Document Title	Explanation of Adaptive Platform Design
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	706

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R23-11

Document Change History

Date	Release	Changed by	Description
2023-11-23	R23-11	AUTOSAR Release Management	• An update of the logical architectural view and introduction of protocol, safety, and security features illustrations in the logical view. • An overall update according to the latest specification change in the State Management chapter • Removal of obsolete 'Recovery Action' and 'Deterministic Client' description in the Execution Management chapter • An update of the architecture diagram in the Log and Trace chapter • An update of the architecture diagram in the Log and Trace chapter • Minor updates in the Architecture, Platform Health Management, Update and Configuration Management, Persistency, Identity and Access Management, Operating System Interface and Core Types chapters • Introduction of the Raw Data Stream chapter, and its removal from the Communication Management chapter.

△

2022-11-24	R22-11	AUTOSAR Release Management	• A new chapter Firewall • Introduction of State Machine in the State Management chapter • Minor updates in Platform Health Management, E2E-Protection, Persistency, Update and Configuration Management and Diagnostics chapters
2021-11-25	R21-11	AUTOSAR Release Management	• Removal of the REST chapter • Introduction of a IDSM (Intrusion Detection System Management) chapter • Introduction of SHM (System Health Management) in the PHM chapter • Refreshed contents in the Persistency chapter • Refreshed UCM and SM contents with regard to their interactions • Minor updates in the Exection Management, Diagnostics, Time Sychronization
2020-11-30	R20-11	AUTOSAR Release Management	• Moderate amount of changes in the State Management, Update and Configuration Management, Cryptography, and Safety • Minor changes in Execution Management, Diagnostics, Persistency, Identity and Access Management • Minor changes in the architecture logical view

▽

△

2019-11-28	R19-11	AUTOSAR Release Management	• Updated the architecture logical view • Updates in Execution Management, Communication Management, Security, Diagnostics, Persistency, State Management, Network Management, Update and Configuration Management, Platform Health Management, Core Types chapters updated due to changes in SWS • Various minor updates for clarification • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Changes to reflect the latest SWS contents • Chapter 17.4 C++ coding guidelines deleted
2018-10-31	18-10	AUTOSAR Release Management	• Changes to reflect the latest SWS contents
2018-03-29	18-03	AUTOSAR Release Management	• Update of a logical view of AP architecture • Addition of Update and Configuration Management, State Management, Time Synchronization, Adaptive Network Management, Identity Access Management, Cryptography, and Core types
2017-10-27	17-10	AUTOSAR Release Management	• Added RESTful Communication
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Contents

1 Introduction

1.1 Contents

This specification describes the AUTOSAR Adaptive Platform (AP) design. The purpose of this document is to provide an overview of AP but is not to detail all the elements of AP design. It is to provide the overall design of the AP and key concepts for both AP users and AP implementers.

The document is organized as follows. It starts with Technical Scope and Approach to provide some background of AP, followed by Architecture describing both logical and physical views of AP. Independent chapters of Methodology and Manifest and all Functional Clusters follow, which are the units of functionalities of AP, each containing its overview and introductions to their key concepts.

The detailed specification and discussions on the explained concepts are defined in the relevant RS, SWS, TR and EXP documents.

1.2 Prereads

This document is one of the high-level conceptual documents of AUTOSAR. Useful pre-reads are [1], [2], and [3].

1.3 Relationship to other AUTOSAR specifications

The detailed specification and discussions on the explained concepts are defined in the relevant RS, SWS, TR and EXP documents. A detailed technical architecture documentation of the AUTOSAR Adaptive Platform is provided in [4].

2 Related Documentation

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] Main Requirements
AUTOSAR_FO_RS_Main

[3] Methodology for Adaptive Platform
AUTOSAR_AP_TR_Methodology

[4] Explanation of Adaptive Platform Software Architecture
AUTOSAR_AP_EXP_SWArchitecture

[5] The 4+1 View Model of Architecture

[6] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[7] Specification of Execution Management
AUTOSAR_AP_SWS_ExecutionManagement

[8] Specification of Platform Types for Adaptive Platform
AUTOSAR_AP_SWS_PlatformTypes

[9] Specification of Synchronized Time-Base Manager
AUTOSAR_CP_SWS_SynchronizedTimeBaseManager

[10] Specification of State Management
AUTOSAR_AP_SWS_StateManagement

[11] Explanation of Identity and Access Management
AUTOSAR_AP_EXP_IdentityAndAccessManagement

[12] Explanation of Safety Overview
AUTOSAR_FO_EXP_SafetyOverview

[13] Safety Requirements for AUTOSAR Adaptive Platform and AUTOSAR Classic Platform
AUTOSAR_FO_RS_Safety

[14] ISO 26262:2018 (all parts) – Road vehicles – Functional Safety
https://www.iso.org

[15] E2E Protocol Specification
AUTOSAR_FO_PRS_E2EProtocol

[16] Specification of Communication Management
AUTOSAR_AP_SWS_CommunicationManagement

[17] Requirements on Health Monitoring
AUTOSAR_FO_RS_HealthMonitoring

[18] Specification of Health Monitoring
AUTOSAR_FO_ASWS_HealthMonitoring

[19] Requirements on Platform Health Management
AUTOSAR_AP_RS_PlatformHealthManagement

[20] Specification of Platform Health Management
AUTOSAR_AP_SWS_PlatformHealthManagement

3 Technical Scope and Approach

3.1 Overview - a landscape of intelligent ECUs

Traditionally ECUs mainly implement functionality that replaces or augments electro-mechanical systems. Software in those deeply-embedded ECUs controls electrical output signals based on input signals and information from other ECUs connected to the vehicle network. Much of the control software is designed and implemented for the target vehicle and does not change significantly during vehicle lifetime.

New vehicle functions, such as highly automated driving, will introduce highly complex and computing resource demanding software into the vehicles and must fulfill strict integrity and security requirements. Such software realizes functions, such as environment perception and behavior planning, and integrates the vehicle into external back-end and infrastructure systems. The software in the vehicle needs to be updated during the lifecycle of the vehicle, due to evolving external systems or improved functionality.

The AUTOSAR Classic Platform (CP) standard addresses the needs of deeply-embedded ECUs, while the needs of ECUs described above cannot be fulfilled. Therefore, AUTOSAR specifies a second software platform, the AUTOSAR Adaptive Platform (AP). AP provides mainly high-performance computing and communication mechanisms and offers flexible software configuration, e.g. to support software update over-the-air. Features specifically defined for the CP, such as access to electrical signals and automotive specific bus systems, can be integrated into the AP but is not in the focus of standardization.

3.2 Technology Drivers

There are two major groups of technology drivers behind. One is Ethernet, and the other is processors.

The ever-increasing bandwidth requirement of the on-vehicle network has led to the introduction of Ethernet, that offers higher bandwidth and with switched networks, enabling the more efficient transfer of long messages, point-to-point communications, among others, compared to the legacy in-vehicle communication technologies such as CAN. The CP, although it supports Ethernet, is primarily designed for the legacy communication technologies, and it has been optimized for such, and it is difficult to fully utilize and benefit from the capability of Ethernet-based communications.

Similarly, performance requirements for processors have grown tremendously in recent years as vehicles are becoming even more intelligent. Multicore processors are already in use with CP, but the needs for the processing power calls for more than multicore. Manycore processors with tens to hundreds of cores, GPGPU (General Purpose use of GPU), FPGA, and dedicated accelerators are emerging, as these offer orders of magnitudes higher performance than the conventional MCUs. The increasing number of cores overwhelms the design of CP, which was originally designed for a single core

MCU, though it can support multicore. Also, as the computing power swells, the power efficiency is already becoming an issue even in data centers, and it is in fact much more significant for these intelligent ECUs. From semiconductor and processor technologies point of view, constrained by Pollack's Rule, it is physically not possible to increase the processor frequency endlessly and the only way to scale the performance is to employ multiple (and many) cores and execute in parallel. Also, it is known that the best performance-per-watt is achieved by a mix of different computing resources like manycore, co-processors, GPU, FPGA, and accelerators. This is called heterogeneous computing - which is now being exploited in HPC (High-Performance Computing) - certainly overwhelms the scope of CP by far.

It is also worthwhile to mention that there is a combined effect of both processors and faster communications. As more processing elements are being combined in a single chip like manycore processors, the communication between these processing elements is becoming orders of magnitude faster and efficient than legacy inter-ECU communications. This has been made possible by the new type of processor interconnect technologies such as Network-on-Chip (NoC). Such combined effects of more processing power and faster communication within a chip also prompts the need for a new platform that can scale over ever-increasing system requirements.

3.3 Adaptive Platform - Characteristics

The characteristic of AP is shaped by the factors outlined in sections 3.1 and 3.2. The landscape inevitably demands significantly more computing power, and the technologies trend provides a baseline of fulfilling such needs. However, the HPC in the space of safety-related domain while power and cost efficiencies also matter, is by itself imposes various new technical challenges.

To tackle them, AP employs various proven technologies traditionally not fully exploited by ECUs, while allowing maximum freedom in the AP implementation to leverage the innovative technologies.

3.3.1 C++

From top-down, the applications can be programmed in C++. It is now the language of choice for the development of new algorithms and application software in performance critical complex applications in the software industry and in academics. This should bring faster adaptations of novel algorithms and improve application development productivity if properly employed.

3.3.2 SOA

To support the complex applications, while allowing maximum flexibility and scalability in processing distribution and compute resource allocations, AP follows service-oriented-architecture (SOA). The SOA is based on the concept that a system consists of a set of services, in which one may use another in turn, and applications that use one or more of the services depending on its needs. Often SOA exhibits system-of-system characteristics, which AP also has. A service, for instance, may reside on a local ECU that an application also runs, or it can be on a remote ECU, which is also running another instance of AP. The application code is the same in both cases - the communication infrastructure will take care of the difference providing transparent communication. Another way to look at this architecture is that of distributed computing, communicating over some form of message passing. At large, all these represent the same concept. This message passing, communication-based architecture can also benefit from the rise of fast and high-bandwidth communication such as Ethernet.

3.3.3 Parallel processing

Distributed computing is inherently parallel. The SOA, as different applications use a different set of services, shares this characteristic. The advancement or manycore processors and heterogeneous computing that offer parallel processing capability offer technological opportunities to harness the computing power to match the inherent parallelism. Thus, the AP possesses the architectural capability to scale its functionality and performance as the manycore-heterogeneous computing technologies advance. Indeed, the hardware and platform interface specification are only parts of the equation, and advancements in OS/hypervisor technologies and development tools such as automatic parallelization tools are also critical, which are to be fulfilled by AP provider and the industry/academic eco-system. The AP aims to accommodate such technologies as well.

3.3.4 Leveraging existing standard

There is no point in re-inventing the wheels, especially when it comes to specifications, not implementations. As with already described in section 3.3.1, AP takes the strategy of reusing and adapting the existing open standards, to facilitate the faster development of the AP itself and benefiting from the eco-systems of existing standards. It is, therefore, a critical focus in developing the AP specification not to casually introduce a new replacement functionality that an existing standard already offers. For instance, this means no new interfaces are casually introduced just because an existing standard provides the functionality required but the interface is superficially not easy to understand.

3.3.5 Safety and security

The systems that AP targets often require some level of safety and security, possibly at its highest level. The introduction of new concepts and technologies should not undermine such requirements although it is not trivial to achieve. To cope with the challenge, AP combines architectural, functional, and procedural approaches. The architecture is based on distributed computing based on SOA, which inherently makes each component more independent and free of unintended interferences, dedicated functionalities to assist achieving safety and security, and guidelines such as C++ coding guideline, which facilitates the safe and secure usage of complex language like C++, for example.

3.3.6 Planned dynamics

The AP supports the incremental deployment of applications, where resources and communications are managed dynamically to reduce the effort for software development and integration, enabling short iteration cycles. Incremental deployment also supports explorative software development phases.

For product delivery, AP allows the system integrator to carefully limit dynamic behavior to reduce the risk of unwanted or adverse effects allowing safety qualification. Dynamic behavior of an application will be limited by constraints stated in the Execution manifest (see section 4.6). The interplay of the manifests of several applications may cause that already at design time. Nevertheless, at execution time dynamic allocation of resources and communication paths are only possible in defined ways, within configured ranges, for example.

Implementations of an AP may further remove dynamic capabilities from the software configuration for production use. Examples of planned dynamics might be:

- Pre-determination of the service discovery process
- Restriction of dynamic memory allocation to the startup phase only
- Fair scheduling policy in addition to priority-based scheduling
- Fixed allocation of processes to CPU cores
- Access to pre-existing files in the file-system only
- Constraints for AP API usage by Applications
- Execution of authenticated code only

3.3.7 Agile

Although not directly reflected in the platform functionalities, the AP aims to be adaptive to different product development processes, especially agile based processes. For agile based development, it is critical that the underlying architecture of the system is

incrementally scalable, with the possibility of updating the system after its deployment. The architecture of AP should allow this.

3.4 Integration of Classic, Adaptive and Non-AUTOSAR ECUs

As described in previous sections, AP will not replace CP or Non-AUTOSAR platforms in IVI/COTS. Rather, it will interact with these platforms and external backend systems such as road-side infrastructures, to form an integrated system (see Figures 3.1 and 3.2). As an example, CP already incorporates SOME/IP, which is also supported by AP, among other protocols.

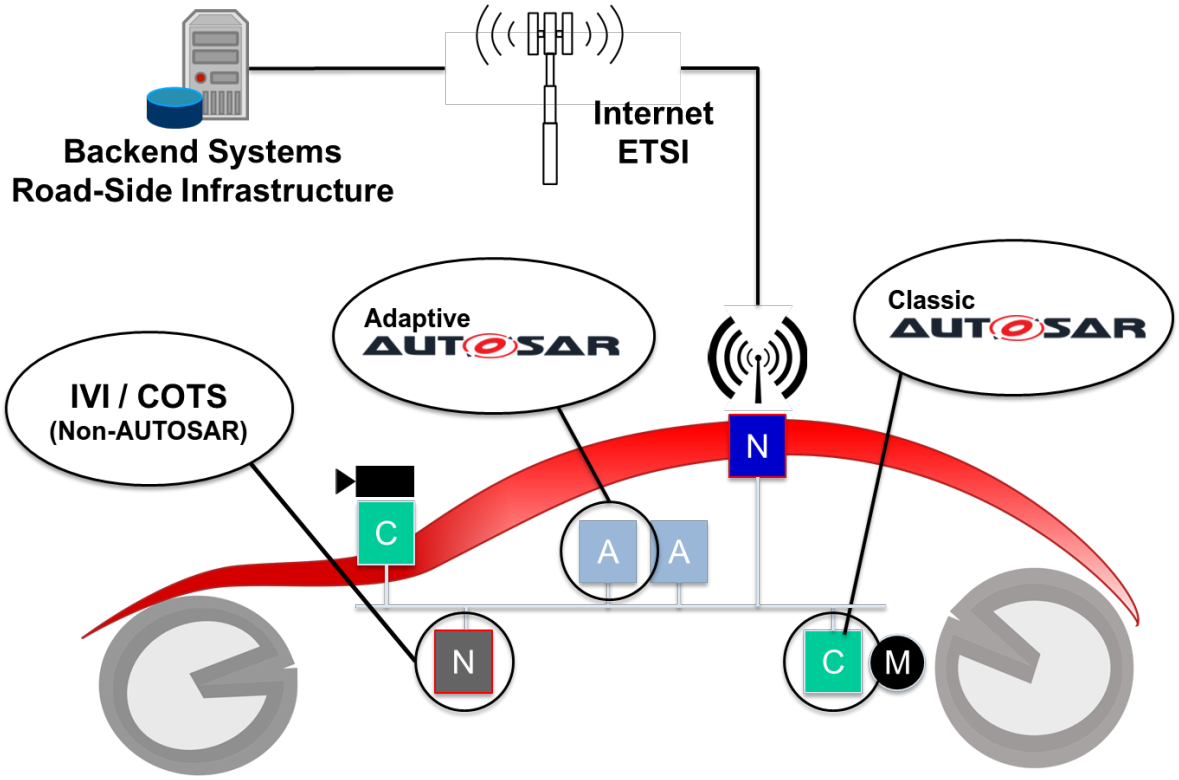

Figure 3.1: Exemplary deployment of different platforms

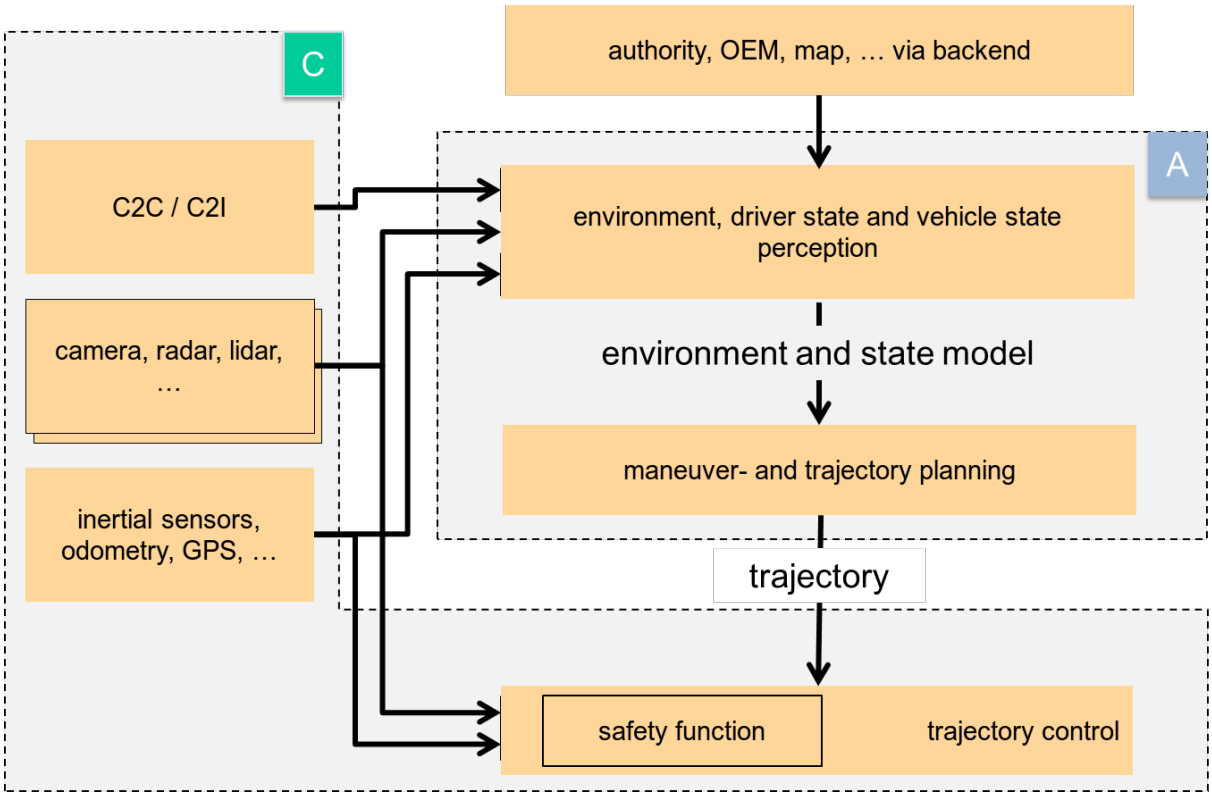

Figure 3.2: Exemplary interactions of AP and CP

3.5 Scope of specification

AP defines the runtime system architecture, what constitutes a platform, and what functionalities and interfaces it provides. It also defines machine-readable models that are used in the development of such a system. The specification should provide necessary information on developing a system using the platform, and what needs to be met to implement the platform itself.

4 Architecture

4.1 Logical view

4.1.1 ARA

Figure 4.1 shows the architecture of AP. The **Adaptive Applications (AA)**,or just **Applications** run on top of **ARA, AUTOSAR Runtime for Adaptive applications**. ARA consists of application interfaces provided by **Functional Clusters**, or **FCs** which belong to either **Adaptive Platform Foundation**, **Adaptive Platform Services**, **Standard Application/Interfaces** or **Vehicle Services**. Adaptive Platform Foundation provides fundamental functionalities of AP, and Adaptive Platform Services provide platform standard services of AP. Standard Application/Interfaces FCs are the Applications or Interfaces standardized by AP, and Vehicle Service FCs are the services that provide vehicle-level functionalities. Note that Any AA can also provide Services to other AA.

The interface of Functional Clusters, either they are those of Adaptive Platform Foundation or Adaptive Platform Services, are indifferent from AA point of view - they just provide specified C++ interface or any other language bindings AP may support in future. There are indeed differences under the hood. Also, note that underneath the ARA interface, including the libraries of ARA invoked in the AA contexts, may use other interfaces than ARA to implement the specification of AP and it is up to the design of AP implementation.

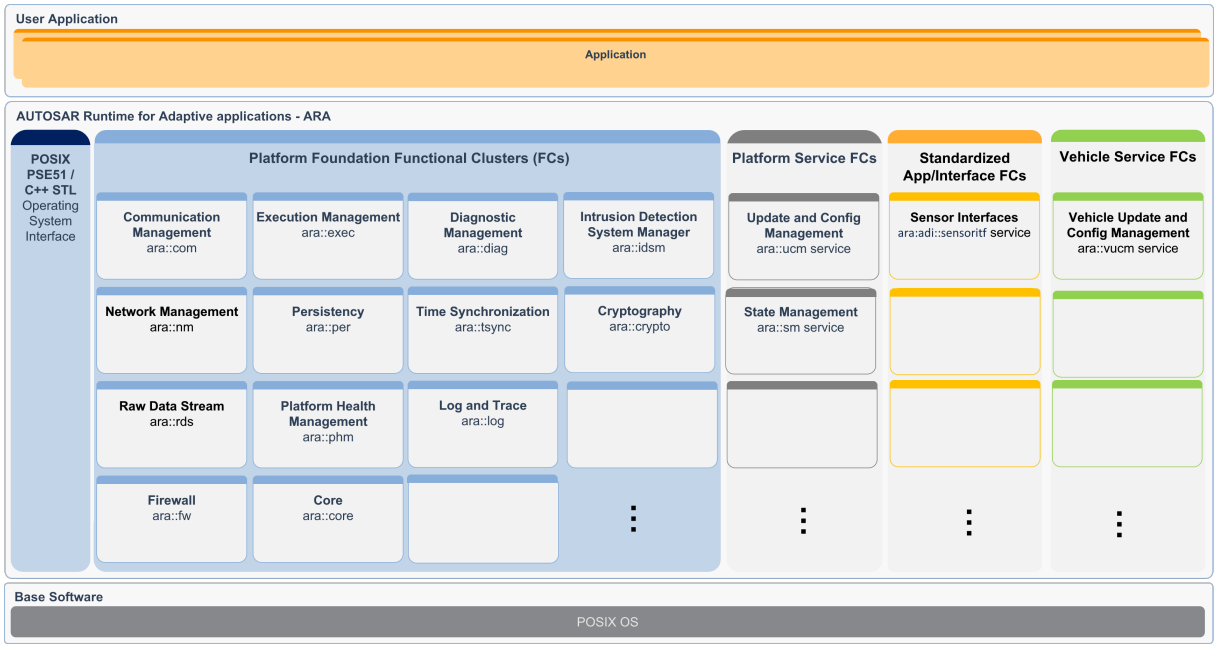

Figure 4.1: AP architecture logical view

Be aware that Figure 4.1 contains Functional Clusters that are not part of the current release of AP, to provide a better idea of overall structure. Further new Functional Clusters not shown here may well be added future releases of AP.

