deploy mixed-criticality applications to a single host while guaranteeing spatial isolation and interference-free execution.

Table 31 provides an overview, which elements of the FDF concept can be mapped to AP elements and how.

Nevertheless, there are two major concerns, which need to be solved in the subsequent works:

1. Evaluate how the FDF API can be mapped to the AP or rather how the AP API can be wrapped into the FDF API: The concept of service-oriented communication used in AP differs from the concept of a centralized repository for variables in the FDF (the VariableManager and MessageManager modules). This issue needs to be addressed as soon, as the application API is specified by CONNECTA.
2. The other uncertainty is that AP explicitly not defined to achieve hard real time requirements due to the fact, that the service-oriented approach systemically adds communication latencies. AP currently not specifies mechanisms to provide any real time limitations to communication except for timeout detection. Nevertheless, it can be assumed that COTS implementations will be able to guarantee maximum transfer times. Furthermore, it is very likely that such requirements will be added in future AP specification releases.

Element	Mapping available[4]	Restrictions	Comments
Variable	L	Variables managed in containing service, not in a VariableManager	
Message	L	Treated the same way as variables.	
Variable Memory	N	Not needed due to the concept of service-oriented communication	Variables are stored in the containing/providing service.
Message Memory	N	Not needed due to the concept of service-oriented communication	
Application Function	F		Mapped to the AP element ‘executable’
Service Function	F		Mapped to the AP element ‘executable’
Process	F		POSIX processes are used
Partition	N/F	AP needs to be combined with a hypervisor to achieve partitioning	
Partition Schedule	N/F	Only in combination with a hypervisor	
Process Schedule	L	Applications are entitled to control their scheduling policy	Provided by the OS module

[4] N – Not available, P – Partially available, L – Largely available, F – Fully available

Element	Mapping available[4]	Restrictions	Comments
Function Schedule	L	Applications are entitled to control their scheduling policy	Provided by the OS module
FileManager	F		Provided by the persistency module
MemoryManager	N	Not needed due to the concept of service-oriented communication	
ConcurrencyManager	F		Provided by the OS module
TimeManager	F		Provided by the time synchronization module
SocketManager	F		Provided by the Network Management module and the underlying OS
LibraryManager	F		Handled by the underlying OS
ExecutionManager	L/F	Partitioning functionality only available in combination with a hypervisor	Provided by the OS module and the hypervisor
VariableManager	N	Not used by the AP due to the concept of service-oriented communication	
MessageManager	N	Not used by the AP due to the concept of service-oriented communication	
ConfigurationManager	F		Provided by the execution manager module
NetworkManager	F		Provided by the communication manager module
MonitoringManager	L	Variables can only be monitored if they are configured to provide notification	Provided by the communication manager module
IOManager	N	IO handling is not in the scope if AP.	Needs to be implemented as non-platform service

Element	Mapping available[4]	Restrictions	Comments
SynchronizationManager	F		Provided by the time synchronization module
FunctionManager	F		Provided by the execution manager and OS modules
HealthManager	F		Functionality distributed among execution manager, identity and access manager, OS and platform health management modules
LogManager	P	AP Logging does not provide logging of variables/fields but logging of arbitrary messages	Logging of variables/fields could be implemented as a non-platform service
TopologyManager	N		Needs to be implemented as a non-platform service
RedundancyManager	N		Needs to be implemented as a non-platform service
DeploymentManager	F		Provided by the execution manager module

Table 31 - Summary of FDF design instantiation based on AUTOSAR

Chapter 4 FDF design instantiation based on RTOS INTEGRITY

This design instantiation is based on a Real Time Operating System. After performing an analysis on several RTOS and Hypervisors, result of which is included in Safe4RAIL deliverable D2.2 [2], it was decided to use Integrity, a solution which has been already used in railway industry, guarantees system resources for individual processes and supports Asymmetrical Multiprocessing (AMP) and Symmetrical Multiprocessing (SMP) among other functionalities. This RTOS is the Flagship OS of Green Hills Software®. It supports a wide variety of Hardware platforms and APIs and is compliant to many different standards, such as IEC 61508 SIL3 for industrial control systems, EN 50128 SW SIL 4 in Railway control or DO-178B Level A, for Avionics software systems. In the so-called integration file, the executable image is generated, which includes the RTOS and the applications altogether.

4.1 Mapping conceptual design

For this mapping, it is shown how each of the physical and logic elements is realized in our design instantiation. This mapping will basically consist in the use of C++ Object Oriented programming language and Integrity's Integration File to generate the necessary infrastructure. According to EN 50128 (table A.15), C or C++ textual programming language is recommended to generate software code up to integrity level SIL4 [3].

4.1.1 Variable

This will be an instantiable class in C++ code. A constructor will be provided in which a set of attributes will need to be given. Among these parameters, we can find identifier, type, minimum size or maximum size.

4.1.2 Message

This will be an instantiable class in C++ code. A constructor will be provided in which a set of attributes will need to be given. Among these parameters, we can find identifier, message, set of Variables that compose it, length.

4.1.3 Shared Memory

Integrity's Memory Region will be used. A Memory Region is a contiguous portion of the addresses in an AddressSpace and it can be either Physical or Virtual. A physical MemoryRegion represents the right and ability to access a range of actual physical memory on the board whereas a virtual MemoryRegion represents the right and ability to map and access a range of addresses in a virtual AddressSpace. A MemoryRegion is defined in the Integrity Integration File as shown in Figure 22.

```
Object  22      # object number is optional
        MemoryRegion
        Name    physmemreg
                # length is optional
                # will default to DefaultMemoryRegionSize or 0x10000
        Length  0x10000
                # first is optional, will default to some
                # legal physical memory location
        First   0x207000
                # attributes can optionally go here
EndObject
```

Figure 22 - Example MemoryRegion

A name and a given size are given and then the necessary access rights must be given to the processes or AddressSpaces, which will then in the code open, read or write in the defined regions by the use of the API.

4.1.3.1 Variable Memory

This is a shared memory which will contain Variables, i.e., Variables will be mapped to this concrete storage.

4.1.3.2 Message Memory

This is a shared memory which will contain Messages, i.e., Messages will be mapped to this concrete storage.

4.1.4 Function

The functions are schedulable pieces of software that executes some logic and, thus, they must be coded in this design instantiation.

4.1.4.1 Application Function

Application functions will be implemented in any language and they will implement the logic of the application. They will be provided and instantiated by the user.

4.1.4.2 Service Function

On the other hand, service functions, i.e., those offered by the FDF, will be written in C++ language and must implement the IFunction Interface provided by the FunctionManager and be registered inside this component to be executed.

4.1.5 Process

Processes are mapped to Integrity's Address Spaces, which are defined as individual spaces of memory addresses. Their execution can be sequential or concurrent. It will be sequential if only a process or AddressSpace is defined pro Partition, which should not imply a big additional computing cost since the only context switch is that between processes.

Otherwise, i.e., if there is more than one process per Partition, this execution will be concurrent. AddressSpaces look so in the Integrity integration File.

```
AddressSpace
    Name                information
    Filename            information
    Language            C
    Object 10
       Connection       phonecompany
    EndObject
    Task Initial
       Weight           100
       StartIt          true
       StackLength      0x1000
       TimeSlice        MINIMUM
    EndTask
EndAddressSpace
```

Figure 23 - Example AddressSpace

4.1.6 Partition

The partition needs to be understood as an execution environment with an isolated memory address space and limited execution time. This means that we provide temporal and spatial separation by the use of a partition. For our design instantiation, we will be making use of Integrity's element of the same name, i.e., Integrity Partition.

These partitions need to be defined in Integrity's integration file and for each of them a set of processes, or address spaces in Integrity, need to be defined as well as an execution time within the major frame period of the partition scheduler and an execution offset or execution slice.

4.1.7 Schedule

In Integrity the whole structure of the partitions and processes and the complete set of shared resources is defined in ist Integration File. Here the Partition Schedules with their set of Partitions and the list of processes running within each of these partitions are identified, where each of the partitions have execution offsets and slices. By the use of this file a Monolitic image is built, which contains the Operating System itself and all the set of resources that have been defined in the file. Prior to this step each of the processes needs to be created and compiled in order to create an .elf extension executable file, which is then referenced in the Integration file and will run within an address space.

The execution of the partitions is always sequential in integrity. On the other hand, the execution of processes can be squential if there is only one address space per partition or concurrent otherwise. The execution of the list of functions within a process will always be sequential.

4.1.7.1 Partition Schedule

The partitions need to be configured to specify the plan, sequence and time allocation for their executions. That configuration is defined in partition schedules where, among others, the period, priority, address spaces, offset and execution times may be defined for each partition. Figure 24 shows a partition schedule example where the features mentioned before are configured for two partitions.

```
PartitionSchedule 1
        # Major Frame period is 1 second long
        MajorFramePeriod 1.0
        # Any tasks with priority 3 or less run in the background
        BackgroundMaxPriority 3
        # Release semaphore 60 at the top of the major frame
        FrameReleaseNotification 60
        # Each partition can contain multiple address spaces
        Partition One
                AddressSpace pschedv1
                AddressSpace pschedv2
                # Release semaphore 61 at the top of this window
                PartitionReleaseNotification 61
                # At 0.0 seconds into the major frame run .1 seconds
                Offset 0.0
                Exectime 0.1
                # At 0.5 seconds into the major frame run .1 seconds
                Offset 0.5
                Exectime 0.1
        EndPartition
        Partition Two
                AddressSpace pschedv2
                # At 0.1 seconds into the major frame run .4 seconds
                Offset 0.1
                Exectime 0.4
        EndPartition
EndPartitionSchedule
```

Figure 24 - Example Partition Schedule

4.1.7.2 Process Schedule

Processes belong to different process schedules shall also be configured using schedules. Figure 25 shows a process schedule example where the address space, offset and execution time for a process are specified.

```
Partition One
        AddressSpace pschedv1
        AddressSpace pschedv2
        # Release semaphore 61 at the top of this window
        PartitionReleaseNotification 61
        # At 0.0 seconds into the major frame run .1 seconds
        Offset 0.0
        Exectime 0.1
        # At 0.5 seconds into the major frame run .1 seconds
        Offset 0.5
        Exectime 0.1
EndPartition
```

Figure 25 - Example Process Schedule

4.1.7.3 Function Schedule

The execution of functions will always be sequential. The functions to be executed for every process of each partition are registered in the FunctionManager which will hold the list of Functions that it needs to execute. The FunctionManager will go through this list function by function, so the execution of functions will always be sequential.

4.2 Mapping structural design

This chapter explains how the different software components of the architecture are mapped in this concrete design instantiation. As mentioned before, the FDF components are grouped in three blocks: Functional Distribution Services (FDS), Hardware Access Services (HAS) and Operating System Services (OSS). Since the first one needs to be portable across different platforms, the FD Services block needs to be implemented and will interact with other software components through the well-defined interfaces offered by the other two blocks. Regarding the other two blocks the solution will be to create wrapper functions to access services provided by the underlying Operating System, in the case of the OSS, and the diverse set of drivers, in the case of the HAS.

4.2.1 Hardware Access Services

As mentioned, the idea is to create wrapper functions to access the underlying driver functionalities for all the software components grouped in this block.

4.2.1.1 ECUDriverManager

This component provides services to access the ECU's data. It allows getting the load and the temperature of the ECU, which may be used to detect over/under temperature and overloads.

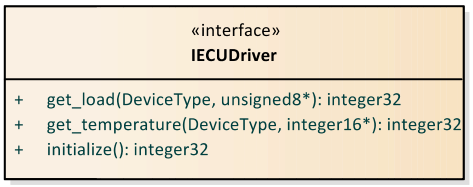

Figure 26 - ECUDriverManager.