4.1.2 Supported protocols, safety and security features

Adaptive Platform supports a series of well-known protocols, safety and security features. The figures and tables below provide overviews.

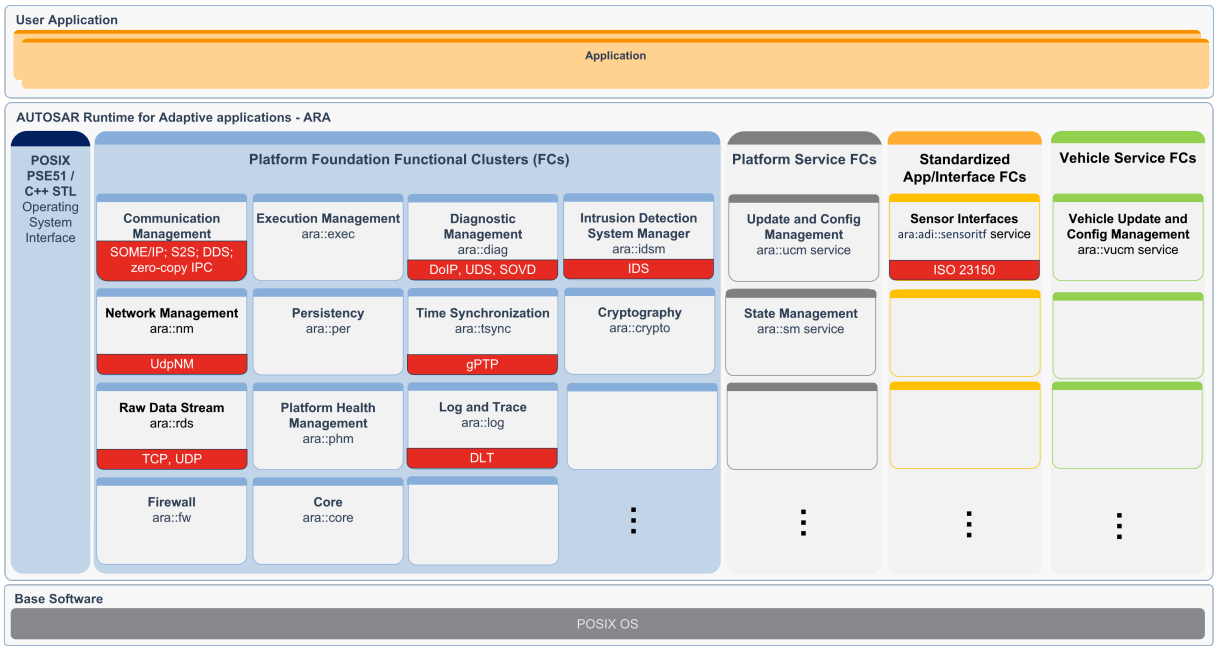

Figure 4.2: Supported protocols

Functional Cluster	Supported protocols
Communication Management	SOME/IP protocol incl. service discovery protocol
	S2S - Signal2Service
	DDS - Date Distribution Service (by OMG)
	zero-copy IPC - Inter-Process Communication
Diagnostic Management	DoIP - Diagnostics over IP (ISO 13400-2)
	UDS - Unified Diagnostic Services (ISO 14229-1)
	SOVD - Service Oriented Vehicle Diagnostics (by ASAM)
Intrusion Detection System Manager	IDS - Intrusion Detection System Protocol
Sensor Interfaces	ISO 23150 - Data communication between sensors and data fusion unit for automated driving functions
Network Management	UdpNM - AUTOSAR Network Management Protocol
Time Synchronization	gPTP = IEEE 802.1AS
	AUTOSAR provides the Time Synchronization Protocol Specification which is an extension and profiling of this IEEE Norm
Raw Data Stream	TCP - Transmission Control Protocol for IPv4 and IPv6
	UDP - User Datagram Protocol for IPv4 and IPv6
Log and Trace	DLT - Log and Trace Protocol Specification

Table 4.1: Supported Protocols

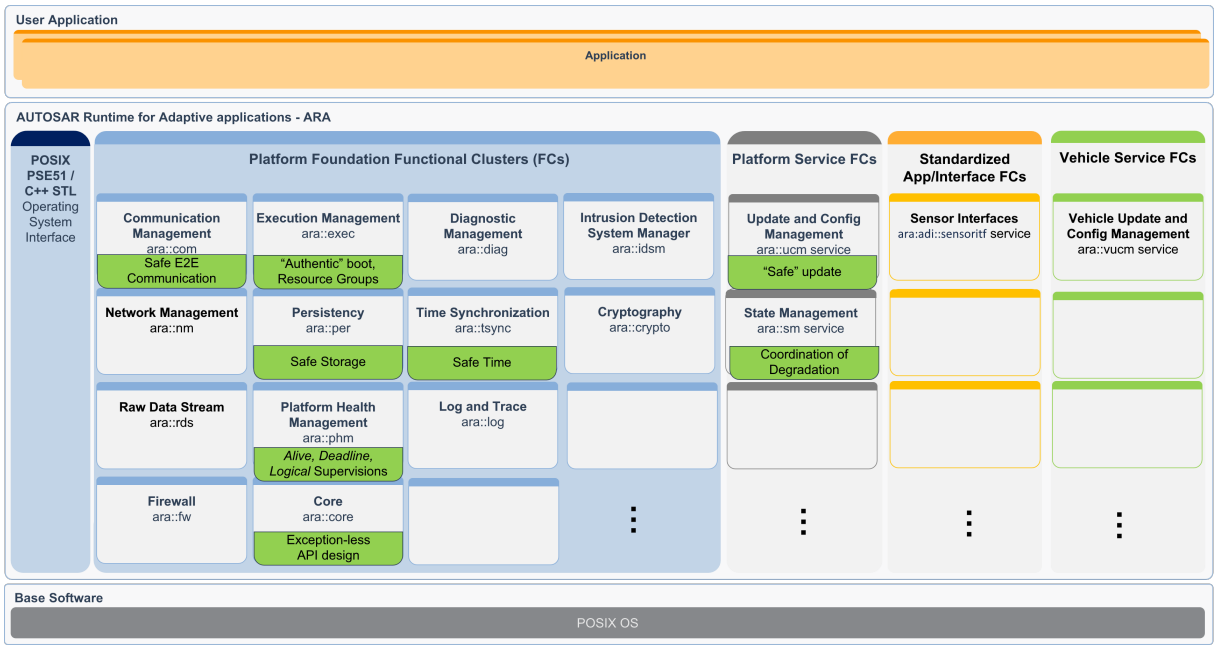

Figure 4.3: Safety features

Functional Cluster	Safety feature
Communication Management	Safe E2E Communication
Execution Management	Trusted Platform
	Resource Groups
Update and Configuration Management	Safe update
Persistency	Safe Storage
Time Synchronization	Safe Time
State Management	Coordination of Degradation
Platform Health Management	Alive/Deadline/Logical Supervisions
	Watchdog
Core	Exception-less API design

Table 4.2: Safety features

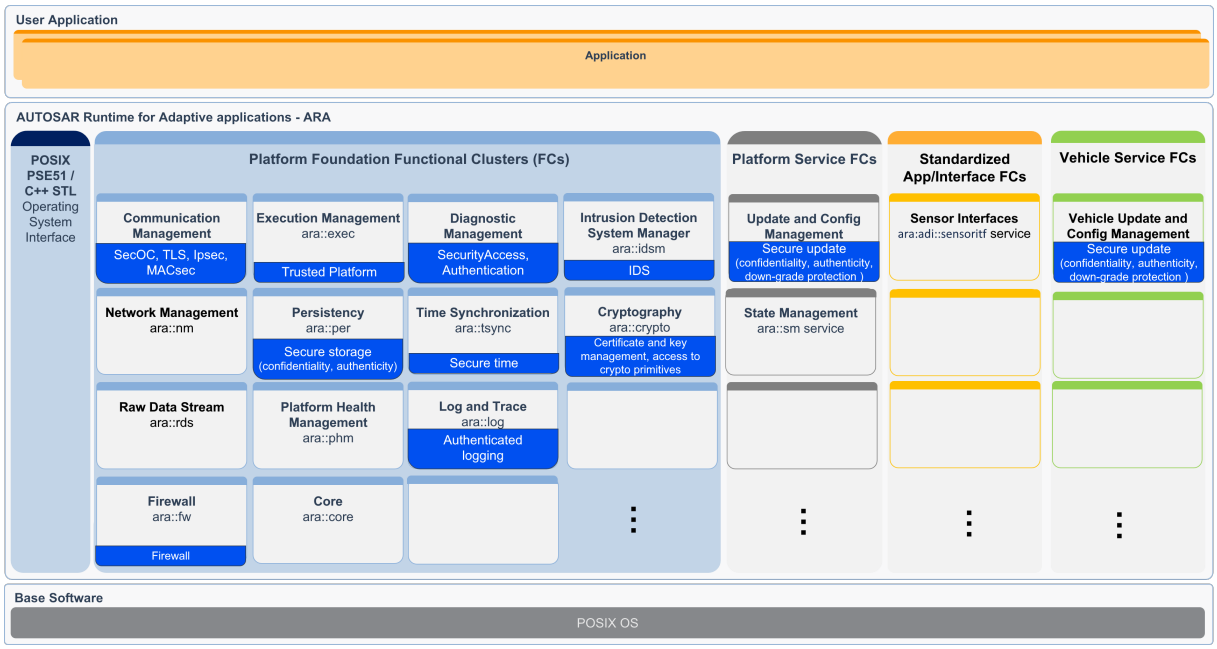

Figure 4.4: Security features

Functional Cluster	Security features
Communication Management	SecOC - Specification of Secure Onboard Communication Protocol (PRS UID 969)
	(D)TLS - (Datagram) Transport Layer Security (by IETF)
	IPSeC - Internet Protocol Security (by IETF)
	MACsec - MAC Security IEEE 802.1AE
Execution Management	Trusted Platform
Diagnostic Management	SecurityAccess - Service 0x27 in UDS
	Authentication - Service 0x29 in UDS
	Authorization - SOVD
Intrusion Detection System Manager	IDS - Intrusion Detection monitoring and reporting
Update and Configuration Management	Secure update - confidentiality/authenticity/down-grade protection
Vehicle Update and Configuration Management	Secure update - confidentiality/authenticity/down-grade protection
Persistency	Secure storage - confidentiality/authenticity
Time Synchronization	Secure time - authenticated time synchronization
Cryptography	Certificate and key management/access to crypto primitives
	Access to secure hardware (e.g TPM/HSM/TEE)
Log and Trace	Authenticated services for logging
Firewall	Filtering on inbound messages based on rules

Table 4.3: Security features

4.1.3 Language binding, C++ Standard Library, and POSIX API

The language binding of these API is based on C++, and the C++ Standard library is also available as part of ARA. Regarding the OS API, only PSE51 interface, a single-process profile of POSIX standard is available as part of ARA. The PSE51 has been selected to offer portability for existing POSIX applications and to achieve freedom of interference among applications.

Note that the C++ Standard Library contains many interfaces based on POSIX, including multi-threading APIs. It is recommended not to mix the C++ Standard library threading interface with the native PSE51 threading interface to avoid complications. Unfortunately, the C++ Standard Library does not cover all the PSE51 functionalities, such as setting a thread scheduling policy. In such cases, the combined use of both interfaces may be necessary.

4.1.4 Application launch and shutdown

Lifecycles of applications are managed by Execution Management (EM). Loading/launching of an application is managed by using the functionalities of EM, and it needs appropriate configuration at system integration time or at runtime to launch an application. In fact, all the Functional Clusters are applications from EM point of view, and they are also launched in the same manner, except for EM itself. Figure 4.5 illustrates different types of applications within and on AP.

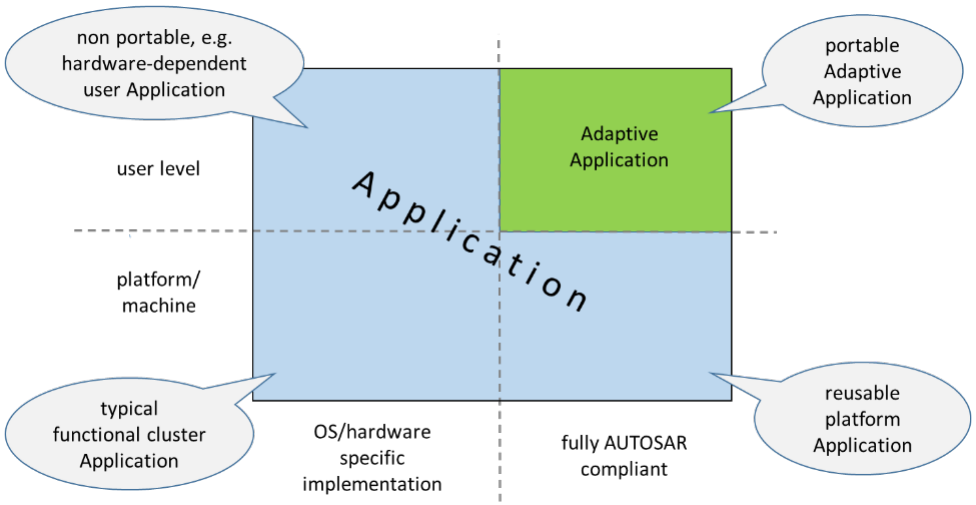

Figure 4.5: Applications

Note that decisions on which and when the application starts or terminates are not made by EM. A special FC, called State Management (SM), is the controller, commanding EM based on the design of a system, arbitrating different states thus controlling the overall system behavior. Since the system here refers to the whole machine AP and its application are running, the internal behavior thus the implementation is project specific. The SM also interact with other FCs to coordinate the overall machine behavior. The SM should use only the standard ARA interface to maintain portability among different AP stack implementations.

4.1.5 Application interactions

Regarding the interaction between AAs, PSE51 do not include IPC (Inter-Process-Communication), so there is no direct interface to interact between AAs. Communi-

cation Management (CM) is the only explicit interface. CM also provides Service Oriented Communication for both intra-machine and inter-machine, which are transparent to applications. CM handles routing of Service requests/replies regardless of the topological deployment of Service and client applications. Note that other ARA interfaces may internally trigger interactions between AAs, however, this is not an explicit communication interface but just a byproduct of functionalities provided by the respective ARA interfaces.

4.1.6 Non-standard interfaces

AA and Functional Clusters may use any non-standard interfaces, provided that they do not conflict with the standard AP functionalities and also that they conform to the safety/security requirements of the project. Unless they are pure application local runtime libraries, care should be taken to keep such use minimal, as this will impact the software portability onto other AP implementations.

4.2 Physical view

The physical architecture [1] of AP is discussed here. Note that most of the contents in this section are for illustration purpose only, and do not constitute the formal requirement specification of AP, as the internals of AP is implementation-defined. Any formal requirement on the AP implementation is explicitly stated. As an additional source of information, refer to [4] which describes the AP internal architecture in more detail.

4.2.1 OS, processes, and threads

The AP Operating System is required to provide multi-process POSIX OS capability. Each AA is implemented as an independent process, with its own logical memory space and namespace. Note that a single AA may contain multiple processes, and this may be deployed onto a single AP instance or distributed over multiple AP instances. From the module organization point of view, each process is instantiated by OS from an executable. Multiple processes may be instantiated from a single executable. Also, AA may constitute multiple executables.

Functional Clusters are also typically implemented as processes. A Functional Cluster may also be implemented with a single process or multiple (sub) processes. The Adaptive Platform Services and the non-platform Services are also implemented as processes.

All these processes can be a single-threaded process or a multi-threaded process. However, the OS API they can use differs depending on which logical layer the pro-

[1] The 'physical architecture' here means mainly the Process View, Physical View, and some Development View as described in [5].

cesses belong to. If they are AAs running on top of ARA, then they should only use PSE51. If a process is one of the Functional Clusters, it is free to use any OS interface available.

In summary, from the OS point of view, the AP and AA forms just a set of processes, each containing one or multiple threads - there is no difference among these processes, though it is up to the implementation of AP to offer any sort of partitioning. These processes do interact with each other through IPC or any other OS functionalities available. Note that AA processes, may not use IPC directly and can only communicate via ARA.

4.2.2 Library-based or Service based Functional Cluster implementation

As in Figure 4.1, a Functional Cluster can be an Adaptive Platform Foundation module or an Adaptive Platform Service. As described previously, these are generally both processes. For them to interact with AAs, which are also processes, they need to use IPC. There are two alternative designs to achieve this. One is "Library-based" design, in which the interface library, provided by the Functional Cluster and linked to AA, calls IPC directly. The other is "Service-based" design, where the process uses Communication Management functionality and has a Server proxy library linked to the AA. The proxy library calls Communication Management interface, which coordinates IPC between the AA process and Server process. Note it is implementation-defined whether AA only directly performs IPC with Communication Management or mix with direct IPC with the Server through the proxy library.

A general guideline to select a design for Functional Cluster is that if it is only used locally in an AP instance, the Library-based design is more appropriate, as it is simpler and can be more efficient. If it is used from other AP instance in a distributed fashion, it is advised to employ the Service-based design, as the Communication Management provides transparent communication regardless of the locations of the client AA and Service. Functional Clusters belonging to Adaptive Platform Foundation are "Library-based" and Adaptive Platform Services are "Service-based" as the name rightly indicate.

Finally, note that it is allowed for an implementation of an FC to not to have a process but realize in the form of a library, running in the context of AA process, as long as it fulfills the defined RS and SWS of the FC. In this case, the interaction between an AA and the FC will be regular procedure call instead of IPC-based as described previously.

4.2.3 The interaction between Functional Clusters

In general, the Functional Clusters may interact with each other in the AP implementation-specific ways, as they are not bound to ARA interfaces, like for example PSE51, that restricts the use of IPC. It may indeed use ARA interfaces of other Functional Clusters, which are `public` interfaces. One typical interaction model be-

tween Functional Clusters is to use implementation-specific `protected` methods to provide access to a Functional Cluster for other Functional Clusters.

Also, from AP18-03, a new concept of Inter-Functional-Cluster (IFC) interface has been introduced. It describes the interface an FC provides to other FCs. Note that it is not part of ARA, nor does it constitute formal specification requirements to AP implementations. These are provided to facilitate the development of the AP specification by clarifying the interaction between FCs, and they may also provide better architectural views of AP for the users of AP specification. The interfaces are described in the Annex of respective FC SWS.

4.2.4 Machine/hardware

The AP regards hardware it runs on as a `Machine`. The rationale behind that is to achieve a consistent platform view regardless of any virtualization technology which might be used. The Machine might be a real physical machine, a fully-virtualized machine, a para-virtualized OS, an OS-level-virtualized container or any other virtualized environment.

On hardware, there can be one or more `Machine`s, and only a single instance of AP runs on a machine. It is generally assumed that this 'hardware' includes a single chip, hosting a single or multiple `Machine`s. However, it is also possible that multiple chips form a single `Machine` if the AP implementation allows it.

4.3 Methodology and Manifest

The support for distributed, independent, and agile development of functional applications requires a standardized approach to the development methodology. AUTOSAR adaptive methodology involves the standardization of **work products** for the description of artifacts like services, applications, machines, and their configuration; and the respective **tasks** to define how these work products shall interact to achieve the exchange of design information for the various activities required for the development of products for the adaptive platform.

Figure 4.6 illustrates a draft overview of how adaptive methodology might be implemented. For the details of these steps see [6].