4.2.1.2 NICDriverManager

This component will consist in a wrapper function to access the DbD driver functionalities. It will make use of two interfaces, one for every sort of traffic. On the one hand, IBEDriver will be in charge of the Best Effort traffic and with it, messages will be sent and received. On the other hand, IRTDriver is responsible for the Real-Time traffic and offers similar functions to handle messages, apart from that to get the time.

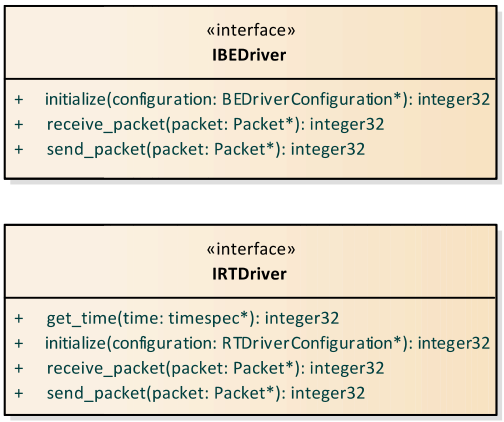

Figure 27 – NICDriverManager

4.2.1.3 IODriverManager

The FDF offers access to the IO through this component. It is a wrapper function, which loads the corresponding drivers and encapsulates their functionalities to provide a set of generic interfaces to the related components. Drivers for the most common IO devices are provided: Analog Input, Analog Output, Digital Input and Digital Output. By the use of these interfaces the value of the device is updated or retrieved.

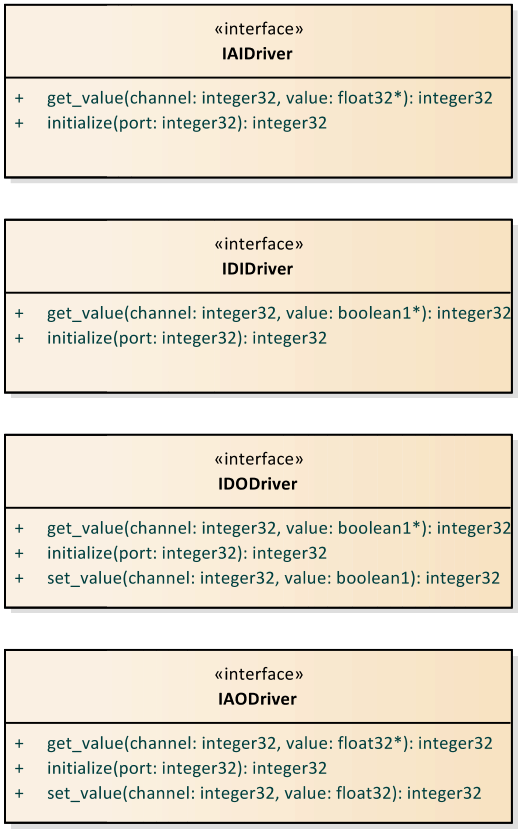

Figure 28 – IODriverManager

4.2.1.4 IWDDriverManager

This manager is responsible for handling the Watchdog. In the case of this RTOS based instantiation, PowerPC's watchdog will be used, so this wrapper function load this concrete driver's logic and provides the IWDDriver functionality so that HealthManager component can refresh the Watchdog.

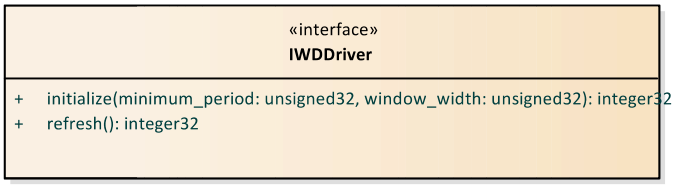

Figure 29 – WDDriverManager

4.2.2 Operating System Services

For this block of components, the interfaces are based in POSIX, so the calls are well-known. Besides, as it happens with the HAS, wrapper functions will be developed to access the underlying OS services.

4.2.2.1 MemoryManager

This Manager is responsible for taking care of accessing the shared memories safely. Its only interface provides the shm_open() function to create or open a previously created shared memory and mmap() maps a shared memory location to a given object.

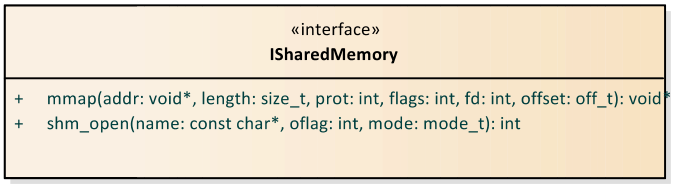

Figure 30 – MemoryManager

4.2.2.2 ExecutionManager

This is Integrity's Partition Scheduler. This partition scheduler fulfils all the functionalities that are expected from the ExecutionManager. It grants the computing resources to the selected partition and executes the process when it is meant safely, offering temporal and spatial separation. This components offers an operation to change the schedule.

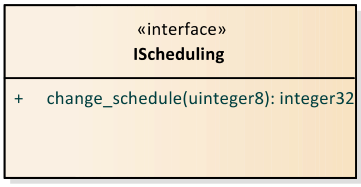

Figure 31 - ExecutionManager.

4.2.2.3 TimeManager

This manager handles the update of the system clock when the global clock comes from the network and provides timers for general use. It offers two interfaces: with IClock the system clock and the resolution can be consulted and the system clock set. On the other hand, with ITimer we create and manage timers.

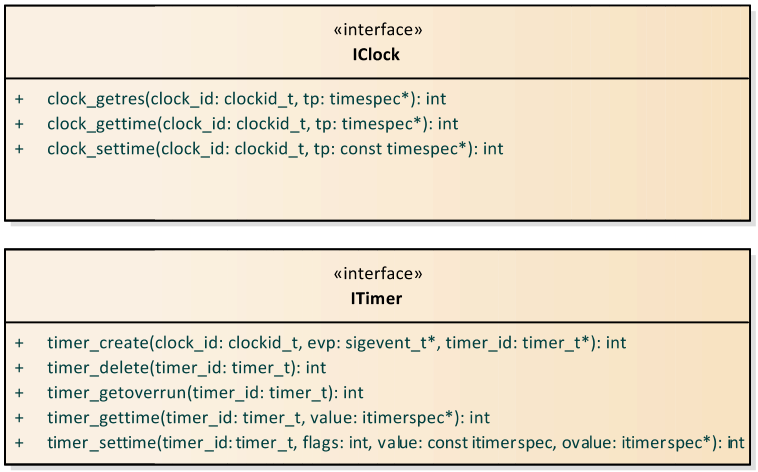

Figure 32 – TimeManager

4.2.2.4 ConcurrencyManager

With ConcurrencyManager semaphores and mutexes can be managed. This manager offers two different interfaces for this task: ISemaphore and IMutex.