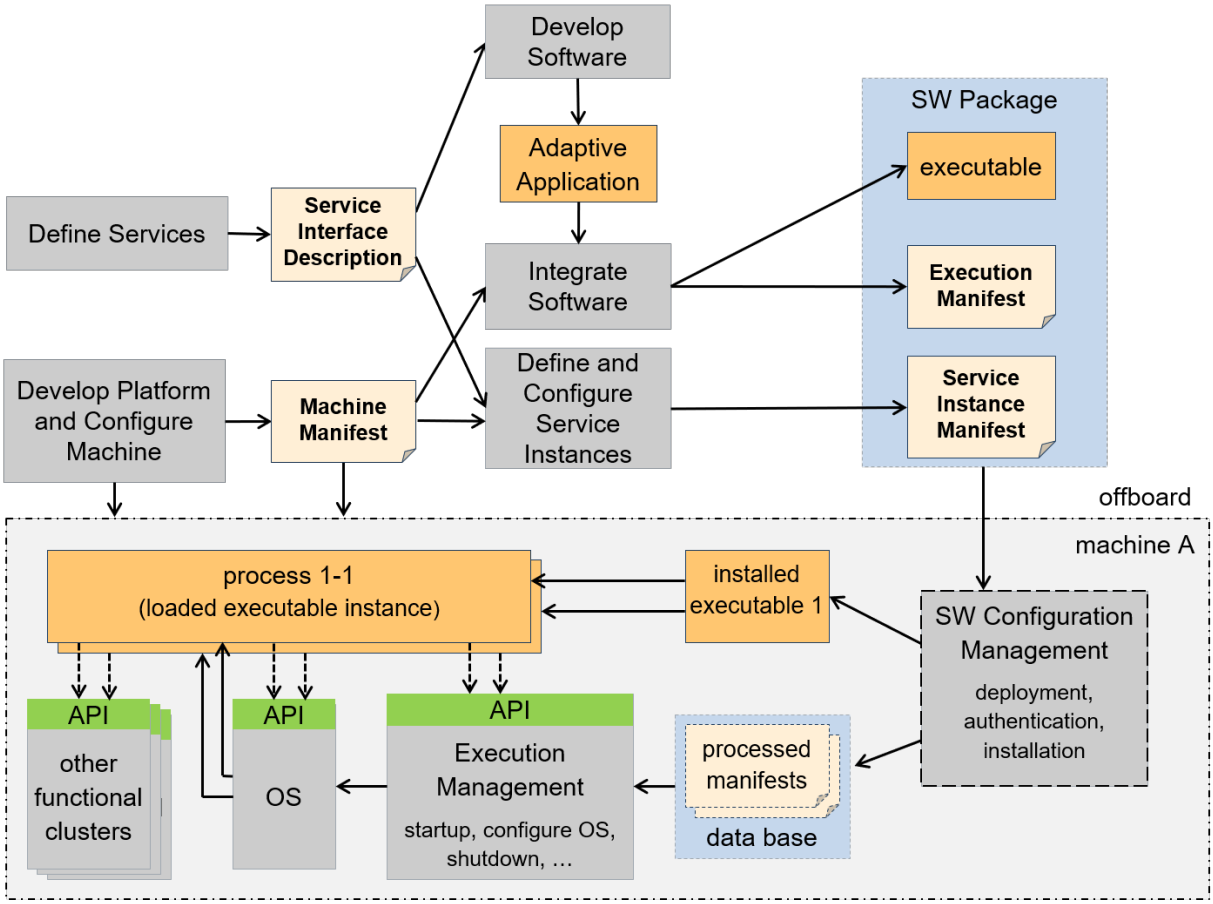

Figure 4.6: AP development workflow

4.4 Manifest

A Manifest represents a piece of AUTOSAR model description that is created to support the configuration of an AUTOSAR AP product and which is uploaded to the AUTOSAR AP product, potentially in combination with other artifacts (like binary files) that contain executable code to which the Manifest applies.

The usage of a Manifest is limited to the AUTOSAR AP. This does not mean, however, that all ARXML produced in a development project that targets the AUTOSAR AP is automatically considered a Manifest. In fact, the AUTOSAR AP is usually not exclusively used in a vehicle project.

A typical vehicle will most likely be also equipped with a number of ECUs developed on the AUTOSAR CP and the system design for the entire vehicle will, therefore, have to cover both - ECUs built on top of the AUTOSAR CP and ECUs created on top of the AUTOSAR AP.

In principle, the term Manifest could be defined such that there is conceptually just one "Manifest" and every deployment aspect would be handled in this context. This does not seem appropriate because it became apparent that manifest-related model-

elements exist that are relevant in entirely different phases of a typical development project.

This aspect is taken as the main motivation that next to the application design it is necessary to subdivide the definition of the term Manifest in three different partitions:

- **Application Design** This kind of description specifies all design-related aspects that apply to the creation of application software for the AUTOSAR AP. It is not necessarily required to be deployed to the adaptive platform machine, but the application design aids the definition of the deployment of application software in the Execution manifest and Service Instance Manifest.
- **Execution Manifest** his kind of Manifest is used to specify the deployment-related information of applications running on the AUTOSAR AP. An Execution manifest is bundled with the actual executable code to support the integration of the executable code onto the machine.
- **Service Instance Manifest** This kind of Manifest is used to specify how service-oriented communication is configured in terms of the requirements of the underlying transport protocols. A Service Instance Manifest is bundled with the actual executable code that implements the respective usage of service-oriented communication.
- **Machine Manifest** This kind of Manifest is supposed to describe deployment-related content that applies to the configuration of just the underlying machine (i.e. without any applications running on the machine) that runs an AUTOSAR AP. A Machine Manifest is bundled with the software taken to establish an instance of the AUTOSAR AP.

The temporal division between the definition (and usage) of different kinds of Manifest leads to the conclusion that in most cases different physical files will be used to store the content of the three kinds of Manifest.

In addition to the Application Design and the different kinds of Manifest, the AUTOSAR Methodology supports a **System Design** with the possibility to describe Software Components of both AUTOSAR Platforms that will be used in a System in one single model. The Software Components of the different AUTOSAR platforms may communicate in a service-oriented way with each other. But it is also possible to describe a mapping of Signals to Services to create a bridge between the service-oriented communication and the signal-based communication.

4.5 Application Design

The application design describes all design-related modeling that applies to the creation of application software for the AUTOSAR AP.

Application Design focuses on the following aspects:

- Data types used to classify information for the software design and implementation
- Service interfaces as the pivotal element for service-oriented communication
- Definition how service-oriented communication is accessible by the application
- Persistency Interfaces as the pivotal element to access persistent data and files
- Definition how persistent storage is accessible by the application
- Definition how files are accessible by the application
- Definition how crypto software is accessible by the application
- Definition how the Platform Health Management is accessible by the application
- Definition how Time Bases are accessible by the application
- Serialization properties to define the characteristics of how data is serialized for the transport on the network
- Description of client and server capabilities
- Grouping of applications in order to ease the deployment of software.

The artifacts defined in the application design are independent of a specific deployment of the application software and thus ease the reuse of application implementations for different deployment scenarios.

4.6 Execution manifest

The purpose of the execution manifest is to provide information that is needed for the actual deployment of an application onto the AUTOSAR AP.

The general idea is to keep the application software code as independent as possible from the deployment scenario to increase the odds that the application software can be reused in different deployment scenarios.

With the execution manifest the instantiation of applications is controlled, thus it is possible to

- instantiate the same application software several times on the same machine, or to
- deploy the application software to several machines and instantiate the application software per machine.

The Execution manifest focuses on the following aspects:

- Startup configuration to define how the application instance shall be started. The startup includes the definition of startup options and access roles. Each startup may be dependent on machines states and/or function group states.

- Resource Management, in particular resource group assignments.

4.7 Service Instance Manifest

The implementation of service-oriented communication on the network requires configuration which is specific to the used communication technology (e.g. SOME/IP). Since the communication infrastructure shall behave the same on the provider and the requesters of a service, the implementation of the service must be compatible on both sides.

The Service Instance Manifest focuses on the following aspects:

- Service interface deployment to define how a service shall be represented on the specific communication technology.
- Service instance deployment to define for specific provided and required service instances the required credentials for the communication technology.
- The configuration of E2E protection
- The configuration of Security protection
- The configuration of Log and Trace

4.8 Machine Manifest

The machine manifest allows to configure the actual adaptive platform instance running on specific hardware (machine).

The Machine Manifest focuses on the following aspects:

- Configuration of the network connection and defining the basic credentials for the network technology (e.g. for Ethernet this involves setting of a static IP address or the definition of DHCP).
- Configuration of the service discovery technology (e.g. for SOME/IP this involves the definition of the IP port and IP multicast address to be used).
- Definition of the used machine states
- Definition of the used function groups
- Configuration of the adaptive platform functional cluster implementations (e.g. the operating system provides a list of OS users with specific rights).
- The configuration of the Crypto platform Module
- The configuration of Platform Health Management
- The configuration of Time Synchronization

- Documentation of available hardware resources (e.g. how much RAM is available; how many processor cores are available)

5 Operating System

5.1 Overview

The Operating System (OS) is responsible for run-time scheduling, resource management (including policing memory and time constraints) and inter-process communication for all Applications on the Adaptive Platform. The OS works in conjunction with Execution Management which is responsible for platform initialization and uses the OS to perform the start-up and shut-down of Applications.

The Adaptive Platform does not specify a new Operating System for highly performant processors. Rather, it defines an execution context and Operating System Interface (OSI) for use by Adaptive Applications.

The OSI specification contains application interfaces that are part of ARA, the standard application interface of Adaptive Application. The OS itself may very well provide other interfaces, such as creating processes, that are required by Execution Management to start an Application. However, the interfaces providing such functionality, among others, are not available as part of ARA and it is defined to be platform implementation dependent.

The OSI provides both C and C++ interfaces. In the case of a C program, the application's main source code business logic include C function calls defined in the POSIX standard, namely PSE51 defined in IEEE1003.13 [1]. During compilation, the compiler determines which C library from the platform's operating system provides these C functions and the applications executable shall be linked against at runtime. In case of a C++ program, application software component's source code includes function calls defined in the C++ Standard and its Standard C++ Library.

5.2 POSIX

There are several operating systems on the market, e.g. Linux, that provide POSIX compliant interfaces. However, applications are required to use a more restricted API to the operating systems as compared to the platform services and foundation.

The general assumption is that a user Application shall use PSE51 as OS interface whereas platform Application may use full POSIX. In case more features are needed on application level they will be taken from the POSIX standard and NOT newly specified wherever possible.

The implementation of Adaptive Platform Foundation and Adaptive Platform Services functionality may use further POSIX calls. The use of specific calls will be left open to the implementer and not standardized.

5.3 Scheduling

The Operating System provides multi-threading and multi-process support. The standard scheduling policies are `SCHED_FIFO` and `SCHED_RR`, which are defined by the POSIX standard. Other scheduling policies such as `SCHED_DEADLINE` or any other operating system specific policies are allowed, with the limitation that this may not be portable across different AP implementations.

5.4 Memory management

One of the reasons behind the multi-process support is to realize 'freedom of interferences' among different Functional Clusters and AA. The multi-process support by OS forces each process to be in an independent address space, separated and protected from other processes. Two instances of the same executable run in different address spaces such that they may share the same entry point address and code as well as data values at startup, however, the data will be in different physical pages in memory.

5.5 Resource control

The Operating System provides resource control support as another utility to realize 'freedom of interference'. It puts every process into a resource group. Each resource group defines resource limits (e.g. CPU time and memory), which can be used by its members.

5.6 Device management

Device management is largely Operating System-specific. Intentionally, the Adaptive Platform Foundation favors the creation of services to expose the main system functionalities.

While there is no current plan to standardize the concrete APIs of device drivers themselves, higher-level functionality fulfilled by such drivers may be standardized through Adaptive Platform Services.

5.7 Networking

The main interconnection mechanism between multiple Machines, as well as with other sensors is expected to be based on Ethernet. Consequently, the use of TCP/IP- and UDP/IP-based protocols is clearly described. It is therefore expected that the Operating System will provide such a networking stack.

Applications will transparently benefit from the networking support by using Communication Management. As part of the Adaptive Platform Foundation, additional features like VLAN, IPSEC and more are enabling secure communications within and across systems.

6 Execution Management

6.1 Overview

Execution Management is responsible for all aspects of system execution management including initialization of the Adaptive Platform and the startup/shutdown of Processes. Execution Management works in conjunction with the Operating System to configure the run-time scheduling of Processes.

6.2 System Startup

When the Machine is started, the OS will be initialized and then Execution Management will be launched as the Platform's initial process. Other platform-level Processes (representing Functional Clusters) of the Adaptive Platform Foundation are then launched by Execution Management. After the Adaptive Platform Foundation is up and running, Execution Management continues launching Processes of Adaptive Applications. The startup order of the platform-level and Application-level Processes are determined by the Execution Management, based on the Machine Manifest and dependencies specified in the Execution manifest.

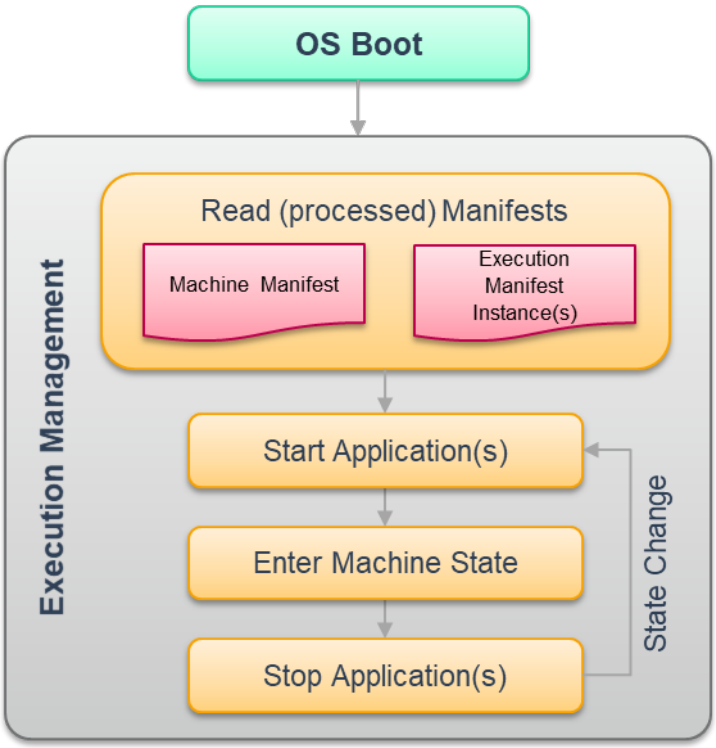

Figure 6.1: AP start-up sequence

An Adaptive Application can consist of multiple Executable elements – which typically would correspond to executable files on the filesystem. Each Executable can have multiple Process configurations.

Execution Management optionally supports authenticated startup where starting from a trust anchor the Adaptive Platform is started while maintaining the chain of trust. During authenticated startup Execution Management validates the authenticity and integrity of applications and will (optionally) prevent their execution if violations are detected. Through these mechanisms, a Trusted Platform can be established.

6.3 Execution Management Responsibilities

Execution Management is responsible for all aspects of Adaptive Platform execution management and Application execution management including:

1. **Platform Lifecycle Startup** Execution Management is launched as part of the Adaptive Platform startup phase and is responsible for the initialization of the Adaptive Platform and deployed Applications.
2. **Application Lifecycle Management** Execution Management is responsible for the ordered startup and shutdown of the deployed Applications. Execution Management determines the set of deployed Applications and derives an ordering for startup/shutdown based on declared execution dependencies.
3. **State Management Support** After Platform Lifecycle Startup, Execution Management provides a service to control the Platform and Application Lifecycle Management by changing Function Group States. This shifts the project specific policy to State Management (see chapter 7).

The Execution Management is not responsible for run-time scheduling of Applications since this is the responsibility of the Operating System. However, Execution Management is responsible for configuration of the OS.

6.4 Resource Limitation

The Adaptive Platform permits execution of multiple Adaptive Applications on the same Machine and thus ensuring freedom from interference is a system property. Hence an incorrectly behaving Adaptive Application should be limited with respect to its ability to affect other applications, for example, an application Process should be prevented from consuming more CPU time than specified due to the potential for consequent impacts on the correct functioning of other applications.

Execution Management supports freedom from interference through the configuration of one or more ResourceGroups to which application's processes are assigned. Each ResourceGroup may then be assigned a limit for CPU time or memory that permits restricting the Application's available resources.

6.5 Trusted Platform

To guarantee the correct function of the system, it is crucial to ensure that the code executed on the platform has legitimate origin. Keeping this property allows the integrator to build a Trusted Platform.

A key property of a system that implements a Trusted Platform is a Trust Anchor (also called Root of Trust). A Trust Anchor is often realized as a public key that is stored in a secure environment, e.g. in non-modifiable persistent memory or in an HSM.

A system designer is responsible to ensure at least that the system starts beginning with a Trust Anchor and that the trust is kept until Execution Management is launched. Depending on the mechanism that is chosen by the system designer to establish the chain of trust, the integrity and authenticity of the entire system may have been checked at this point in the system start-up. However, if the system designer only ensured that the already executed software has been checked regarding integrity and authenticity, Execution Management takes over responsibility on continuing the chain of trust when it takes over control of the system. In this case, the system integrator is responsible to ensure that Execution Management is configured properly.

One example of passing trust from the Trust Anchor to the OS and the Adaptive Platform (i.e. establishing a chain of trust) could look like this: The Trust Anchor - as an authentic entity by definition - authenticates the bootloader before the bootloader is being started. In each subsequent step in the boot process, the to-be-started Executable shall be authenticated first. This authenticity check shall be done by an already authenticated entity, i.e. the authenticity check could be done e.g. by the Executable started previously or by some external entity like an HSM, for example.

After the OS has been authentically started, it shall launch Execution Management as one of its first processes. Before Execution Management is being launched, the OS shall ensure that the authenticity of the Execution Management has been verified by an already authenticated and thus trustworthy entity.

Note: If authentication is not checked by the functionality of the Trust Anchor itself, which is authentic by definition, the Software that shall be applied to verify authenticity of an Executable has to be authenticated before it is applied. For instance, if the Crypto API shall be used to verify authentication of Executables, the Crypto API itself shall be authenticated by some trusted entity before it is used.

Execution Management takes now over the responsibility of authenticating Adaptive Applications before launching them. However, there exists more than one possibility to validate the integrity and authenticity of the Executable code. In [7], a list of possible mechanisms is provided that fulfill this task.

7 State Management

State Management is a unique Functional Cluster that is intended to be mostly an ECU development project specific, and generally, the final implementation is to be performed by the system integrator. It is responsible for all aspects of the operational state of the AUTOSAR Adaptive Platform, including handling of incoming events, prioritization of these events/requests to set the corresponding internal states. State Management may consist of one or more state machines depending on the project needs.

The State Management interact with Adaptive Applications via project specific ara::com service interface consisting of 'Fields' as described below. The interaction between State Management and other Function Clusters shall be done via a standardized interface(s) defined by each Function Cluster.

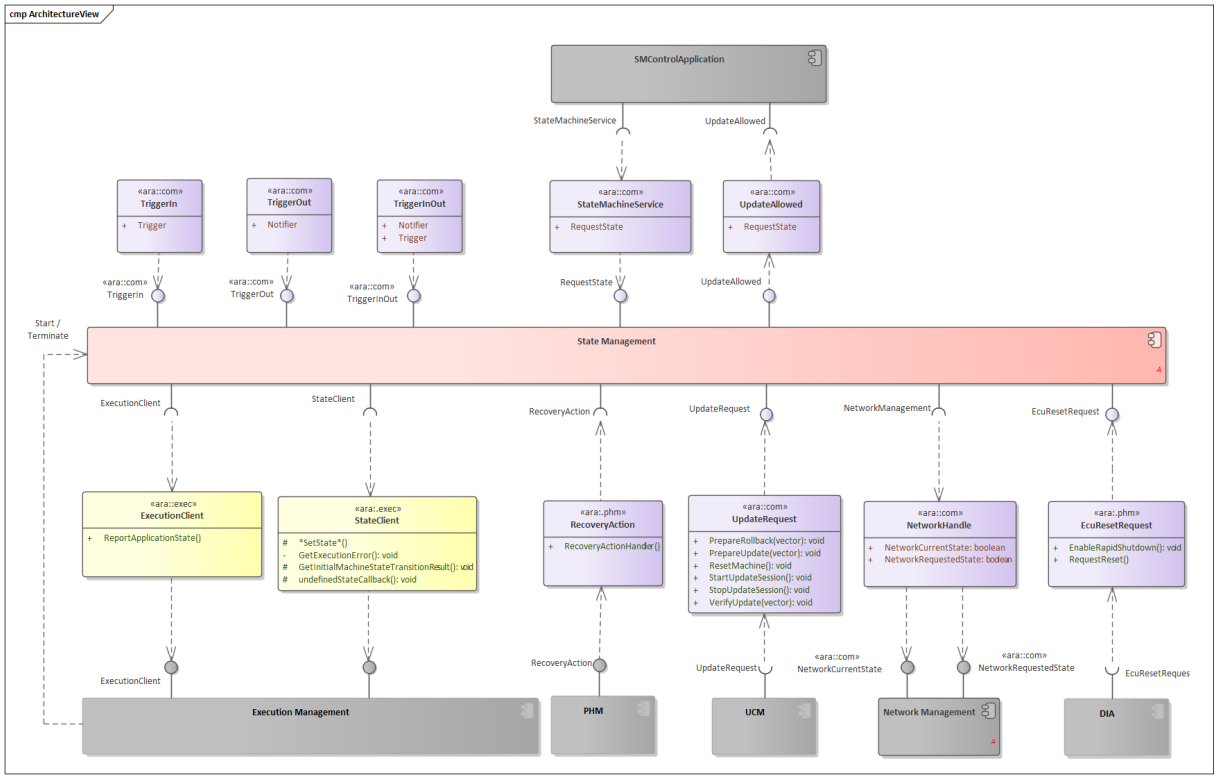

Figure 7.1: State Management interactions

The following effects can be requested by State Management:

- FunctionGroups can be requested to be set to a dedicated state
- (Partial) Networks can be requested to be de- / activated
- The machine can be requested to be shutdown or restarted
- Other Adaptive (Platform) Applications can be influenced in their behavior
- Project-specific actions could be performed

- Recover from (supervision) errors when being informed by Platform Health Management or by Execution Management
- Execution of project specific reset per Diagnostic Addresses on request from Diagnostics
- Preparation and verification of software clusters for being installed, updated or removed on request from Update and Configuration Management

State Management provides a set of 'Trigger' and 'Notifier' fields via ara::com. The SM essentially listens to the 'Triggers', and perform implementation-specific state machine processing internally, and provides the effect to the 'Notifier' fields if there is any.

Since State Management functionality is critical, access from other Functional Clusters or Applications must be secured, e.g. by IAM (Identity and Access Management). State Management is monitored and supervised by Platform Health Management.

The 'lightweight' StateMachine approach has been introduced by State Management to help user of Adaptive Platform, in creation of State Management functionality. StateMachines are designed to cover standard use-cases with minimal configuration effort. Please note that StateMachines are a complementary part of AUTOSAR and thus are optional to use. It is expected that complex use-cases will still require user provided source code.

The StateMachine does not implement project-specific logic. Instead it provides input interface for a project-specific Adaptive Application (i.e. SMControlApplication). This application contains project-specific logic and makes decision which state of the StateMachine should be requested next. Please note that error reaction, to errors reported by Execution Management and/or Platform Health Management, is configured directly inside the StateMachine.