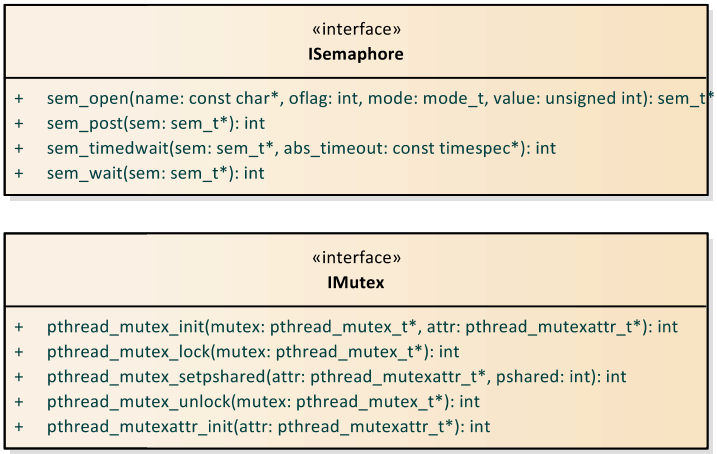

Figure 33 – ConcurrencyManager

4.2.2.5 SocketManager

This manager offers all regular socket functionalities. By the use of its unique interface sockets can be bond, connected or listened, for instance.

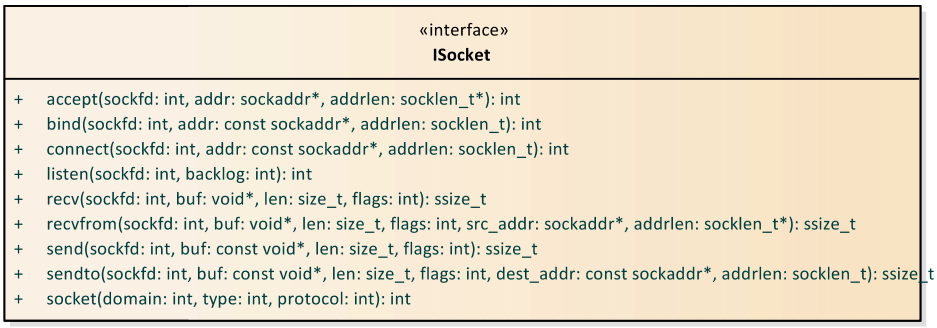

Figure 34 – SocketManager

4.2.2.6 FileManager

IFile interface of this manager offers typical functions to access the file system, i.e., open, read, write and close, and remove paths. It is the only OS Manager with Standard C based functions in its interface.

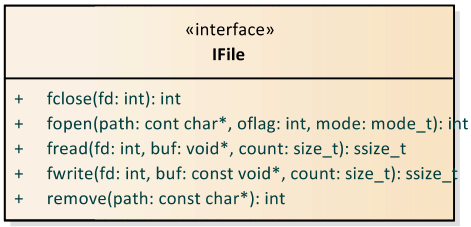

Figure 35 – FileManager

4.2.2.7 LibraryManager

ILibraryManager provides services to handle dynamic libraries (DLL), including open and close libraries and handle errors.

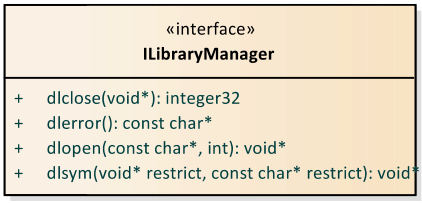

Figure 36 - LibraryManager.

4.3 Summary and Conclusion of Integrity-based Instantiation

In Table 32, the summary of the FDF design instantiation report based on INTEGRITY is provided.

Element	Mapping available[5]	Restrictions	Comments
Variable	L	-	It is an instantiable Class. In the Constructor parameters such as type and identifier given.
Message	L	-	It is an instantiable Class. In the Constructor parameters such as identifier and set of variables that compose it are given.
Variable Memory	L	-	Shared Memories are Integrity's Memory Region. Need to define data structure of variables
Message Memory	L		Shared Memories are Integrity's Memory Region. Need to define data structure of messages.
Application Function	F	-	It will be C++ code. Must implement IFunction Interface in order to be executed.
Service Function	F	-	It will be C++ code. Must implement IFunction Interface in order to be executed.
Process	F	-	It is a .elf extension binary file which is attached to an Integrity's address space.
Partition	F	-	Integrity's Partition.
Partition Schedule	F	-	Integrity's Integration File. More than one ParitionSchedule can be present. Please refer to chapter 4.1.7 for further information.
Process Schedule	F		The set of processes running within a partition are provided in the Integrity Integration File.
Function Schedule	F		This is the list of functions running within an .elf binary file.
FileManager	P	-	Standard C library provided by the RTOS. A wrapper needs to be implemented to get to SIL4.

[5] N – Not available, P – Partially available, L – Largely available, F – Fully available

Element	Mapping available[5]	Restrictions	Comments
MemoryManager	P	-	POSIX API provided by the RTOS. A wrapper needs to be implemented to get to SIL4.
ConcurrencyManager	P	-	POSIX API provided by the RTOS. A wrapper needs to be implemented to get to SIL4.
TimeManager	P	-	POSIX API provided by the RTOS. A wrapper needs to be implemented to get to SIL4.
SocketManager	P	-	POSIX API provided by the RTOS. A wrapper needs to be implemented to get to SIL4.
LibraryManager	P	-	POSIX API provided by the RTOS. A wrapper needs to be implemented to get to SIL4.
ExecutionManager	F	-	Integrity's Enhanced Partition Scheduler. Provides all functionalities expected from the execution manager. It does not offer any interface, since it is not called by any of the other components.