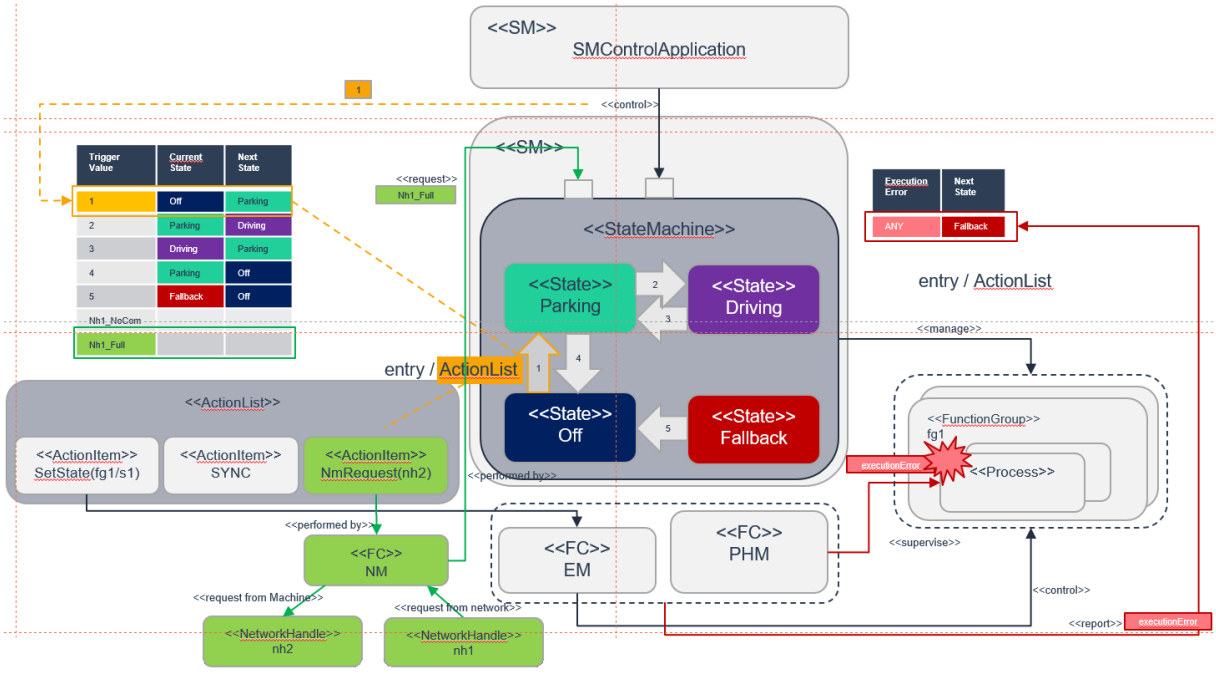

Figure 7.2: Overview of 'lightweight' StateMachine concept

To provide an overview to the StateMachine approach, the following figure shows how the different StateMachine instances, the interfaces for the corresponding StateMachine, the TransitionTables and the ErrorRecoveryTables interact.

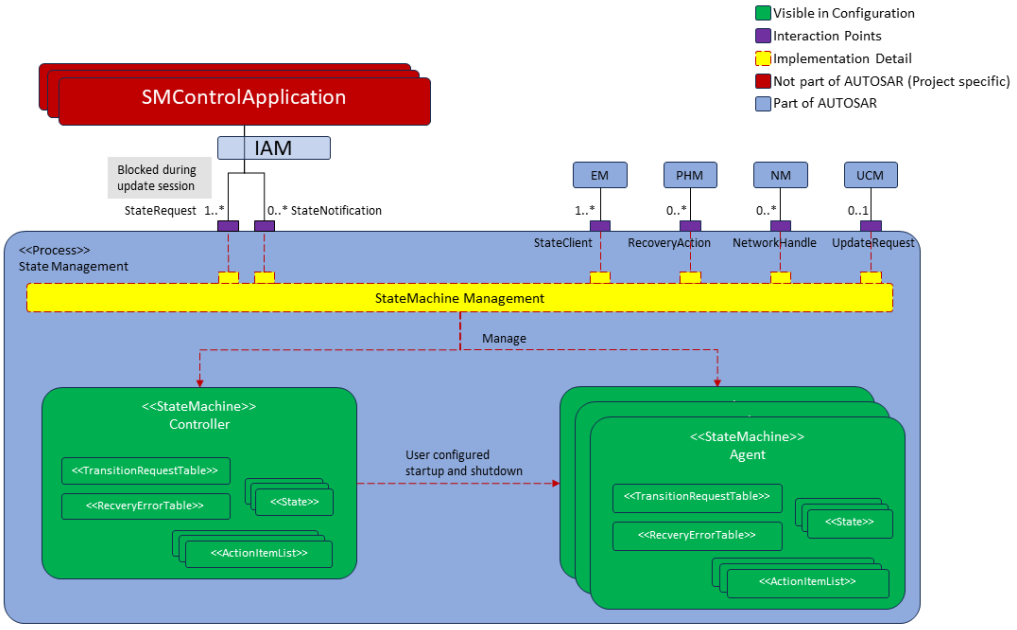

Figure 7.3: StateMachine concept - Big picture

8 Communication Management

8.1 Overview

The Communication Management is responsible for all aspects of communication between applications in a distributed real-time embedded environment.

The concept behind is to abstract from the actual mechanisms to find and connect communication partners such that implementers of application software can focus on the specific purpose of their application.

8.2 Service Oriented Communication

The notion of a service means functionality provided to applications beyond the functionality already provided by the basic operating software. The Communication Management software provides mechanisms to offer or consume such services for intra-machine communication as well as inter-machine communication.

A service consists of a combination of

- Events
- Methods
- Fields

Communication paths between communication partners can be established at design-, at startup- or at run-time. An important component of that mechanism is the `Service Registry` that acts as a brokering instance and is also part of the Communication Management software.

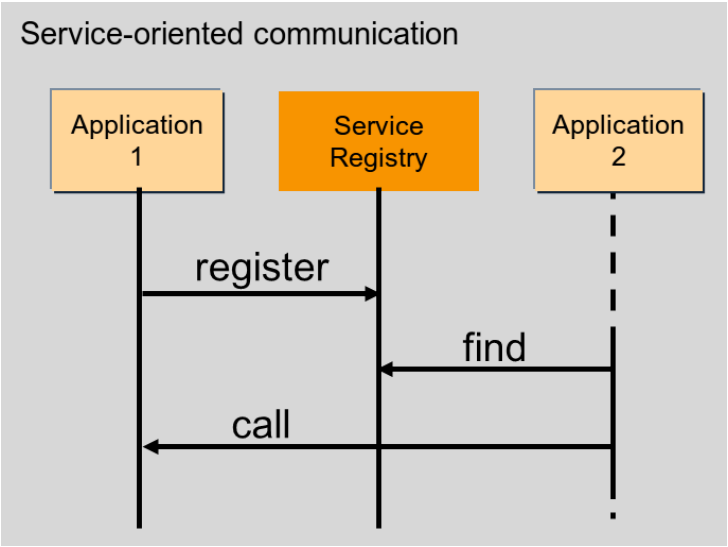

Figure 8.1: Service-oriented communication

Each application that provides services registers these services at the `Service Registry`. To use a service a consuming application needs to find the requested service by querying the `Service Registry`, this process is known as `Service Discovery`.

8.3 Language binding and Network binding

The Communication Management provides standardized means how a defined service is presented to the application implementer (upper layer, `Language Binding`) as well as the respective representation of the service's data on the network (lower layer, `Network Binding`). This assures portability of source code and compatibility of compiled services across different implementations of the platform.

The `Language Binding` defines how the methods, events, and fields of a service are translated into directly accessible identifiers by using convenient features of the targeted programming language. Performance and type safety (as far as supported by the target language) are the primary goals. Therefore, the `Language Binding` is typically implemented by a source code generator that is fed by the service interface definition.

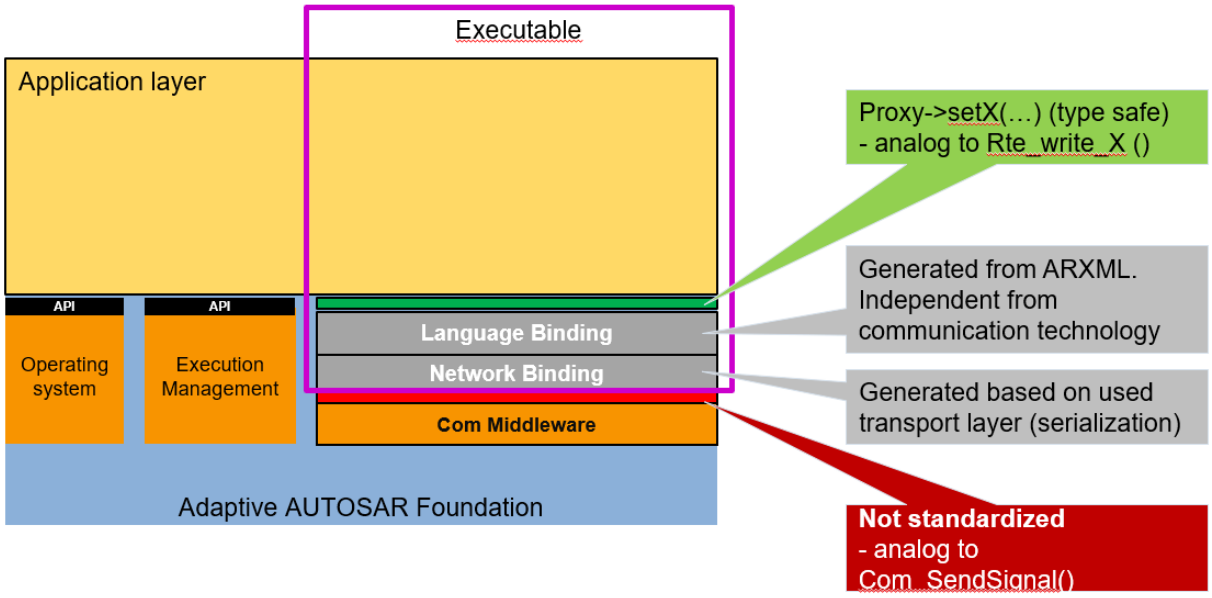

Figure 8.2: Example Language and Network Binding

The `Network Binding` defines how the actual data of a configured service is serialized and bound to a specific network. It can be implemented based on Communication Management configuration (interface definition of the AUTOSAR meta model) either by interpreting a generated service specific recipe or by directly generating the serializing code itself. Currently, Communication Management supports SOME/IP, DDS, IPC (Inter-Process-Communication or any other custom binding), Signal PDU (Signal-Based Network binding) and Signal-Based Static Network binding.

The local `Service Registry` is also part of the `Network Binding`.

Please note: the interface between `Language Binding` and `Network Binding` is considered as a private interface inside Communication Management software. Therefore, a normative specification defining this interface is currently out of scope. Nevertheless, platform vendors are encouraged to define independently such an interface for their software to allow for easy implementation of other `Language Binding`s than C++ together with other `Network Binding`s inside their platform implementation.

8.4 Generated Proxies and Skeletons of C++ Language Binding

The upper layer interface of the C++ Language Binding provides an object-oriented mapping of the services defined in the interface description of the AUTOSAR meta model.

A generator that is part of the development tooling for the Communication Management software generates C++ classes that contain type safe representations of the fields, events, and methods of each respective service.

On the service implementation side, these generated classes are named Service Provider Skeletons. On the client side, they are called Service Requester Proxies.

For Service Methods, a Service Requester Proxy provides mechanisms for synchronous (blocking the caller until the server returns a result) and asynchronous calling (called function returns immediately). A caller can start other activities in parallel and receives the result when the server's return value is available via special features of the Core Type `ara::core::Future` (see Section 18.1.4).

A platform implementation may be configured such that the generator creates mock-up classes for easy development of client functionality when the respective server is not yet available. The same mechanism can also be used for unit testing the client.

Whereas proxy classes can be used directly by the client the Service Provider Skeletons for the C++ binding are just abstract base classes. A service implementation shall derive from the generated base class and implement the respective functionality.

The interfaces of ara::com can also provide proxies and skeletons for safety-related E2E protected communication. These interfaces are designed that compatibility to the applications is assured independent whether E2E protection is switched on or off.

8.5 Static and dynamic configuration

The configuration of communication paths can happen at design-, at startup- or at run-time and is therefore considered either static or dynamic:

- **Full static configuration** Service discovery is not needed at all as the server knows all clients and clients know the server.

- **No discovery by application code** The clients know the server but the server does not know the clients. Event subscription is the only dynamic communication pattern in the application.
- **Full service discovery in the application** No communication paths are known at configuration time. An API for Service discovery allows the application code to choose the service instance at runtime.

8.6 Service Contract Versioning

In SOA environments the client and the provider of a service rely on a contract which covers the service interface and behavior. During the development of a service the service interface or the behavior may change over time. Therefore, service contract versioning has been introduced to differentiate between the different versions of a service. The AUTOSAR Adaptive platform supports contract versioning for the design and for the deployment phase of a service. Additionally, the Service Discovery of a client may be configured to support version backwards-combability. This means that a client service can connect to different provided service versions if these are backwards-compatible to the required service version of the client.

9 Diagnostics

9.1 Overview

The Diagnostic Management (DM) realizes the ISO 14229-5 (UDSonIP) which is based on the ISO 14229-1 (UDS) and ISO 13400-2 (DoIP).

Diagnostic Management represents a functional cluster of the Adaptive Platform on the foundation layer.

The configuration is based on the AUTOSAR Diagnostic Extract Template (DEXT) of the Classic Platform.

The supported Transport Layer is DoIP. DoIP is a vehicle discovery protocol and designed for off-board communication with the diagnostic infrastructure (diagnostic clients, production-/workshop tester).

In-vehicle or for remote diagnostics often other transport protocols are used, wherefore an API to extend the platform with a custom transport layer is provided.

UDS is typically used within the production of a vehicle and within the workshops to be able to repair the vehicle.

9.2 Software Cluster

The atomic updateable/extendable parts are managed by `SoftwareClusters` (SWCL). A `SoftwareCluster` contains all parts which are relevant to update installed or deploy a particular set of new functionalities/applications. Hence the Adaptive Diagnostics Manager supports an own `DiagnosticAddress` for each installed `SoftwareCluster`. But it also supports a single `DiagnosticAddress` for the whole `Machine` or any diagnostic deployment in between. An own `DiagnosticAddress` has always its own Diagnostic Server instance as consequence. An own Diagnostic Server instance per `SoftwareCluster` offers an independent development also for diagnostics like an own independent ODX file. Note that this `SoftwareCluster` is also coupled with the Software Package of UCM so that the `SoftwareCluster` can be updated or newly introduced to a `Machine`.

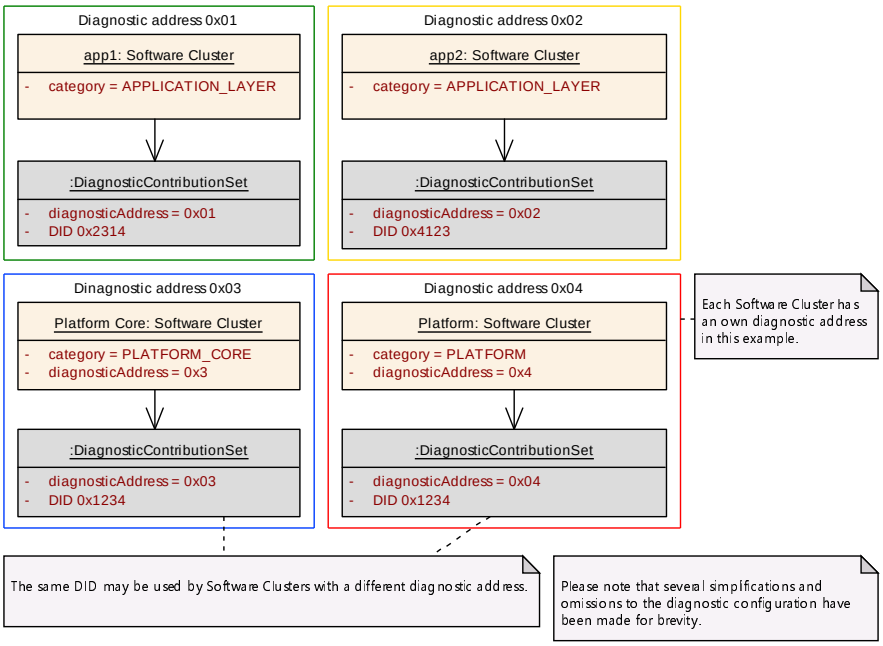

Figure 9.1: One Diagnostic Address Per SWCL

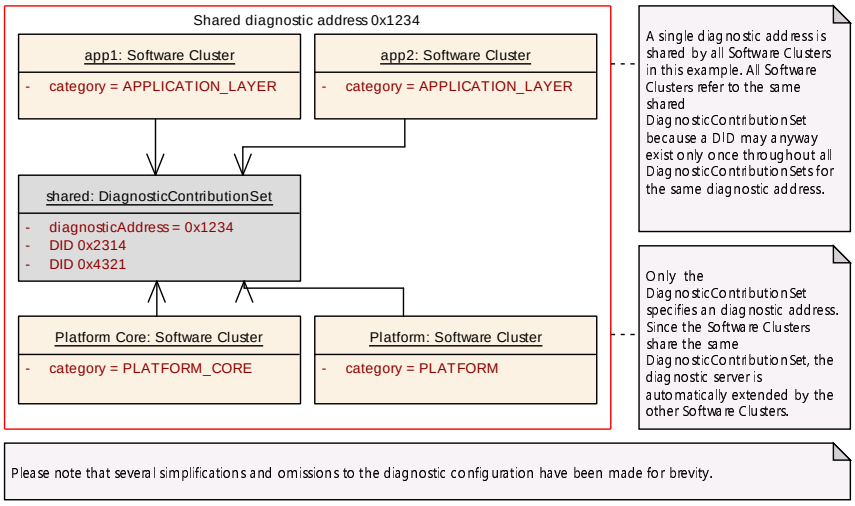

Figure 9.2: Single Diagnostic Address Per Machine

9.3 Diagnostic communication sub-cluster

The diagnostic communication sub-cluster realizes the diagnostic server (like the DCM of the Classic Platform). Currently, the supported services are limited, but the support of further UDS services will be extended in future releases.

DM supports multi client handling according to ISO 14229-1. This allows satisfying the demands of modern vehicle architectures including several diagnostic clients (tester) for data collection, access from the backend and finally some of the classic workshop and production use-cases.

A further feature is the authentication of specific services wire certificates.

9.4 Diagnostic in Adaptive Application (AA)

The DM dispatch as a diagnostic server incoming diagnostic requests (like a routine control or DID service) to the mapped providing port of the corresponding AA.

To realize this the AA needs to provide a specialized DiagnosticPortInterface.

9.5 Typed vs generic interfaces

There are different abstraction levels of DiagnosticPortInterfaces available:

- A RoutineControl message is available as a
  - **Typed interface** The API signature includes all requests- and response message parameters with their primitive types. The DM takes care of the serialization. This API is individual to a specific RoutineControl message.
  - **Generic interface** The API signature includes only a Byte-Vector for the request- and response message. The application is in the responsibility of the request- and response message serialization. The same API could be used for multiple RoutineControl messages.
- A DataIdentifier Message is available as a
  - **Typed interface** The API signature includes all requests- (for writing) and response message (for reading) parameters with their primitive types. The DM takes care of the serialization.
  - **Generic interface** The API signature includes only a Byte-Vector for the request- and response message. The application is in the responsibility of the request- and response message serialization.
  - **DataElement individual** Each request- and response message parameter has its own interface. This is the highest level of abstraction i.e. any change in the request- and response message structure will have no effect on the API. Further, the parameters of the same diagnostic message could be in different processes.

9.6 Diagnostic conversations

As the DM demands pseudo-parallel handling as it is mentioned above, it supports Diagnostic Conversations to reflect a distinct conversation between a Diagnostic Client and a Diagnostic Server. A Diagnostic Server is identified by a target address of the

according UDS request and is dynamically allocated during run-time in the Adaptive Platform.

9.7 Event memory sub-cluster

The event memory sub-cluster is responsible for DiagnosticTroubleCode (DTC) management (like the DEM of the Classic Platform).

An active DTC is representing a certainly detected issue (typically important for production or workshop) in the vehicle. The DM is managing the storage of DTCs and its configured SnapshotRecords (a set of configured environmental data on the occurrence time of the DTC) and/or ExtendedDataRecords (statistical data belonging to the DTC like the number of reoccurrences).

The detection logic is called Diagnostic Monitor. Such a monitor is reporting its recent test result to a DiagnosticEvent in the DM. The UDS DTC status is derived from one or multiple DiagnosticEvent(s).

The DTC can be assigned to PrimaryMemory (accessible via 19 02/04/06) or to configurable UserMemories (accessible via 0x19 17/18/19).

Counter- and Timebase Debouncing are supported. Furthermore, DM offers notifications about internal transitions: interested parties are informed about DTC status byte changes, the need to monitor re-initialization for DiagnosticEvents and if the Snapshot- or ExtendedDataRecord is changed.

A DTC can vanish from the DTC memory if it is not active for a configured amount of Operation Cycles.

The DM supports generalized handling for the enable conditions. Enabling Conditions can be used to control the update of DTCs under special conditions like to disable all network-related DTCs within under-voltage condition.

9.8 Service Oriented Vehicle Diagnostics

With release R22-11 the ASAM related concept SOVD was introduced to the Diagnostics Functional Cluster. The main features are the SOVD gateway, SOVD to UDS translation, backend connectivity, authorization and proximity challenge.

With a further release the SOVD features logging and bulk data were introduced.

10 Persistency

10.1 Overview

Persistency offers mechanisms to applications and other functional clusters of the Adaptive Platform to store information in the non-volatile memory of an Adaptive Machine. The data is available over boot and ignition cycles. Persistency offers standardized interfaces to access the non-volatile memory.

The Persistency API takes `InstanceSpecifiers` for `PersistencyInterfaces` (for applications) or `FunctionalClusterInteractsWithPersistencyDeploymentMappings` (for other functional clusters) as parameters to address different storage locations. The available storage locations fall into two categories:

- Key-Value Storage
- File Storage

Every application or functional cluster may use a combination of multiple of these storage types.

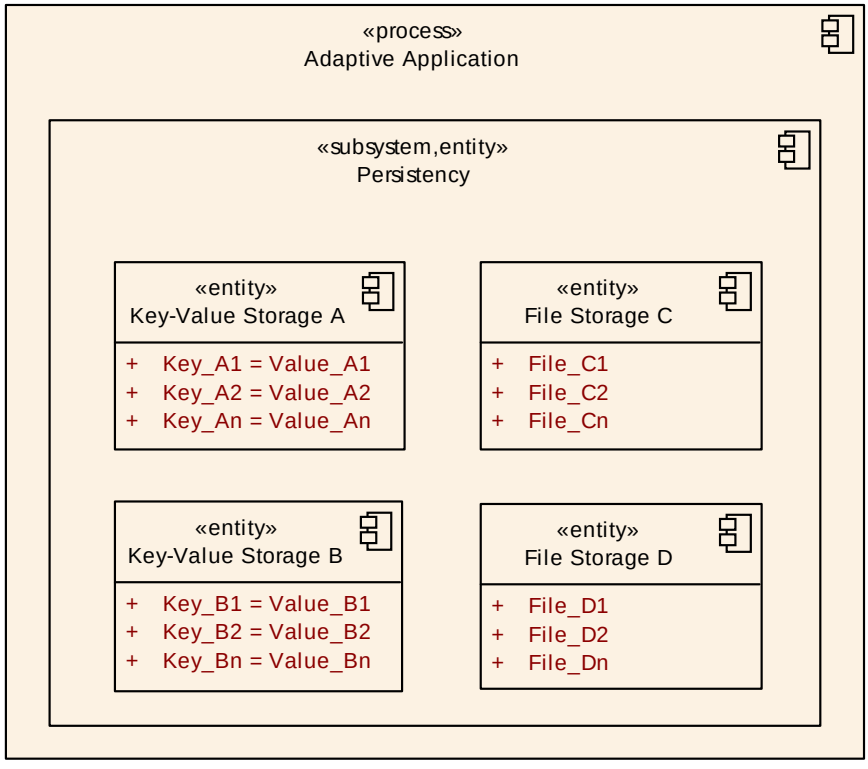

Figure 10.1: Typical usage of `Persistency` within an `Adaptive Application`

Persistent data is always private to one process of one application. There is no mechanism available to share data between different processes using the Persistency. This decision was taken to prevent a second communication path besides the functionality provided by Communication Management.

Persistency is prepared to handle concurrent access from multiple threads of the same application, running in the context of the same Process. To create shared access to a Key-Value Storage or File Storage, either the SharedHandle returned by OpenKeyValueStorage and OpenFileStorage can be passed on (i.e. copied) to another thread or OpenKeyValueStorage and OpenFileStorage can be called in independent threads for the same Key-Value Storage or File Storage, respectively.

Persistency is able to take care of the integrity of the stored data. It uses redundant information to detect data corruption. The redundant information consists of CRC codes, Hash values, and "M out of N" copies. These mechanisms can be used either together or independently.

Persistency offers also safe storage. This is basically implemented using redundancy, but with the additional feature of letting the application know if there was any problem with the stored data, even if it could be recovered using redundant data.

Persistency offers statistics regarding the number of used resources. It can also ensure that a given quota is not exceeded.

Persistency offers encryption for stored data to make sure that sensitive data will be encrypted before storing it on a physical device, ensuring confidentiality of the stored data. Persistency also offers authentication using a MAC (message authentication code) to ensure authenticity of stored data.

10.2 Key-Value Storage

The Key-Value Storage provides a mechanism to store and retrieve multiple Key-Value pairs in one storage location. The following three kinds of data types are supported directly by Key-Value Storage:

- Data types defined in [8].
- Simple byte arrays that result from a streaming of complex types in the application.
- All Implementation Data Types referred via `dataTypeForSerialization` by a `PersistencyKeyValueStorageInterface` or specialized as `PersistencyDataElement`s of that interface in the Application Design.

To be able to migrate values from one type to another during an update, Persistency offers data type mappings, that map a currently used data type to a set of data types that were used in previous versions for the same key.

The keys need to be unique for each Key-Value Storage and are either defined in the configuration (design and/or deployment) or by using SetValue() at runtime.

10.3 File Storage

Not all data relevant for persistent storage is structured in such a way that Key-Value Storages are a suitable storage mechanism.

For this kind of data the mechanism of File Storage was introduced. A File Storage allows an application or another functional cluster to access a storage location and create one or multiple files in it. These files are identified by unique file names.

To give a better impression of this mechanism, a comparison to a file system helps: a File Storage Port can be understood as a filesystem directory in which an application is allowed to create multiple files, but no sub directories.

10.4 Use cases for handling persistent data for UCM

In general, there are three main use cases supported in UCM for handling adaptive applications over the life cycle of the ECU or Adaptive Machine.

- Installation of new application software to the Adaptive Machine
- Update of existing application software to the Adaptive Machine
- Uninstallation of the existing application software from the Adaptive Machine

In the first two scenarios, the application is triggered by UCM via EM to validate the installation/update, which then triggers Persistency to deploy/update the persistent data of an application, based on the configuration of Persistency in the Manifest.

In the third scenario, UCM removes remaining persistent data using the URIs from the Persistency configuration.

Persistency supports the below-mentioned scenarios for deploying persistent data to Key-Value Storages and File Storages.

- Persistency shall be able to deploy the persistent data that was defined by an application designer during the Adaptive Application installation.
- Persistency shall be able to deploy the persistent data that was changed by an integrator.
- Persistency shall be able to deploy the persistent data that was defined by an integrator.
- Persistency shall be able to overwrite or retain the persistent data as per the update strategies configured for the Key-Value Storage or File Storage when a new version of an application is installed.

In general, the Persistency layer is configured during application design and deployment. Persistency shall be able to use the deployment stage configuration to override the application design configuration. If deployment stage configurations are missing,

then configuration from the application design will be considered for the deployment of persistent data.

11 Time Synchronization

11.1 Overview

Time Synchronization (TS) between different applications and/or ECUs is of paramount importance when the correlation of different events across a distributed system is needed, either to be able to track such events in time or to trigger them at an accurate point in time.

For this reason, a Time Synchronization API is offered to the Application, so it can retrieve the time information synchronized with other Entities / ECUs.

The Time Synchronization functionality is then offered by means of different "Time Base Resources" (from now on referred to as TBR) which are present in the system via a pre-build configuration.

11.2 Design

For the Adaptive Platform, the following three different technologies were considered to fulfill all necessary Time Synchronization requirements:

- StbM of the Classic Platform
- Library chrono - either std::chrono (C++11) or boost::chrono
- The Time POSIX interface

After an analysis of the interfaces of these modules and the Time Synchronization features they cover, the motivation is to design a Time Synchronization API that provides a functionality wrapped around the StbM module of the Classic Platform, but with a std::chrono like flavor.

The following functional aspects are considered by the Time Synchronization module:

- Startup Behavior
- Shutdown Behavior
- Constructor Behavior (Initialization)
- Normal Operation
- Error Handling

The following functional aspects will be considered in future releases:

- Error Classification
- Version Check

11.3 Architecture

The application will have access to a different specialized class implementation for each TBR.

The TBRs are classified in different types. These types have an equivalent design to the types of the time bases offered in the Synchronized Time Base Manager specification [9]. The classification is the following:

- Synchronized Master Time Base
- Offset Master Time Base
- Synchronized Slave Time Base
- Offset Slave Time Base

As in StbM, the TBRs offered by the Time Synchronization module (TS from now on), are also synchronized with other Time Bases on other nodes of a distributed system.

From this handle, the Application will be able to inquire about the type of Time Base offered (which shall be one of the four types presented above) to then obtain a specialized class implementation for that type of Time Base. Moreover, the Application will also be able to create a timer directly.

The TS module itself does not provide means to synchronize TBRs to Time Bases on other nodes and/or ECUs like network time protocols or time agreement protocols.

An implementation of TBRs may have a dedicated cyclic functionality, which retrieves the time information from the Time Synchronization Ethernet module or alike to synchronize the TBRs.

The Application consumes the time information provided and managed by the TBRs. Therefore, the TBRs serve as Time Base brokers, offering access to Synchronized Time Bases. By doing so, the TS module abstracts from the "real" Time Base provider.

12 Network Management

12.1 Overview on Network Management Algorithm

The AUTOSAR NM is based on a decentralized network management strategy, which means that every network node performs activities independently depending only on the NM messages received and/or transmitted within the communication system.

The AUTOSAR NM algorithm is based on periodic NM messages, which are received by all nodes in the cluster via multicast messages.

The reception of NM messages indicates that sending nodes want to keep the NM-cluster awake. If any node is ready to go to sleep mode, it stops sending NM messages, but as long as NM messages from other nodes are received, it postpones the transition to sleep mode. Finally, if a dedicated timer elapses because no NM messages are received any more, every node performs the transition to the sleep mode.

If any node in the NM-cluster requires bus-communication, it can keep the NM-cluster awake by starting the transmission NM messages.

12.2 Architecture

The Adaptive Platform specification describes the functionality, the API design and the configuration of the Network Management for the AUTOSAR Adaptive Platform independently of the underlying communication media used. At the moment only Ethernet is considered but the architecture is kept bus - independent.

The Network Management (NM) is intended to be controlled via State Management as the control of partial network needs to be coordinated with the set of the relevant application via Function Group State of EM controlled by SM. The contents in this chapter do not yet reflect the design.

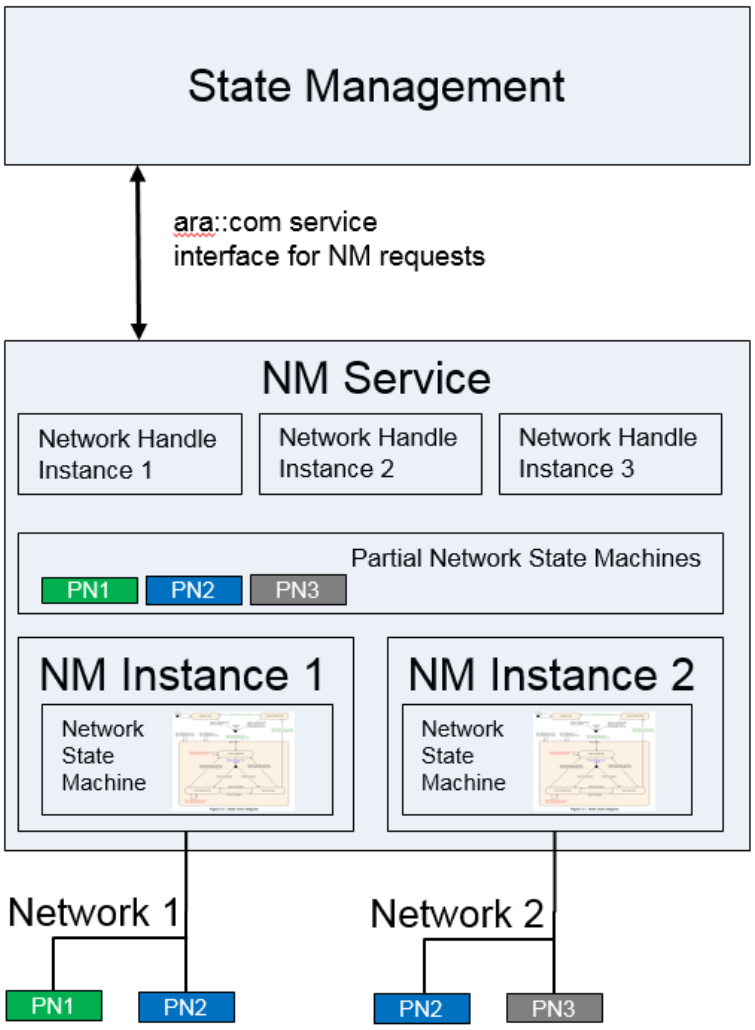

Figure 12.1: Overview NM

Its main purpose is to coordinate the transition between normal operation and bus-sleep mode of the underlying networks (Partial Networks, VLANs or physical channel) in internally coordinated state machines.

It provides a Serviceinterface to the Statemanagement for requesting and releasing networks and querying their actual state. It coordinates the requests of different instances (Network Handles) and provides an aggregated machine request over the network..

If the Partial Networking feature is used the Nm messages can contain Partial Network (PN) requests, giving the ECU the possibility to ignore Nm messages which do not request any PN which is relevant to the ECU. This gives the possibility to shut down the ECU (or parts of it), although communication is still going on in other Partial Networks.

13 Update and Config Management

13.1 Overview

One of the declared goals of the AUTOSAR Adaptive Platform is the ability to flexibly update the software and its configuration through over-the-air updates (OTA). To support changes in the software on an Adaptive Platform, the Update and Configuration Management (UCM) provides an Adaptive Platform service that handles software update requests.

UCM is responsible for updating, installing, removing and keeping a record of the software on an Adaptive Platform. Its role is similar to known package management systems like dpkg or YUM in Linux, with additional functionality to ensure a safe and secure way to update or modify the software on the Adaptive Platform.

V-UCM is providing a standard Adaptive Platform solution to update vehicle software over-the-air or by a diagnostic tester. It is coordinating and distributing packages within a vehicle among several UCMs. Therefore, V-UCM can be considered as an AUTOSAR standard UCM Client.

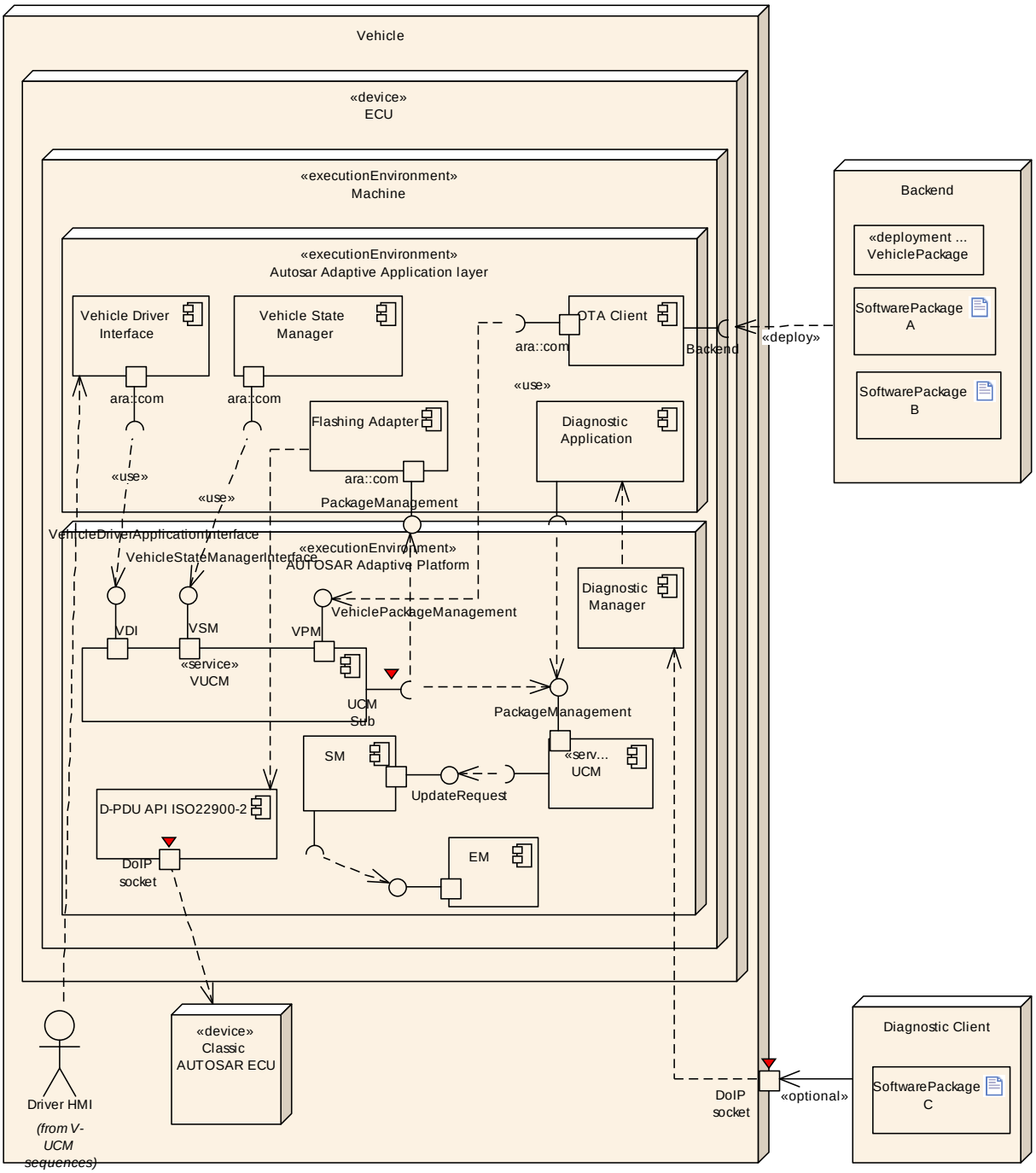

Figure 13.1: Vehicle Update Architecture

13.2 Update protocol

UCM and V-UCM services have been designed to support the software configuration management over vehicle diagnostics and support performing changes in Adaptive Platforms in safe, secure and resource-efficient update processes. To fulfill the requirements to support updates from several clients and to enable fast download, UCM

needs to be capable of transferring Software Packages (UCM input) separately from their processing.

13.2.1 Data transfer

Data transfer is done over ara::com. This enables transferring data into UCM or V-UCM without the need to buffer data on the way from the backend or diagnostic tester. UCM can store packages into a local repository where packages can be processed in the order requested by the UCM client or V-UCM.

The transfer phase can be separated from the processing phase, UCM supports receiving data from multiple clients without limitations.

V-UCM is relying on the same transfer API as UCM but accessible through its own dedicated service interface. It allows the same features as UCM like pausing or resuming of parallel transfers.

13.3 Packages

13.3.1 Software package

The unit of installation which is the input for the UCM is a Software Package.

The package includes, for example, one or several executables of (Adaptive) Applications, operating system or firmware updates, or updated configuration and calibration data that shall be deployed on the Adaptive Platform. This constitutes the Updatable Package part in Software Packages and contains the actual data to be added to or changed in the Adaptive Platform. Beside application and configuration data, each Software Package contains a Software Package Manifest providing metadata like the package name, version, dependencies and possible some vendor-specific information for processing the package.

The format of the Software Package is not specified, which enables using different kind of solutions for the implementation of UCM. Software Package consists of updates to be performed in software and metadata. This content is packaged by the UCM vendor tooling to generate a Software Package which will be processed by the targeted UCM.

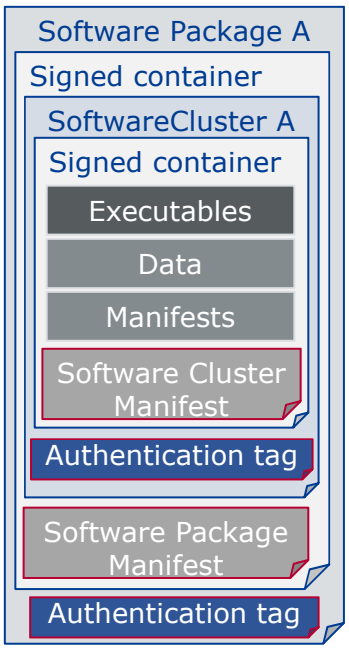

Figure 13.2: Overview Software Package

UCM processes the vendor-specific Software Package based on the provided meta-data. You can find below for information purpose a description of the fields that must be contained in a Software Package Manifest:

General information

- Package name: fully qualified short-name.
- Version: Version from Software Cluster model that has to follow https://semver.org semantic versioning specification with the exception that build number is mandatory for debugging/tracking purposes. Used primitive name is StrongRevisionLabelString
- deltaPackageApplicableVersion: Version of the Software Cluster to which this delta package can be applied
- Minimum supported UCM version: to make sure that the Software Package can be parsed properly by the UCM.
- Dependencies between Software Clusters: TPS Manifest Specification document contains a model describing dependencies between Software Cluster after it is updated or installed.

Sizes to allow checking if there is enough memory available:

- uncompressedSoftwareClusterSize: Size of Software Cluster in the targeted platform
- compressedSoftwareClusterSize: Size of Software Package

For information and tracking purpose

- Vendor: vendor id
- Vendor authentication tag
- Packager: vendor id
- Packager authentication tag: for package consistency check and security purposes (for UCM to check if the Software Package is trustable)
- Type approval: optional, homologation information. Could, for instance, be RXSWIN from UN ECE WP.29
- Release notes: description of this release changes
- License: for instance, MIT, GPL, BSD, proprietary.
- Estimated duration of operation: estimated duration including, transfer, processing and verification.

To distribute the package to the correct UCM within the vehicle:

- Diagnostic address: coming from the Software Cluster model, used in case package is coming from the tester via UDS for instance
- Action type: can be update, install, or remove
- Activation action: can be nothing, reboot (Machine) and restartApplication

13.3.2 Backend package

For an OEM backend to understand packages contents from several package suppliers, a backend package format is proposed as described in Figure 13.3.

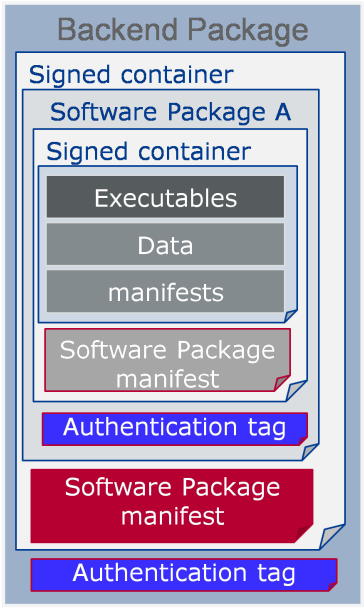

Figure 13.3: Overview Backend Package

The software package format is vendor-specific. However, as the backend package is meant to be vendor-independent, Software Package Manifest (in red in Figure 13.3) must use the ARXML file format.

13.3.3 Vehicle Package

A vehicle package is typically assembled by an OEM backend. It contains a collection of Software Package Manifests extracted from backend packages stored in the backend database. It also contains a Vehicle Package Manifest including a campaign orchestration and other fields needed for packages distribution by V-UCM within the vehicle (Figure 13.4).