Table 32 – Summary of FDF design instantiation based on INTEGRITY

Chapter 5 FDF design instantiation based on Hypervisor PikeOS

In order to map the FDF design into the PikeOS paradigm, the basic architecture of PikeOS is shown in Figure 37.

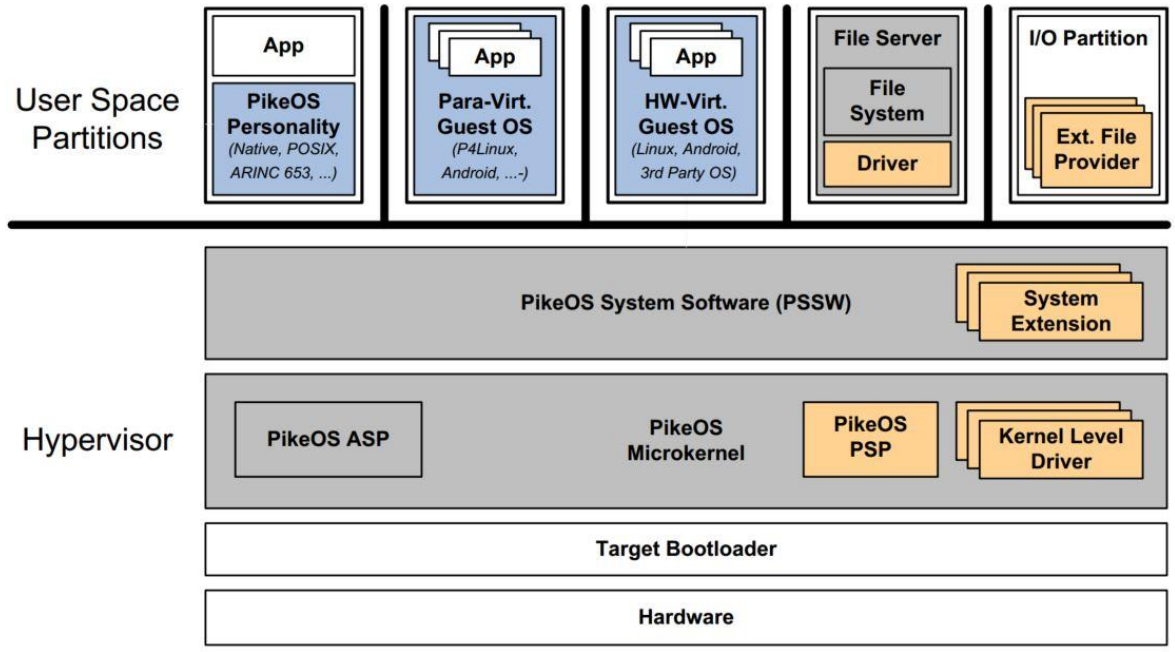

Figure 37 - PikeOS System Architecture

In the PikeOS architecture, the microkernel consists of the Architecture Support Packet (ASP) that depends on the CPU architecture and the platform dependent part called Platform Support Package (PSP). The microkernel is linked with the Kernel Level Drivers to build up the kernel, which runs with supervisor privileges.

The PikeOS System Software (PSSW) contains majorly the built in system extensions and is also responsible for reading the configuration file to initialize the partitions, channels between partitions and health monitoring tables.

The hosted user partitions are resource partitions according to the PikeOS paradigm. Normal applications and system services could be implemented as partitions which make usage of the services provided by either the PSSW or the PikeOS kernel.

PikeOS implements para-virtualization and the microkernel serves as a Virtual Machine Monitor (VMM) to trap attempts of the user applications to execute privileged instructions or to access system resources not assigned to them.

5.1 Mapping conceptual design

In this section, we will analyse the existing FDF conceptual design and try to map each component to the services provided by PikeOS.

5.1.1 Variable

The variable is the defined data structure to share information between processes. The configuration of a variable includes identifier, type, range, default value and deadline. The fields of a variable are defined to contain the value, timestamp and validity flag.

Within PikeOS paradigm, there exists the so-called “Inter Process Communication (IPC)” mechanism which is the primary thread communication mechanism. IPC is designed to copy or map a data from a sending thread’s address space to a receiving thread’s address space. However, IPC is a synchronous and unbuffered method with support of a timeout parameter that specifies the maximum time a sending thread can wait. Through IPC there can be only one receiver and multicast or broadcast are not available. In this case, IPC is not capable to achieve the functionality of the defined variable.

Another mechanism defined in PikeOS for inter process communication is the Event, which is the only asynchronous thread communication in PikeOS as well as the fastest communication service within PikeOS kernel. However, for a receiving thread, it cannot determine which thread has signalled an event, when multiple threads are enabled to signal an event. The defined event in PikeOS maps not well for the variable.

The existing defined inter process communication mechanisms in PikeOS cannot be directly used as the variable in the FDF, a data structure in C/C++ language needs to be defined based on the necessary fields of a variable.

5.1.2 Message

The message is the defined data structure to share variables between partitions residing on different computing nodes. The configuration of a message should contain the identifier of the message, the identifier of the contained variables, indicator of communication direction (e.g., to be send or received), time point of sending or receiving and deadline.

Basically, PikeOS provides the message based communication through static channels which link two ports, i.e. source and destination port. A port can be a sampling port or a queuing port. The system integrator can configure the refresh rate of a message residing in a sampling port, in order to guarantee the validity of a message. Since the defined channel is statically configured before booting the system, it is not directly capable to deal with the train inauguration in the TCMS FDF context. The dynamic behaviour of train inauguration needs to be instantiated on a higher level.

5.1.3 Shared Memory

The shared memory is defined as the memory space that can be simultaneously accessed by processes. And the accessing right of different processes should be configurable.