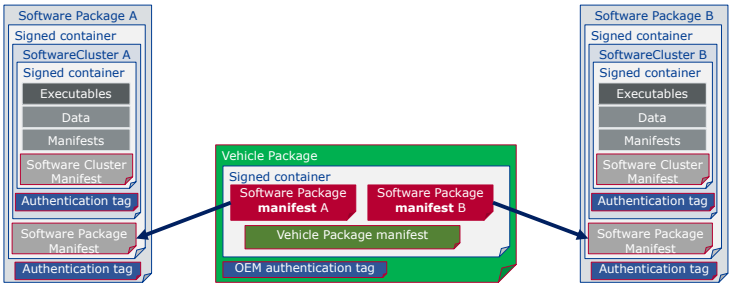

Figure 13.4: Overview Vehicle Package

You can find below for information purpose a description of the fields that should be contained in Vehicle Package Manifest:

- Repository: uri, repository or diagnostic address, for history, tracking and security purposes
- Minimum supported V-UCM version: to make sure that the Vehicle Package can be parsed properly by the V-UCM.
- For update campaign orchestration:
  - UCM identifier: unique identifier within vehicle architecture, to allow V-UCM identifying UCM subordinates in the vehicle
  - Associations of Software Packages to describe the sequence of transfer, processing, and activation
  - Vehicle driver notification: to interact with vehicle driver, asking for his consent or notifying him at several steps of the vehicle update of the optional safety measures to be taken during update.

The Vehicle Package could be used by a garage to fix a car having issues downloading an update for instance. Therefore, like backend Package, Vehicle Package Manifest shall be an ARXML file format for interoperability purposes.

13.3.4 Software release and packaging workflow

In order to create a backend package, an integrator has to use a packager compatible with the targeted UCM. This package could be provided by an Adaptive Platform stack vendor including the targeted UCM. After the integrator is assembling executable, Manifests, persistency, etc., he uses the packager to create a Software Package using UCM vendor-specific format. This same Software Package is then embedded into a backend Package along with ARXML Software Package Manifest. The Software Package could be signed by the packager or integrator and authentication tag included in Software Package. As backend Package might be transferred via the internet between an integrator and an OEM backend, both Software Package and Software Package Manifest should be signed into a container along with its authentication tag in order to avoid any Software Package Manifest modification.

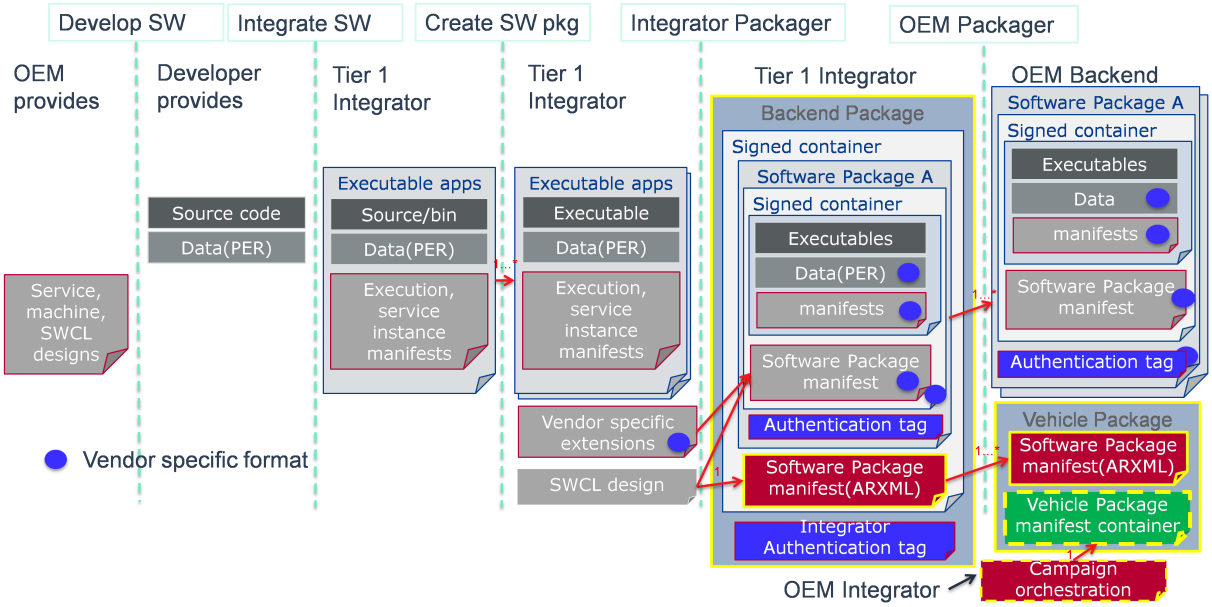

Figure 13.5: Packaging steps

Backend Packages assembled by integrator can then be put in the backend database or repository. When a vehicle needs an update or new installation, the backend server will query software packages from backend package database and associate the related Software Package Manifests into a Vehicle Package. In this package, backend server embeds a campaign orchestration selected based on the vehicle specific electronic architecture, deducted for instance from Vehicle Identifying Number.

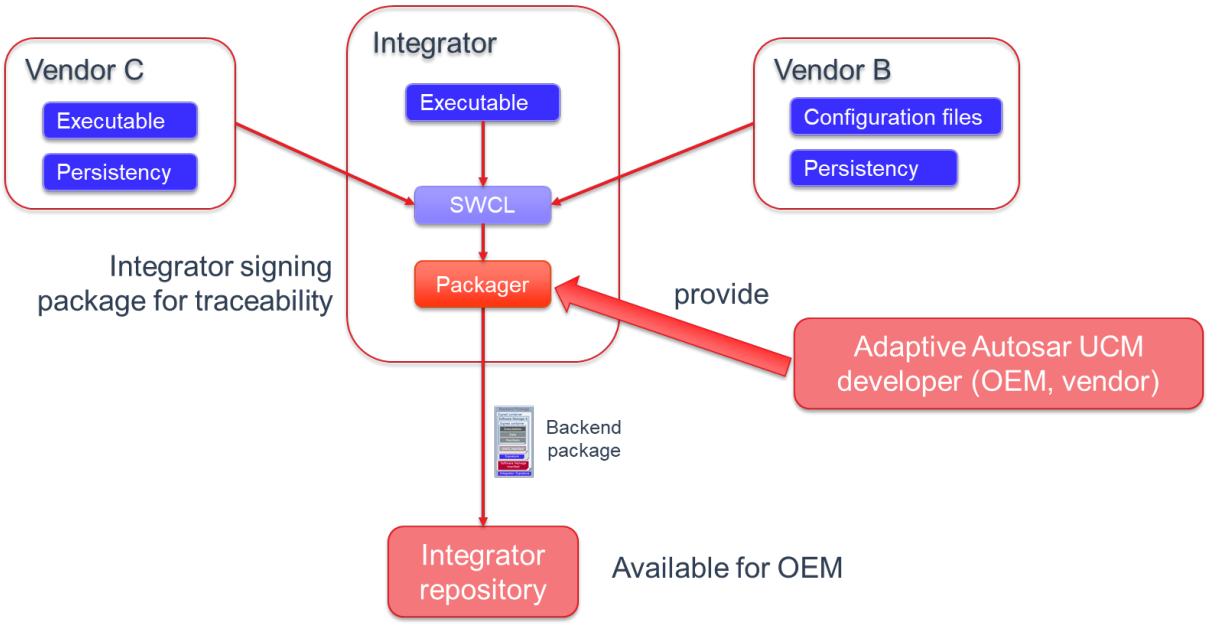

Figure 13.6: Packages distribution to vehicle

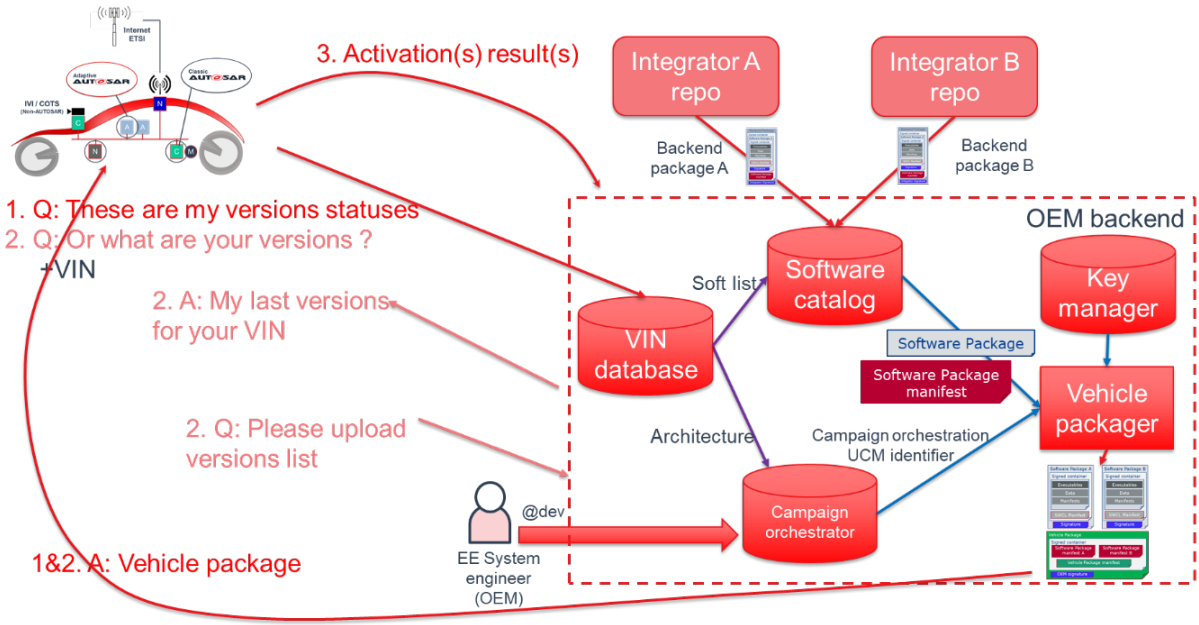

Figure 13.7: Packages distribution to vehicle

13.4 UCM processing and activating Software Packages

Install, update, and uninstall actions are performed through the ProcessSwPackage interface where UCM parses from metadata which actions need to be performed.

UCM sequence has been designed to support for example A/B update scenario or 'in-place' scenario where package manager provides the possibility to roll back into the previous version if this is needed.

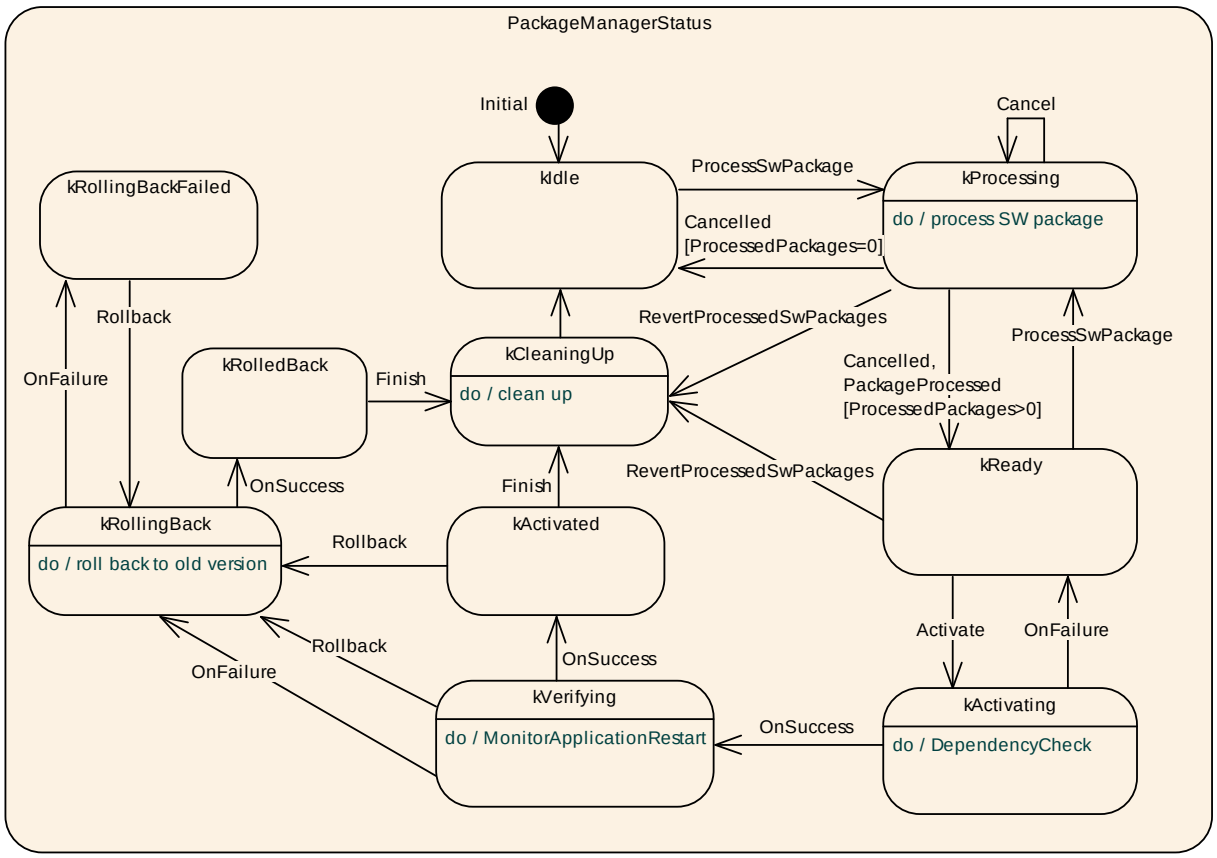

Figure 13.8: Overview of Processing and Activation of Software Package

To keep implementation simpler and more robust, only one client at a time can request to process a Software Package with the ProcessSwPackage method, switching UCM state to PROCESSING. Several clients can request to process transferred packages in sequence. In the case of A/B partition update scenario, several clients can process the inactive /B partition being updated; in case of software cluster cross dependencies, each client must update in sequence into "B partition". Once, processing is finished, UCM state switches to READY for activation or another processing.

Activation of changes with the Activate method is done for all processed packages regardless of the requesting client. V-UCM is coordinating this multi-client scenario. UCM might not know if all targeted Software Packages have been processed, but it shall perform a dependency check to see that system is consistent with the requirements of the installed software in "B partition". In case of dependencies are not fulfilled, UCM shall reject the activation and switch back to READY state.

When updates are being activated, UCM opens an UpdateSession at SM via ara::com. For each Function Group in each affected Software Cluster the PrepareUpdate method is called. It executes Function Group specific preparation steps. On success, the state changes to VERIFYING. UCM then requests either a machine reset or a Function

Group restart depending on the type of update via SM interface. For instance, if the update includes the operating system or functional cluster updates, UCM might want to reset the machine. However, if the update is only about a low criticality function, only restarting Function Groups could be enough, reducing annoyance to the driver. In this phase, UCM requests from SM to verify that targeted Function Groups are running properly. Once these restarts are finished successfully, UCM switches to ACTIVATED state.

When updates have been ACTIVATED, other processing requests will be rejected until activation has been resolved. In this phase, UCM Client or V-UCM can either call Finish for confirming the changes or Rollback for ignoring the changes and going back to the previous version of the software. This is intended for instance in case such update is part of a global update campaign coordinated by V-UCM, during which the update of another ECU has failed. After Finish is called, UCM cleans all unneeded resources and returns to IDLE.

In the case of Rollback is called, UCM is switched to the ROLLING-BACK state to re-activate the old versions of the software clusters by calling PrepareRollback method for each Function Group in each affected Software Cluster. For instance, in this state, in case of an A/B partition scenario, UCM will prepare the "A partition" to be reactivated/executed at the next restart. Then, when the restart takes place by calling the SM interface and the "A partition" is reactivated, UCM switches to the ROLLED-BACK state.

In both cases, Rollback and successful activation, UCM has to finish the update session at SM.

Processing while transferring is supported by UCM design in order to avoid storing Software Packages in Adaptive Platform, reducing costs and update time. For instance, in the case of Software Cluster containing only Adaptive application, UCM could decompress received blocks, place files to its target location, finally authenticate and check integrity of the Software Package.

13.5 V-UCM update campaign coordination

As V-UCM is coordinating several elements within the vehicle, its state machines are accessible from the CampaignState or TransferState fields, allowing to reduce V-UCM's API complexity. V-UCM is continuously discovering the UCM service instances in the vehicle using service discovery from ara::com.

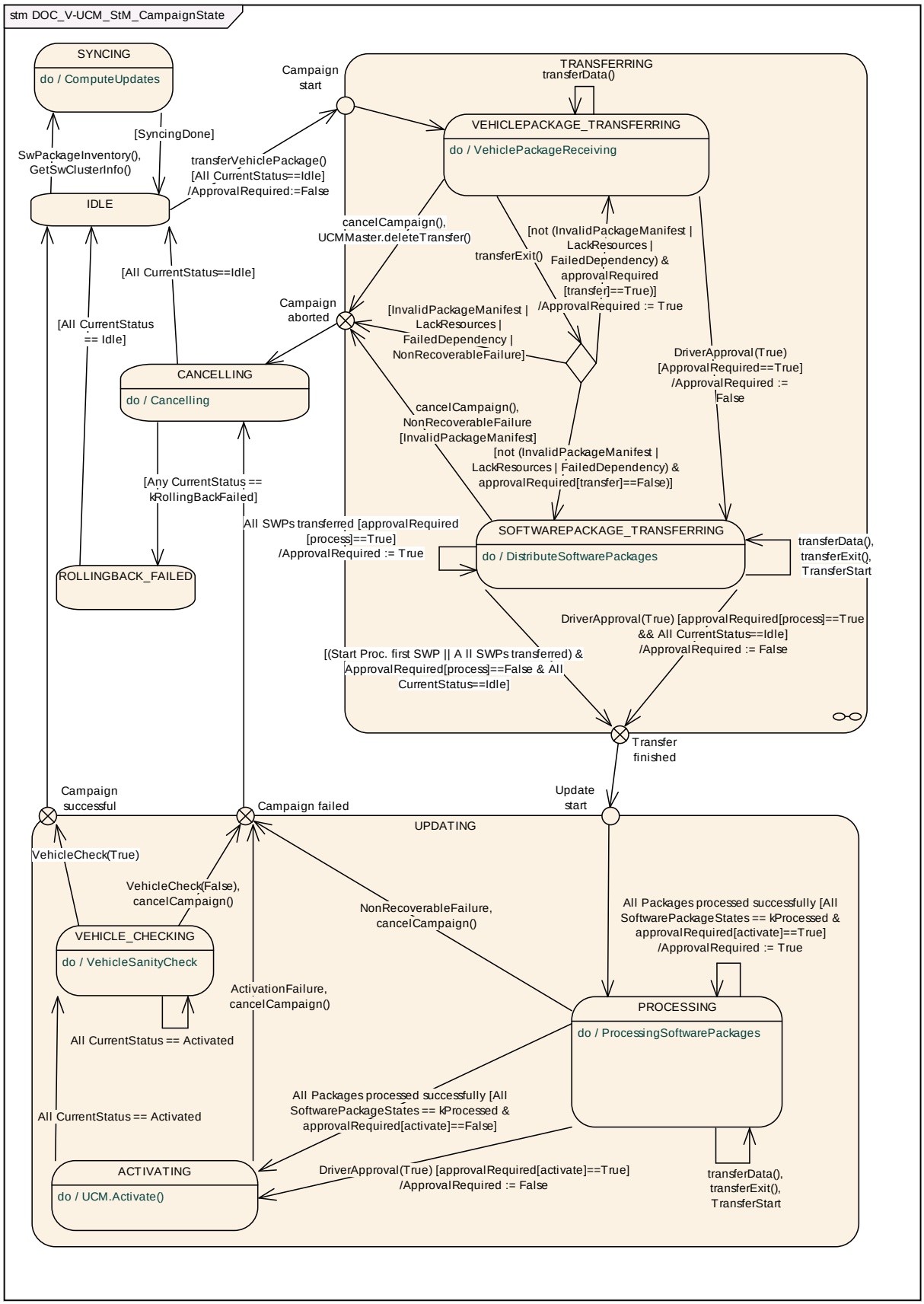

Figure 13.9: V-UCM state machine

The V-UCM state machine is not completely matching the UCM state machine as specific vehicle aspects have to be considered. For instance, the vehicle package transfer, synchronization of available software in vehicle and backend or vehicle integrity check after update, are specific to V-UCM.

13.5.1 Adaptive applications interacting with V-UCM

As vehicle update involves OEM specificities, OEM specific aspects are pushed by design into the Adaptive Application side. In order to have interoperability and exchangeability for those applications with several vendors platforms, the V-UCM interface is standardized as a Platform Service, like UCM. V-UCM assumes three applications to interact with itself, as described below.

13.5.1.1 OTA Client

OTA Client sets the communication channel between backend and V-UCM. The communication protocol between backend and OTA Client is not specified. OTA Client could include a scheduler regularly triggering synchronization of databases (managed by backend or V-UCM) containing available software from backend and present software in the vehicle. Updatable, installable or removable software are computed by the difference between these two in backend or V-UCM.

If a V-UCM is failing, it could be replaced by another one present in the vehicle. OTA Client should then include the decision mechanism to choose with which V-UCM to interact.

13.5.1.2 Vehicle driver

During an update, it could be necessary to interact with the vehicle human driver to:

- get consent to download (impacting data transfer costs), process or activate the software (safety measures acknowledgment)
- put the vehicle in a specific state (to guarantee safety during a critical update, it could be asked to stop vehicle and shutdown engine)

13.5.1.3 Vehicle state manager

Vehicle State Manager Adaptive Application is collecting states from all vehicle ECUs or Machines. From these collected states, Vehicle State Manager is computing a vehicle state based on the SafetyConditions field exposed by V-UCM, which is contained in the Vehicle Package. If the computed vehicle state is changing, the Vehicle State Manager

has to call V-UCM's method PublishSafetyState. If the update's safety is not met, the V-UCM can decide to postpone, pause or cancel an update.

13.5.1.4 Flashing Adapter

The Flashing Adapter is an Adaptive Application exposing same interface as UCM Subordinate to V-UCM but includes OEM specific sequences related to flashing via diagnostic. It uses an implementation of diagnostic protocol data unit application programming interface (D-PDU API following ISO22900) to communicate with Classic ECUs.

13.6 Software information reporting

UCM provides service interfaces that expose functionality to retrieve Adaptive Platform software information, such as names and versions of transferred packages, for processed but not committed software and for the last committed software. As the UCM update process has clear states, UCM provides information in which state is the processing of each Software Package.

V-UCM also provides service interfaces to expose Software information but at the vehicle level, aggregating information from several UCMs. This information is then exchanged with backend through OTA Client, for instance, to resolve what Software could be updated in the vehicle. Furthermore V-UCM provides a way to access the history of its actions like activation time and the result of processed packages. This history can be used by the backend to gather update campaign statistics from a fleet of cars or to troubleshoot issues at garage with a Diagnostic Tester.

13.7 Software update consistency and authentication

UCM and V-UCM shall authenticate their respective packages using an authentication tag covering the whole package as described in Figure 13.2 and Figure 13.4. The Adaptive platform shall provide necessary checksum algorithms, cryptographic signatures or other vendor and/or OEM specific mechanisms to validate the package, otherwise, an error will be returned by UCM or V-UCM. Practically, a package should be packaged by the tool coming from the same vendor as the one developing the targeted UCM or V-UCM in order to have authentication algorithm compatibility.

As authentication algorithms are using hashes, consistency is also checked when authenticating a package. Packages authentication and consistency could be checked at TransferData, TransferExit and ProcessSwPackages calls to cover many possible use cases and scenarios but shall be performed before any package is processed by UCM or V-UCM for maximum security.

13.8 Securing the update process

UCM and V-UCM provide services over ara::com. There is no authentication step of a client in both UCM and V-UCM update protocol. Instead, it is up to Identity and Access Management to ensure that the client requesting services over ara::com is legit.

13.9 Appropriate State Management during an update process

UCM is using the UpdateRequest service interface from State Management to request an update session that can be rejected due to state conflicts or safety considerations. It can also prepare FunctionGroups for an activation with PrepareUpdate method and verify the update, installation or remove with the VerifyUpdate method. If the verification is failing, UCM could request to change FunctionGroup states with rollback method. Reset of Machine can also be requested by UCM to SM if needed, otherwise a reparse of Manifests is necessary after activation to keep the platform's configuration consistent.

In case it is needed to perform Software Updates of more than one Machine running on the same ECU (virtualized or hierarchical environment), there is a possibility that the updates of each of those virtualized Machines would affect the state of other Machines on the same ECU, including but not limited to resetting Machine/ECU. As result, UCM Subordinates / Flashing Adapters responsible for updating SWCLs on those Machines would need to coordinate their update activities with a single supervising State Management instance running on one of the Machines on that ECU.

According to the description of "State Management in a virtualized/hierarchical environment" in AUTOSAR_AP_SWS_StateManagement [10], even though both Machines have their respective State Manager, one of them should take a role of supervising State Management instance to coordinate starting/stopping update sessions on each Machine as well as prevent unnecessary shutdowns during the activation (as result of power loss or machine/ecu reset).

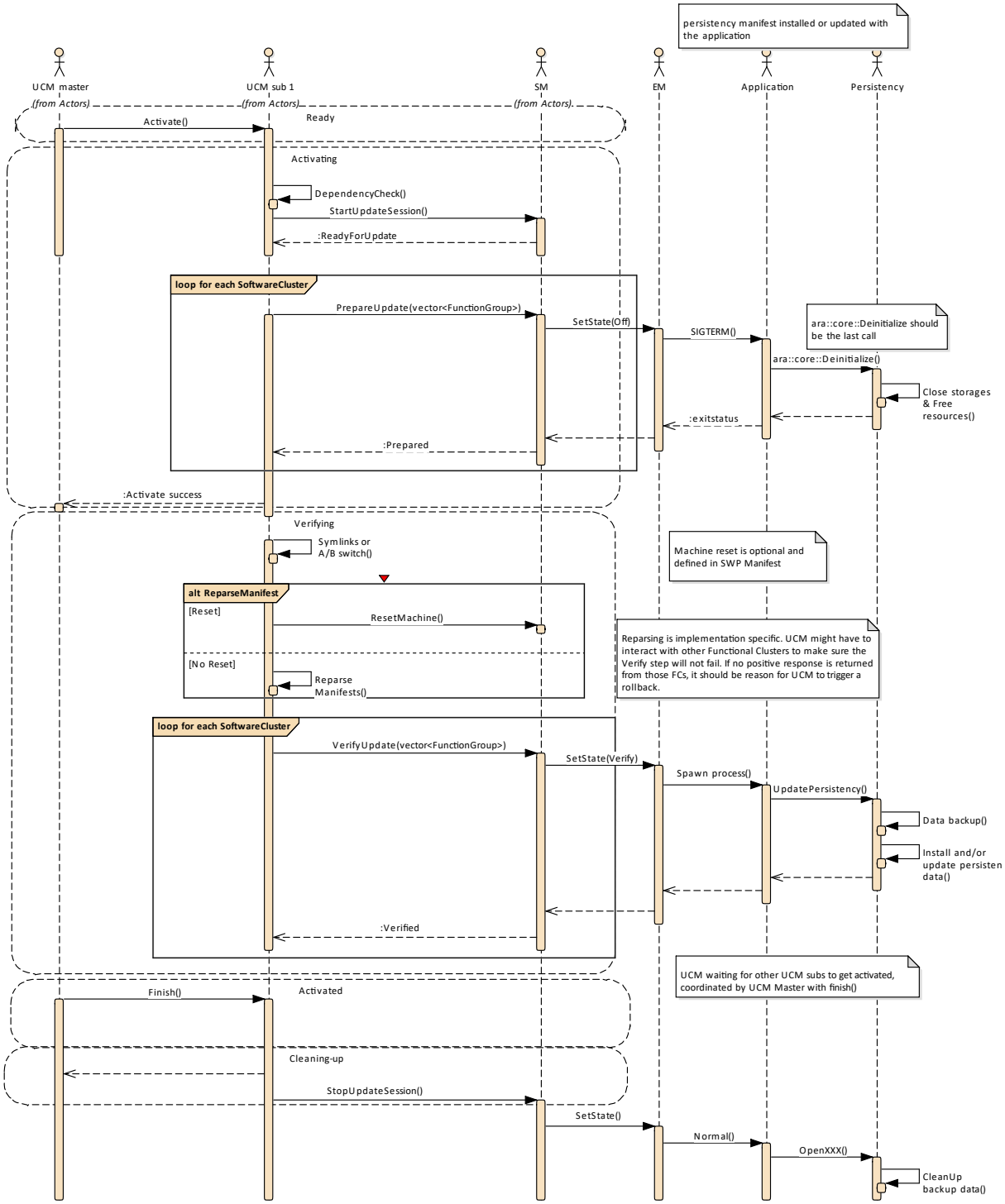

Figure 13.10: State Management during an update process

14 Identity and Access Management

The concept of Identity and Access Management (IAM) is driven by the increasing need for security, as the AUTOSAR Adaptive Platform needs a robust and well-defined trust relationship with its applications. IAM introduces privilege separation for Adaptive Applications and protection against privilege escalation in case of attacks. In addition, IAM enables integrators to verify access on resources requested by Adaptive Applications in advance during deployment. Identity and Access Management provides a framework for access control for requests from Adaptive Applications on Service Interfaces, Functional Clusters of the Adaptive Platform Foundation and related modeled resources.

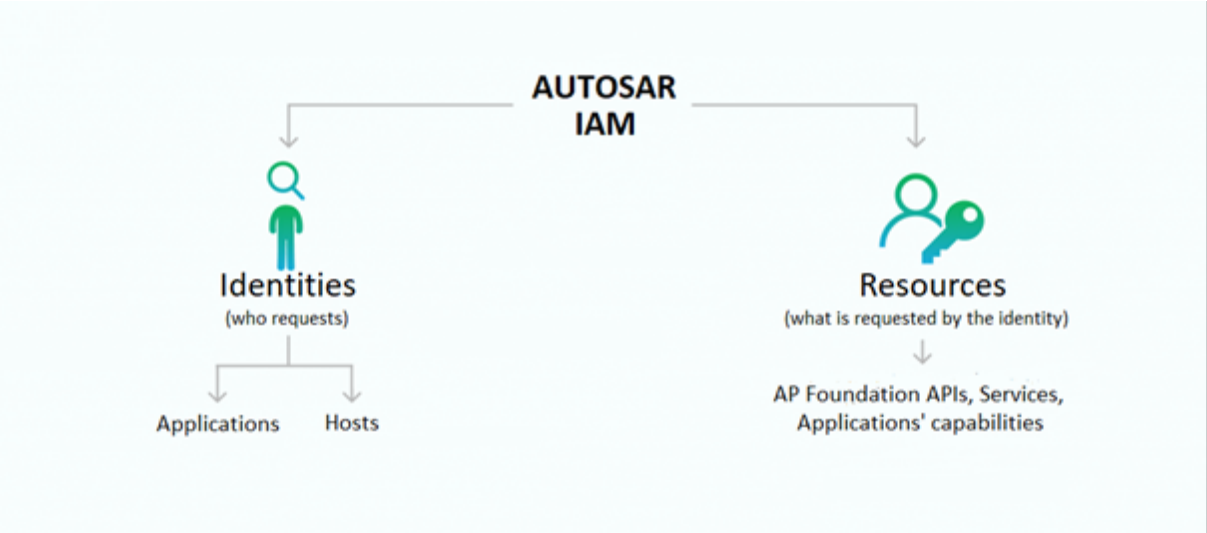

Figure 14.1: AUTOSAR IAM

IAM is broadly implemented by two logical entities. The PDP (Policy Decision Point) and PEP (Policy Enforcement Point) work together to verify if a subject can access a particular resource and then grant access to the said resource. The PDP represents the logic in which the access control decision is made. It determines if the application is allowed to perform the requested task. The PEP represents the logic in which the Access Control Decisions are enforced. It communicates directly with the associated PDP to receive the Access Control Decision.

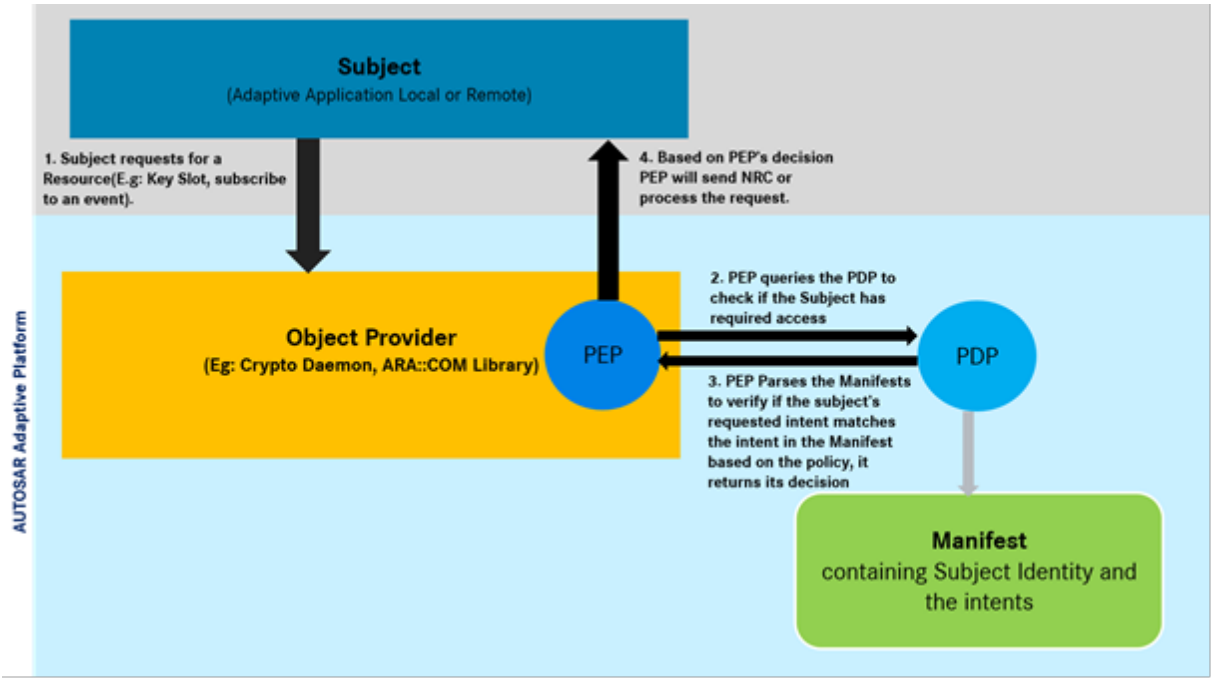

Figure 14.2: IAM sequence

A more detailed explanation of Identity and Access Management can be found in AUTOSAR_AP_EXP_IdentityAndAccessManagement [11].

15 Cryptography

AUTOSAR Adaptive Platform supports an API for common cryptographic operations and secure key management. The API supports the dynamic generation of keys and crypto jobs at runtime, as well as operating on data streams. To reduce storage requirements, keys may be stored internally in the crypto backend or externally and imported on demand.

The API is designed to support encapsulation of security-sensitive operations and decisions in a separate component, such as a Hardware Security Module (HSM). Additional protection of keys and key usage can be provided by constraining keys to particular usages (e.g., decrypt-only), or limiting the availability of keys to individual applications as reported by IAM.

Depending on application support, the API can also be used to protect session keys and intermediate secrets when processing cryptographic protocols such as TLS and SecOC.

The FC Crypto offers applications and other Adaptive AUTOSAR Functional Clusters

a standardized interface, which provides operations for cryptographic and related calculations. These operations include cryptographic operations, key management and certificate handling. FC Crypto handles the actual implementation of all operations, including all necessary configuration and brokering of operations between requesting application and stack-provided implementation. The standardized interface is exposed by the CryptoAPI.

X.509 Certificate Management Provider (CMP, namespace ara::crypto::x509) is responsible for X.509 certificates parsing, verification, authentic storage and local

searching by different attributes. In addition, CMP is responsible for storage, management, and processing of Certificate Revocation Lists (CRLs) and Delta CRLs. CMP supports of requests preparation and responses parsing for On-line Certificate Status Protocol (OCSP).

15.1 Security Architecture

While AUTOSAR AP only defines the high-level Crypto Stack API exposed to applications, this API is defined with a security architecture in mind that was designed to meet above security and functional requirements.

The general architecture is depicted in Figure 15-1. On the highest layer, AUTOSAR AP, as well as native and hybrid applications, link against the AUTOSAR AP Crypto Stack API. The API implementation may refer to a central unit (Crypto Service Manager) to implement platform-level tasks such as access control and certificate storage consistently across applications. The implementation may also use the Crypto Service Manager to coordinate the offloading of functionality to a Crypto Driver, such as a Hardware Security Module (HSM). Indeed, the offloading functionality of the Crypto Stack

API this way is expected to be a typical implementation strategy: The Crypto Driver may implement the complete set of key management and crypto functions in order to accelerate crypto operations and shield managed keys from malicious applications.

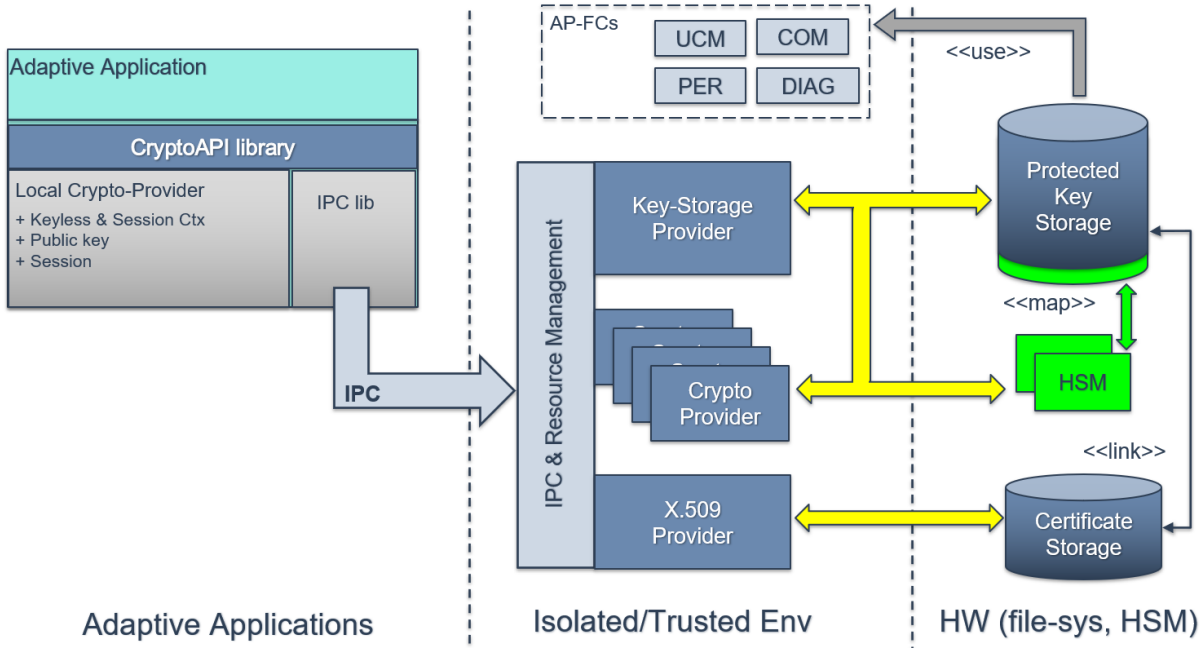

Figure 15.1: Crypto Stack - Reference Architecture

In order to realize this layered security architecture, the Crypto Stack API does not only perform typical crypto operations like encryption and decryption but also provides native support for:

1. Operating with encrypted keys or key handles
2. Managing keys securely despite possible application compromise
3. Constraining application access to and allowed operations on keys

15.2 Key Management Architecture

To support the secure remote management of keys despite potential application compromise, the Crypto Stack integrates a key management architecture where keys and associated data are managed in end-to-end protected form. Keys can be introduced into the system either in a trusted fashion, based on an existing provisioning key, or in an untrusted fashion via local key generation. Assuming an appropriately secured crypto backend/driver, applications are unable to modify keys except via well-defined, authorized requests such as key update or revocation.

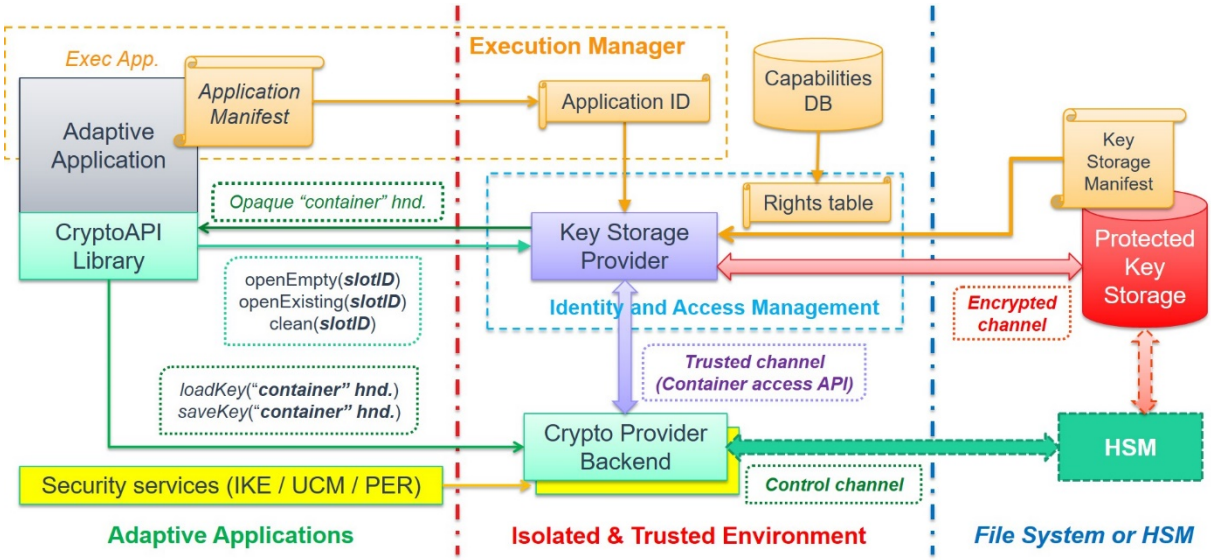

Figure 15.2: CKI Key Management Interactions

15.3 Remarks on API Extension

Significant new usages and interactions that require the introduction of new or modified permission/policy validation logic should be tied to corresponding new key usage policy flags. For example, alternative provisioning keys with different ownership/permission checks can be introduced by adding a corresponding new key usage policy and enforcing the new logic in all key management operations involving those new keys.

16 Log and Trace

16.1 Overview

The Log and Trace Functional Cluster is responsible for managing and instrumenting the logging features of the AUTOSAR Adaptive Platform. The logging and tracing features can be used by the platform during development as well as in and after production. These two use cases differ, and the Log and Trace component allows flexible instrumentation and configuration of logging in order to cover the full spectrum. The logging information can be forwarded to multiple sinks, depending on the configuration, such as the communication bus, a file on the system and a serial console. The provided logging information is marked with severity levels and the Log and Trace component can be instrumented to log the information only above a certain severity level, this enables complex filtering and straightforward fault detection of issues on the logging client side. For each severity level, a separate method is provided to be used by Adaptive applications or by Functional Clusters.

The AUTOSAR Adaptive Platform and the logging Functional Cluster are responsible for maintaining the platform stability to not overload the system resources.

Log and Trace relies on the LT protocol standardized within the AUTOSAR consortium. The protocol ensures that the logging information is packed into a standardized delivery and presentation format. Furthermore, the LT protocol can add additional information to the logging messages, such as an ECU ID. This information can be used by a logging client to relate, sort or filter the received logging frames.

In addition, utility methods are provided, e.g. to convert decimal values into the hexadecimal numeral system or into the binary numeral system. These are necessary to enable applications to provide data to Log and Trace which conforms to the standardized serialization format of the LT protocol.

The Log And Trace Functional Cluster also offers two principal "classes" of log messages: *Modeled* and *Non-Modeled* messages. Both these support adding one or more "arguments" to a log message. A log message without any arguments serves no purpose and is discarded.

Non-Modeled messages are the traditional way of composing log messages: All arguments of the message are added to an internal message buffer and then eventually serialized for output, either to a console/file, or via network. All parts of the messages will be sent via network. In the DLT protocol, these messages are called "verbose" messages.

Modeled messages are designed to reduce traffic on the network, by omitting certain static (i.e. unchanging) parts of a message from the network. As the name suggests, these parts are instead added to the application ARXML model. In the DLT protocol, these messages are called "non-verbose" messages. A log message viewer application is able to display the full message by combining the static parts from the model with the dynamic parts from the received message.

Non-modeled messages are mainly used during development, as the information required for the modeled messages may not be available at that time. However, nonmodeled messages can impose a high load on the network, making modeled messages usually the preferred choice in production systems.