From the viewpoint of PikeOS, it also provides shared memory objects which can be accessed by different partitions within the same computing node. The shared memory objects are created at boot time and accessible during run time. A shared memory object is treated as a file in the file system name space of a file provider called SHM, which enables a

shared memory object to be accessible through the PikeOS file system API. The accessing permissions of a shared memory object can be configured in the partition's file access list.

5.1.3.1 Variable Memory

As aforementioned, a shared memory object needs to be compatible to store the defined variables and access the variable with the variable identifier.

5.1.3.2 Message Memory

As aforementioned, a shared memory object needs to be compatible to store the defined messages and access a message with the variable identifier.

5.1.4 Function

Function is defined in the FDF concept as the schedulable software unit which processes data in variables or messages. A function can be categorised into application function and service function that are provided and instantiated by the users or the FDF.

In PikeOS system, a thread is the schedulable entity, which is identified by a thread ID. A thread ID is unique within a process/task. Addressing threads residing in different tasks will need to combine the task ID and the thread ID, in order to generate a system wide unique ID.

5.1.4.1 Application Function

Since an application function implements the user defined logic, the application functions could be implemented within a normal resource partition in the PikeOS paradigm. A resource partition defines the system resources that the functions within it can exclusively use and functions residing in other resource partitions cannot access them.

5.1.4.2 Service Function

The service functions are designed to provide the FDF services for the application functions. In the design of PikeOS system, it uses the time partitioning mechanism to allocate CPU time among resource partitions in a fixed cyclic way. Moreover, PikeOS also introduces a background time partition, which is active during runtime and contains threads of both high priority (e.g., error handler) and low priority. This background time partition is designed for the safety critical threads to pre-empt other threads and for the non-safety critical threads to consume the idle CPU time of all the time partitions. Based on this design, the service functions could be developed and assigned to the background time partition.

5.1.5 Process

A process in the FDF concept consists of threads and owns isolated address space to other processes. A process needs to be configured with the scheduling plans of the functions during runtime. For accessing a shared memory, only one dedicated process is allowed to have write access, and other processes could only have read access.

From the PikeOS point of view, more than one PikeOS task can be started within one resource partition and each task has its own virtual address space and memory. All the tasks within one resource partition also share the system resources which belong to this resource partition and not assigned to specific tasks. A task is configured with the Maximum

Controlled Priority (MCP) which is used for the PikeOS priority-based scheduling. The access control of a shared memory in PikeOS is in the granularity of resource partition instead of process/task. It would be necessary to implement the access control in the shared memory manager.

5.1.6 Partition

The partition is defined to be the process execution environment with temporal and spatial isolation to other partitions. A process in the FDF can belong to different partitions and the scheduling plan of the processes in one partition is configurable.

In the PikeOS paradigm, the above defined partition is a combination of time partition and resource partition in a PikeOS system. The resource partition in PikeOS defines the system resources that the processes within it can exclusively use and processes residing in other resource partitions cannot access them. This concept contributes to the defined spatial isolation between partitions. In another case, time partitioning in PikeOS defines the mechanism to allocate CPU time amongst resource partitions. The mapping between resource partitions, time partitions and CPU time windows is shown in Figure 38. It is also possible that different threads in the same resource partition belong to different time partitions. A task/process in PikeOS is able to change a thread's time partition during runtime. When a process/task needs to belong to different partitions, i.e. the threads of a process/task belong to different partitions, the synchronization mechanisms between threads provided by PikeOS like mutexes, conditional variables, semaphores, etc. need to be used for the thread synchronization.

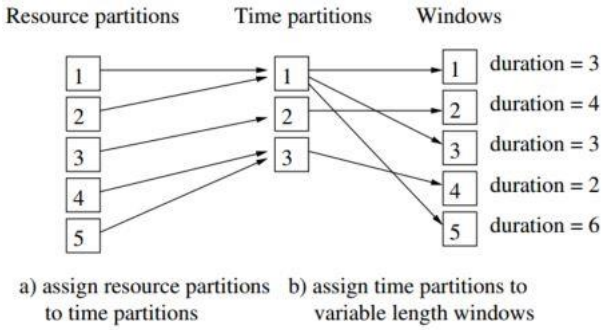

Figure 38 - Mapping between partitions and CPU time windows

5.1.7 Schedule

5.1.7.1 Partition Schedule

In the FDF concept design, a partition schedule represents for a scheduling plan of partitions. And a partition can also belong to different partition schedule. During runtime there is only one active partition schedule and the other partition schedules are standby to be loaded.

During the integration of a PikeOS project, the system integrator can define multiple scheduling schemes and switching between pre-defined time partitioning schemes is feasible during runtime.

5.1.7.2 Process Schedule

The process schedule is defined as the scheduling plan of the processes and one process can belong to different process schedules. However, in a PikeOS system, it is possible to indirectly schedule the processes through defining the priority of a process, because the scheduler of PikeOS dispatches threads based on their priorities. Switching between different process schedules is impossible due to the unchangeable process priorities.

5.1.7.3 Function Schedule

The function schedule is similar to the process above and the implementation in PikeOS system is different, because the priority and time partition of a thread can be changed during runtime. Changing the thread priority can indirectly changing the schedule of threads. But there is no defined PikeOS APIs to switch directly between pre-defined thread scheduling plans. One thread scheduling plan could be mapped to one setup of the threads' priorities that makes it possible to define different threads' priority setups and switch between them.

5.2 Mapping structural design

The FDF structural design consists majorly of the grouped services in the FDF architecture. In this section, we will analyse the ones related to the services provided by PikeOS and provide the detail report on how to implement these components based on the PikeOS services.

5.2.1 Hardware Access Services

The hardware access services depend on the target hardware. For specific hardware like I/O devices, NIC card and watchdog devices, the implementation of the hardware access services could be achieved through Kernel Level Device Drivers, drivers in PSSW layer or as a system partition. The pros and cons of these options are discussed as follows.

In terms of configuration, the system extensions and user partition can access the PikeOS property file system to retrieve their configuration data, while a kernel driver could only be configured by binary configuration which is compiled from dedicated XML files. Accessing the property file system is flexible with the cost of increased system overhead. For the operational requests (e.g., read, write…), the drivers implemented as a user partition or a system extension involve address space switches from the requesting applications to the drivers that results in the longer turn-around times comparing to the kernel drivers which require no address or thread switch according to the PikeOS thread model. From the view point of address space separation, a driver implemented as a user partition only needs to at the highest criticality of the other partitions that access the drivers. For a system extension or a kernel driver, they always need to have the highest criticality level in the whole system. The interrupts occurring in the system are routed to a user partition or a system extension by blocking that requires address space and thread switching, while a kernel driver provides pre-defined call backs for interrupts with limited operation like data transfer. As aforementioned, address space switch and thread switch come with extra system overhead. For accessing to system resources except interrupts, a user partition or system extension has more restrictions than a kernel driver (e.g., accessing high resolution timers). Based on the PikeOS thread model, blocking is easier to be achieved in a user partition or a kernel driver than in a system extension due to the non-blocking property of a partition daemon in the PSSW layer. The last point is that a user partition and system extension can start new

worker threads when necessary, while a kernel driver runs in the thread context of the calling application.

5.2.2 Operating System Services

5.2.2.1 FileManager

The PikeOS file system provides a uniform way to access various types of files and the PSSW module contains a file system layer to dispatch calls to different file providers. Moreover, PikeOS also provides volume providers to have complex hierarchical file systems that are capable for dynamic creating of files and directories. Since the defined FileManager provides interfaces to read and write files, it is necessary to define wrapper functions to convert these interfaces into the PikeOS file system APIs.

The major file system APIs are as follows.

- vm_open () / vm_open_at (): to open a file
- vm_read () / vm_read_at (): to read from a file
- vm_write () / vm_write_at (): to write into a file
- vm_close (): to close a file
- vm_map () / vm_map_to (): to map a file into the caller's address space
- vm_ioctl (): file provider specific control function

These are the major APIs to be used in the implementation. The other functions link file controlling are not listed here.

5.2.2.2 MemoryManager

Since the shared memory objects in PikeOS is treated as a file in the file system name space of a file provider called SHM, which enables a shared memory object to be accessible through the PikeOS file system API. For reading and writing a variable or a message in a file, an extra data structure needs to be defined, in order to record the offset of a variable or message from the beginning of the shared memory. The reason is that the file system in PikeOS does not provide any index functions.

For memory management, PikeOS provides the following APIs for the applications:

- vm_mem_lookup (): to retrieve the memory descriptor when given a name
- vm_mem_pool_alloc (): the caller can allocate memory from a partition's memory pool
- vm_mem_stat (): used to retrieve information about a memory object's status

5.2.2.3 ConcurrencyManager

In the PikeOS system, the synchronization between threads can be achieved by the provided services like mutexes, semaphores, etc. In order to implement the ConcurrencyManager, it is necessary to develop the necessary wrapper functions using the provided PikeOS kernel APIs. Another point is to implement a store to keep the mapping between mutexes/semaphores with the owner threads.

For mutexes, the provided APIs of the PikeOS is listed as followed.

- p4_mutex_init (): to initialize a mutex

- p4_mutex_owned (): to test the mutex lock owner
- p4_mutex_lock () / p4_mutex_trylock (): used to lock a mutex
- p4_mutex_unlock (): to unlock a mutex

Similarly, the APIs for semaphores are as following:

- p4_sem_init (): to initialize a semaphore
- p4_sem_value (): to get the counter value of a semaphore
- p4_sem_ wait () / p4_sem_ try_wait (): to lock a semaphore
- p4_sem_ post (): to unlock a semaphore

5.2.2.4 TimeManager

The defined TimeManager needs to be able to get and set the time of the system clock. However, PikeOS kernel provides a logical time base for system time and timeouts. The system time value records the elapsed time since the system starts up. PikeOS provides only the API to get the system clock.

The APIs regarding timeouts in PikeOS kernel are as following:

- p4_sleep (): for a thread to delay for an amount of time
- p4_get_time_syscall () / p4_get_time (): to get the system time since boot in nanoseconds
- p4_get_ts_syscall () / p4_get_ts (): to read the CPU time stamp counter

5.2.2.5 ExecutionManager

The defined ExecutionManager needs to guarantee isolation between partitions and execute partitions and processes according to the corresponding partition/process schedule.

In PikeOS, it is feasible to define the scheduling plans of partitions for the scheduler residing in the PikeOS kernel. In this case, it is possible to control the kernel to switch between pre-defined schedule schemes of partitions. For the process execution, there is no possibility to directly define the process schedules for the PikeOS kernel scheduler. But one feasible way is to define the plan for changing the threads' priority which leads to the changing of process execution, because within a partition, the threads are priority based scheduled.

For switching between different time partition scheduling plans, the PSSW provides the following APIs:

- vm_tsched_lookup (): to lookup the scheduling plan id by its name
- vm_tsched_stat (): to retrieve the time partition scheduling status of a CPU
- vm_tsched_change (): to change the time partition plan to another specified one

In order to change the priority of a thread, the PikeOS kernel provides the following APIs:

- p4_thread_get_sched (): to retrieve the thread priority and time partition
- p4_thread_set_sched (): to set the thread priority and time partition
- p4_thread_set_priority (): to set the priority of a thread
- p4_thread_get_priority (): to retrieve the priority of a thread

5.2.2.6 SocketManager

In the PikeOS system, the PikeOS native personality supports the ANIS/CIP that is a certifiable UDP/IP stack. In case that TCP/IP stack is necessary, the LwIP that comes with the PikeOS POSIX personality could also be used.

5.2.2.7 LibraryManager

The LibraryManager provides the services to handle operations (e.g., open, close, etc.) on dynamic libraries. However, in the native PikeOS system, there is no pre-defined system services of dynamic libraries. Since PikeOS provides POSIX personality and dynamic linking is provided as one of the standard POSIX header files, it is feasible to implement the LibraryManager leveraging the POSIX personality.

5.3 Summary and Conclusion of Hypervisor-based Instantiation

In Table 33, the summary of the FDF design instantiation report based on PikeOS is provided. Since the details regarding the Hardware Access Services are tightly related to the hardware platform, the report of driver instantiation options is provided in section 5.2.1.

Element	Mapping available[6]	Restrictions	Comments
Variable	P	Unbuffered synchronized unicast	IPC and Event need to have higher level logic, better use shared memory
Message	P	Static port based channel	Better to use shared memory and implement memory manager
Variable Memory	L	-	Need to define data structure of a variable and implement the manager
Message Memory	L	-	Need to define data structure of a message and implement the manager
Application Function	F	-	Thread priorities affect the scheduling
Service Function	L	-	Either implemented in a service partition or defined as libraries for the applications
Process	L	Shared memory access control in the granularity of partition	Need to control the shared memory access at the process level

[6] N – Not available, P – Partially available, L – Largely available, F – Fully available

Element	Mapping available[6]	Restrictions	Comments
Partition	F	-	Combination of time partition and resource partition
Partition Schedule	L	Only possible to switch between pre-defined scheduling scheme	-
Process Schedule	L	Impossible to switch between schedule plans	Changing the process priority can implicitly change the schedule plan
Function Schedule	L	No direct way to change the thread schedule	Modifying a thread's priority is possible during runtime
FileManager	L	Extra file index needs to be maintained	Either use native APIs or POSIX APIs
MemoryManager	L	Extra index needs to be maintained	Either use native APIs or POSIX APIs
ConcurrencyManager	L	Mapping between synchronization objects and threads need to be maintained	Either use native APIs or POSIX APIs
TimeManager	P	No API for direct setting the system clock	Time partitions need to be synchronized
ExecutionManager	L	-	The resource partition defines the spatial isolation.
SocketManager	L	PikeOS native personality only support UDP/IP stack	PikeOS POSIX personality supports UDP/IP and TCP/IP
LibraryManager	P	PikeOS native personality does not support dynamic linking	Implementation based on PikeOS POSIX personality

Table 33 – Summary of FDF design instantiation based on PikeOS

Chapter 6 Summary and conclusion

The present document evaluates three different design instantiations of the Functional Distribution Framework for next generation TCMSs. Those different approaches cover a variety of initial situations:

Coming from the application side, the PikeOS hypervisor together with PikeOS native API mainly provide an interface to a virtual machine. Thus, implementing the FDF on top of this solution will probably requires the most implementation work but provide the highest level of freedom and consequently enables implementers to stay as close as possible to the initial design and requirements.

INTEGRITY, on the other hand, is a POSIX compliant Real-time operating system built with partitioning support and aiming at applications that require high RAMS[7] requirements. In terms of implementation effort for the FDF instantiation, this solution is probably less complex than the PikeOS approach, as multiple FDF service functionalities are already covered by the INTEGRITY RTOS and need to be wrapped.

However, it should be mentioned that both approaches would require the implementation of the Functional Distribution Services on top of Hardware and OS Abstraction layers. By using defined APIs for the latter portability between different hardware platforms is enabled.

The AUTOSAR Adaptive platform in contrast provides its own set of services and APIs to applications. The result of the analysis is that AP modules cover most of the FDF's functionality, although there are some conceptual differences and missing features. One major issue is the fact that AP does not provide a partitioning concept. This however can be solved by combining the hypervisor approach (e.g. by using PikeOS Hypervisor) with AP in such a manner that multiple AP instances run in different PikeOS partitions and thus guarantee freedom of interference. It is to be evaluated in subsequent work, if and how the AP application API can be wrapped to be able to run FDF applications.

In conclusion, all three approaches seem to be viable to provide a basis for next generation TCMSs. PikeOS and INTEGRITY are proven and tested in lots of ECUs in various safety-related applications. AP on the other hand is still in development and aims specifically at the automotive domain, omitting some railway-relevant features and leaving open questions, when it comes to certification as both domains take different approaches and railway applications usually require a higher safety level.

During subsequent work in Safe4RAIL, proof-of-concept implementations based on the three platforms analysed here will be performed. Corresponding reports will provide further details, while the outcome of D2.6[8] will contain detailed comparison of the different approaches.

[7] Reliability, Availability, Maintainability, Safety

[8] This deliverable is confidential

Chapter 7 List of Abbreviations

AA	Adaptive Application
AMP	Asymmetrical Multiprocessing
AP	Adaptive Platform
API	Application Program Interface
ARA	AUTOSAR Runtime for Adaptive Applications
ASP	Architecture Support Package
AUTOSAR	Automotive Open System Architecture
CP	Classic Platform
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DbD	Drive-by-Data
ECU	Electronic Control Unit
FDF	Functional Distribution Framework
FDS	Functional Distribution Services
HAS	Hardware Access Services
IO	Input Output
IOC	IO Controller
IP	Internet Protocol
IPC	Inter Process Communication
NIC	Network Interface Controller
OEM	Original Equipment Manufacture
OS	Operating System
OSS	Operating System Services
POSIX	Portable Operating System Interface for UNIX

PSP	Platform Support Package
PSSW	PikeOS System Software
RTOS	Real Time Operating System
Safe4RAIL	Safe architecture for Robust distributed Application Integration in roLling stock
SIL	Safety Integrity Level
SMP	Symmetrical Multiprocessing
SoC	Service-oriented Communication
SOME/IP	Scalable service-Oriented MiddlewarE over IP
TCMS	Train Control and Monitoring System
TTDB	Train Topology Database
WDG	Watch Dog

Table 34: List of Abbreviations

Chapter 8 Bibliography

[1] „D2.2 Report on analysis of ‘functional distribution architecture’ frameworks and solutions,“ Safe4RAIL project, 2016.

[2] „D2.3 Report on ‘TCMS framework concept’ design, security concepts, and assessment,“ Safe4RAIL prokect, 2016.