16.2 Architecture

The Log and Trace interfaces are provided in the namespace ara::log for applications to forward logging onto one of the aforementioned logging sinks.

The Log and Trace interfaces rely on the back-end implementation that is a part of the Logging framework. The Logging framework can use other Functional Clusters to fulfill certain features, such as Time Synchronization.

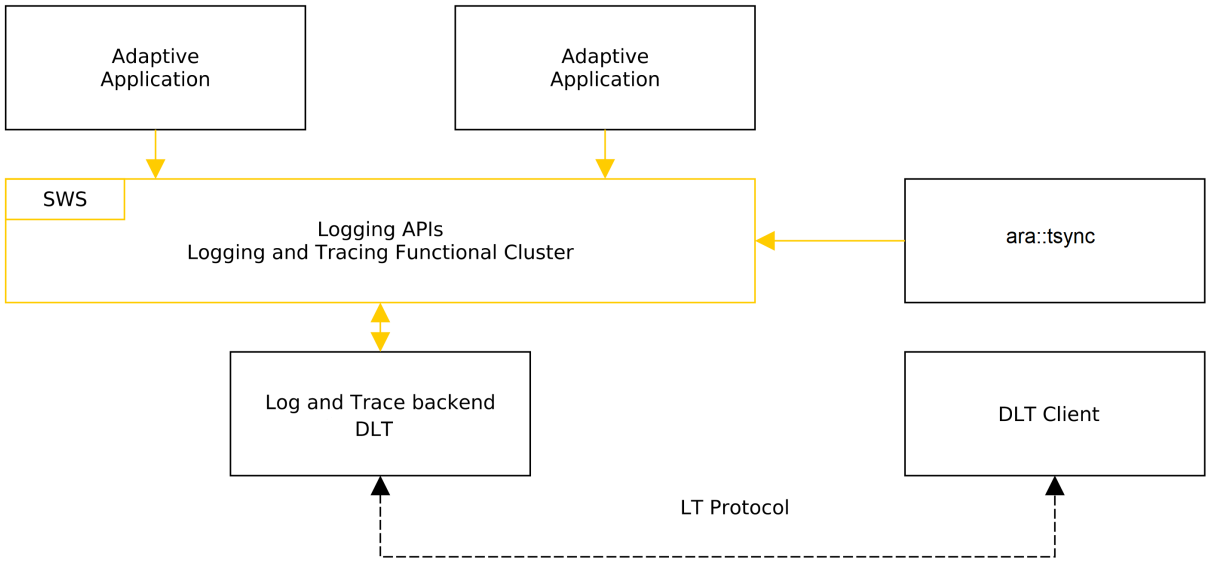

Figure 16.1: Overview Log and Trace

The `ara::log` Functional Cluster defines a "Builder"-pattern inspired set of APIs for constructing non-modeled messages and a single member function `ara::log::Logger::Log` for sending modeled messages.

Unlike the non-modeled message APIs, it represents a single-call interface, i.e. a single function call passes all arguments to the `Logger` instance and performs all necessary actions to generate and send the message. This has the advantage that the runtime cost for a modeled message that is eventually not being output (because the messages log level does not reach the configured log level threshold) can be made very small: after parameter passing and function call, a single `if` clause checks the log level threshold and immediately returns if the threshold is not reached. This contrasts with the non-modeled message APIs, where multiple function calls are performed for constructing a message object, even if that is then eventually discarded.

17 Safety

17.1 Functional Safety Architecture

AUTOSAR provides a safety overview and safety requirements for the Adaptive Platform to support the integration of the Adaptive Platform in safety projects. For this release, the safety overview is presented in the form of an explanatory document (AUTOSAR_EXP_SafetyOverview [12]) and safety requirements in the form of requirements document (RS_Safety [13]).

These documents shall help functional safety engineers to identify functional safety-related topics within the AUTOSAR Adaptive Platform. The following list provides a general guidance on how to map contents in RS_Safety [13] and AUTOSAR_EXP_SafetyOverview [12] to the contents and structures in ISO 26262 [14]:

- AUTOSAR Adaptive Platform assumptions, objectives, scenarios and use-cases (AUTOSAR_EXP_SafetyOverview [12])
- System definition, system context and fault considerations (AUTOSAR_EXP_SafetyOverview [12])
- Hazard analysis (AUTOSAR_EXP_SafetyOverview [12])
- Safety Goals (AUTOSAR_EXP_SafetyOverview [12])
- Functional safety concept and functional safety requirements (RS_Safety [13])
- Technical safety requirements (RS_Safety [13])

The objective of the safety overview document (AUTOSAR_EXP_SafetyOverview [12]) is to state top-level safety goals and assumed use-cases or scenarios. The explanatory document contains assumptions, exemplary items such as reference models, and/or references to exemplary technical solutions, devices, processes or software. Any such assumptions or exemplary items contained in this document are for illustration purposes only. These assumptions are not part of the AUTOSAR standard.

The requirement specification (RS_Safety [13]) elaborates the high-level safety requirements written in RS_Main [2]. It makes use of the intended functionality described in this document. The functional safety requirements are derived from the safety goals and hazards mentioned in EXP_SafetyOverview [12]. Technical safety requirements towards the AUTOSAR functional cluster and safety relevant applications are derived from the functional safety requirements.

The following content is scheduled for later releases:

- Technical safety concept and technical safety requirements
- Validation of safety requirements, safety analysis, and exemplary use-cases

As a final remark, the use of the AUTOSAR Adaptive Platform does not imply ISO 26262 [14] compliance. It is still possible to build unsafe systems using the AUTOSAR

Adaptive Platform safety measures and mechanisms. The architecture of the AUTOSAR Adaptive Platform can, in the best case, only be considered to be a Safety Element out-of Context (SEooC).

17.2 Protection of Information Exchange (E2E-Protection)

Protection of information exchange is supported within AUTOSAR by several E2E profiles for different use cases. The provided functionality gives the possibility to check

- if information has changed during the transmission.
- if messages have been lost or repeated during the transmission.
- the reliability of the communication channel by the E2E state machine.

E2E protection within AUTOSAR will be supported to allow safe communication between all combinations of AUTOSAR AP and CP instances, whether they are in the same or different ECUs. Where useful, mechanisms will be provided to allow safe communication using more capabilities of the service-oriented approaches within the Adaptive Platform.

E2E protection detects communication faults as described in ISO 26262 [14] but with limitations see document AUTOSAR_PRS_E2EProtocol [15].

E2E checks for communication faults and provides results of the checks but does not trigger further reactions to communication faults. Acknowledgment of transmission and transmission security is not provided in the E2E context.

For this release E2E supports:

- Events (with limitation, see AP SWS Communication Management)
- Methods (with limitation, see AP SWS Communication Management)
- Fields (with limitation, see AP SWS Communication Management)

The following use cases are not supported:

- Events in callout mode
- Non-periodic events
- Methods (without constraints)

The profiles that can be used for E2E protection and diagrams for message flow are described in (AUTOSAR_PRS_E2EProtocol [15] and AUTOSAR_SWS_CommunicationManagement [16]).

17.3 Platform Health Management

The Platform Health Management supervises the execution of software. It offers the following supervision functionalities (all supervision functions can be invoked independently):

- Alive supervision
- Deadline supervision
- Logical supervision
- Health Channel Supervision
- Inform State Manager about supervision failures
- Trigger watchdog

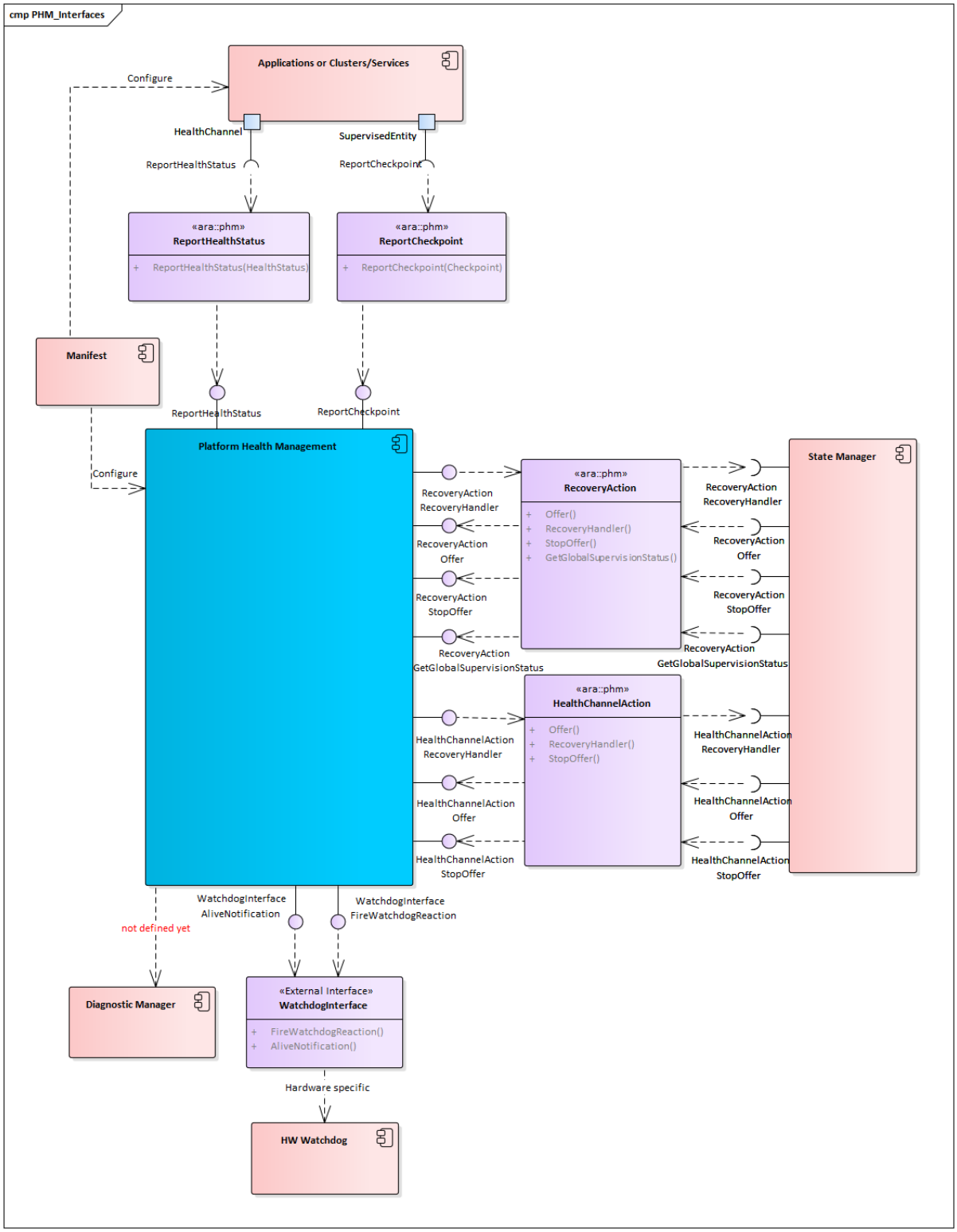

Figure 17.1: PHM interface with other components

Note: `Health Channels` (HealthChannelExternalStatus) are currently under discussion. The architectural question to which Functional Cluster this feature belongs to is expected to be resolved for the next release.

Alive Supervision checks that a supervised entity is not running too frequently and not too rarely.

Deadline supervision checks that steps in a supervised entity are executed in a time that is within the configured minimum and maximum limits.

Logical supervision checks that the control flow during execution matches the designed control flow.

Alive, Deadline and Logical Supervision are performed based on reporting of checkpoints by applications/non-platform services or functional clusters via API ReportCheckpoint.

Health channel supervision provides the possibility to hook external supervision results (like RAM test, voltage monitoring, etc.) to the Platform Health Management.

Health channel supervision is performed based on reporting of Health statuses via API ReportHealthStatus.

Platform Health Management notifies State Manager if a failure is detected in the supervised entities via API RecoveryHandler of RecoveryAction.

In case a failure in Execution Management or State Management is detected, Platform Health Management will trigger a reset through watchdog.

Known limitations for this release:

- Dependency on the Diagnostic Manager is not yet defined.

Functionality shared by CP and AP is described in the foundation documents and named "Health Monitoring" (RS_HealthMonitoring [17] and ASWS_HealthMonitoring [18]). Additional specifications for AP only are described in the AP documents and named "Platform Health Management" (RS_PlatformHealthManagement [19], SWS_PlatformHealthManagement [20]).

Note that the architectural elements EM, SM and PHM are highly safety-relevant; safe execution management and safe health monitoring are fundamental to the safe operation of an Adaptive Application. The EM, PHM, SM elements are inter-dependent and coordinate their activities to ensure functional safety within the AUTOSAR Adaptive Platform.

17.4 System Health Monitoring

System Health Monitoring (SHM) introduces platform agnostic health monitoring. SHM focuses on system wide coordination of error handling across multiple platforms on multiple controllers and machines.

The `System Health Monitor` component can be instantiated either as a master instance or a client instance. SHM client is responsible for communicating platform level health data to the master instance whereas SHM master is responsible for de-

termination of `Health Indicators`. The `Health Indicators` can be determined at subsystem level, feature level, domain level and eventually at vehicle level. These `Health Indicators` can be used either for platform level recovery actions or to enhance the services with a Health of Service parameter, similar to Quality of Service.

`System Health Monitor` requirements are described in the foundation document (RS_HealthMonitoring [17]. Abstract specifications of `System Health Monitor` interfaces and HealthIndicator format are described in ASWS_HealthMonitoring [18]).

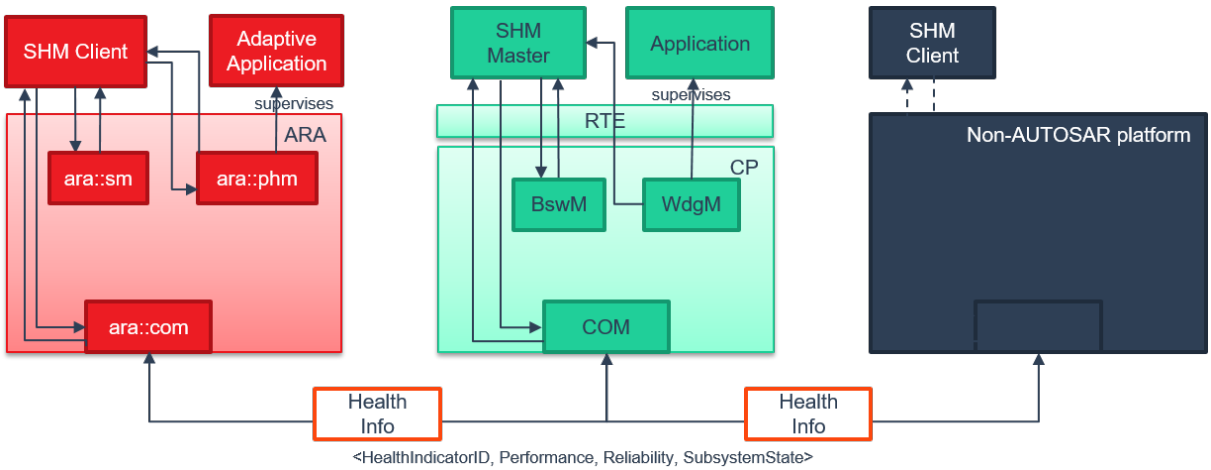

Figure 17.2: SHM overview

18 Core Types

Core Types defines common classes and functionality used by multiple Functional Clusters as part of their public interfaces. One of the rationale to define Core Types was to include common complex data types that are often used in the interface definition.

18.1 Error Handling

18.1.1 Overview

Handling errors is a crucial topic for any software development. For safety-critical software, it is even more important, because lives can depend on it. However, current standards for the development of safety-critical software impose significant restrictions on the build toolchain, especially with regard to C++ exceptions. For ASIL applications, using C++ exceptions is usually not possible due to the lack of exceptions support with ASIL-certified C++ compilers.

The Adaptive Platform introduces a concept that enables error handling without C++ exceptions and defines a number of C++ data types to aid in this.

From an application programmer's point of view, the central types implementing this concept are `ara::core::ErrorCode` and `ara::core::Result`.

18.1.2 ErrorCode

An instance of `ara::core::ErrorCode` represents a specific error condition within a software. It is similar to `std::error_code`, but differs in significant aspects from it.

An `ErrorCode` always contains an enumeration value (type-erased into an integral type) and a reference to an error domain. The enumeration value describes the specific type of error, and the error domain reference defines the context where that error is applicable. Additional optional members are a user-defined message string and a vendor-defined supplementary error description value.

Within the Adaptive Platform, each Functional Cluster defines one or more error domains. For instance, the Functional Cluster "Core Types" defines two error domains "Core" and "Future", which contain error codes for different sets of error conditions.

18.1.3 Result

Class `ara::core::Result` is a wrapper type that either contains a value or an error. Due to its templated nature, both value and error can be of any type. However, the error type is defaulted to `ara::core::ErrorCode`, and it is expected that this assignment is kept throughout the Adaptive Platform.

Because the error type has a default, most declarations of `ara::core::Result` only need to give the type of the value, e.g. `ara::core::Result<int>` for a Result type that contains either an int or an `ara::core::ErrorCode`.

The contained value or error can be accessed via the member functions `Result::Value` or `Result::Error`. It is the caller's responsibility to ensure that these access functions are called only if the Result instance contains a value or an error, respectively. The type of the content of a Result, i.e. a value or an error, can be queried by `Result::HasValue`. None of these member functions throw any exceptions and thus can be used in environments that do not support C++ exceptions.

In addition to the exception-less workflow described above, the class `ara::core::Result` allows to convert a contained `ara::core::ErrorCode` object into a C++ exception, by calling `ara::core::Result::ValueOrThrow`. This call returns any contained value as-is, but treats a contained error by throwing the corresponding exception type, which is automatically derived from the contents of the contained `ara::core::ErrorCode`.

18.1.4 Future and Promise

Similar to the way `ara::core::Result` is used as a generalized return type for synchronous function calls, `ara::core::Future` is used as a generalized return type for asynchronous function calls.

`ara::core::Future` is closely modeled on `std::future`, but has been extended to interoperate with `ara::core::Result`.

Similar to `ara::core::Result`, `ara::core::Future` is a class that either contains a value or an error. This content can be extracted in two ways:

1. by calling `ara::core::Future::get`, which returns the contained value, if it exists, or throws an exception otherwise
2. by calling `ara::core::Future::GetResult`, which returns a `ara::core::Result` object which contains the value or the error from the Future

Both of these calls will block until the value or error has been made available by the asynchronous function call.

18.2 Advanced data types

In addition to the error-handling data types mentioned in the previous section, the Adaptive Platform also contains a number of other data types and helper functions.

Some of these types are already contained in the C++11 standard; however, types with almost identical behavior are re-defined within the `ara::core` namespace. The reason

for this is that the memory allocation behavior of the `std::` types is often unsuitable for automotive purposes. Thus, the `ara::core` ones define their own memory allocation behavior.

Examples of such data types are `Vector`, `Map`, and `String`.

Other types defined in `ara::core` have been defined in or proposed for a newer C++ standard, and the Adaptive Platform includes them into the ara::core namespace, because they are necessary for supporting certain constructs of the Manifest, or because they are deemed very useful to use in an API.

Examples of such data types are `StringView`, `Span`, `Optional`, and `Variant`.

18.3 Primitive data types

Another document, [8], exists, which defines primitive types that can be used in ServiceInterface descriptions. This document may be considered to be merged with Core Types document in the future.

18.4 Global initialization and shutdown functions

The following functions are available to initialize and de-initialize the AUTOSAR Runtime for Adaptive Application. This includes, for example, data structures and threads.

- `ara::core::Initialize`
- `ara::core::Deinitialize`

`ara::core::Initialize` initializes e.g., data structures and threads of the AUTOSAR Adaptive Runtime for Applications. Prior to this call, no interaction with the ARA is possible, except for very few that are explicitly defined in [SWS_CORE_15002] because they are required for the initialization and de-initialization itself. The call to `ara::core::Initialize()` must be made inside of `main()`, i.e., in a place where it is guaranteed that static memory initialization has completed.

Depending on the individual functional cluster specification, after the call to `Initialize()`, the calling application may have to provide additional configuration data (e.g., set an Application ID for Logging) or make additional initialization calls (e.g., start a FindService in ara::com) before other API calls to the respective functional cluster can be made. Such calls must be made after the call to `Initialize()`. Calls to ARA APIs made before static initialization has completed lead to undefined behavior. Calls made after static initialization has completed but before `Initialize()` was called will lead to implementation-defined behavior or a Violation depending on the concrete call.

`ara::core::Deinitialize` destroys used resources e.g., all data structures and threads of the AUTOSAR Adaptive Runtime for Applications. After this call, no in-

teraction with the ARA is possible, except for very few that are explicitly defined in [SWS_CORE_15002] because they are required for the initialization and de-initialization itself. The call to `ara::core::Deinitialize()` must be made inside of `main()`, i.e., in a place where it is guaranteed that the static initialization has completed and destruction of statically initialized data has not yet started. Calls made to ARA APIs after a call to `ara::core::Deinitialize()` but before destruction of statically initialized data will lead to implementation-defined behavior. Calls made to ARA APIs after the destruction of statically initialized data will lead to undefined behavior.

19 Intrusion Detection System Manager

The IdsM is part of the AUTOSAR Intrusion Detection System (IDS).

The functional cluster Intrusion Detection System Manager (IdsM) provides a standardized interface for receiving notifications of security events (SEv). The SEvs can be reported by security sensors implemented in other functional clusters and adaptive applications. Additionally, the SEvs can be reported with optional context data such as event type and suspicious data, which can be useful information for the security forensic performed at the back end. Besides collecting, the IdsM has the capability of qualifying SEvs according to configurable rules. The IdsM filters and transforms reported SEvs to qualified onboard security events (QSEv). The QSEv is further handled by the IdsM for storage or forwarding. Depending on the overall security concept, QSEv can be persisted locally on the ECU or propagated towards the Ids Reporter Module (IdsR), which might pass the QSEv data to a security operation center (SOC) in the back end.

Note: IDSR and SOC are not part of standardization work in AUTOSAR. SEM will follow in one of the next releases in accordance with the IDSM for Classic Platform.

The following picture from the concept folder demonstrates the IDS architecture overview.

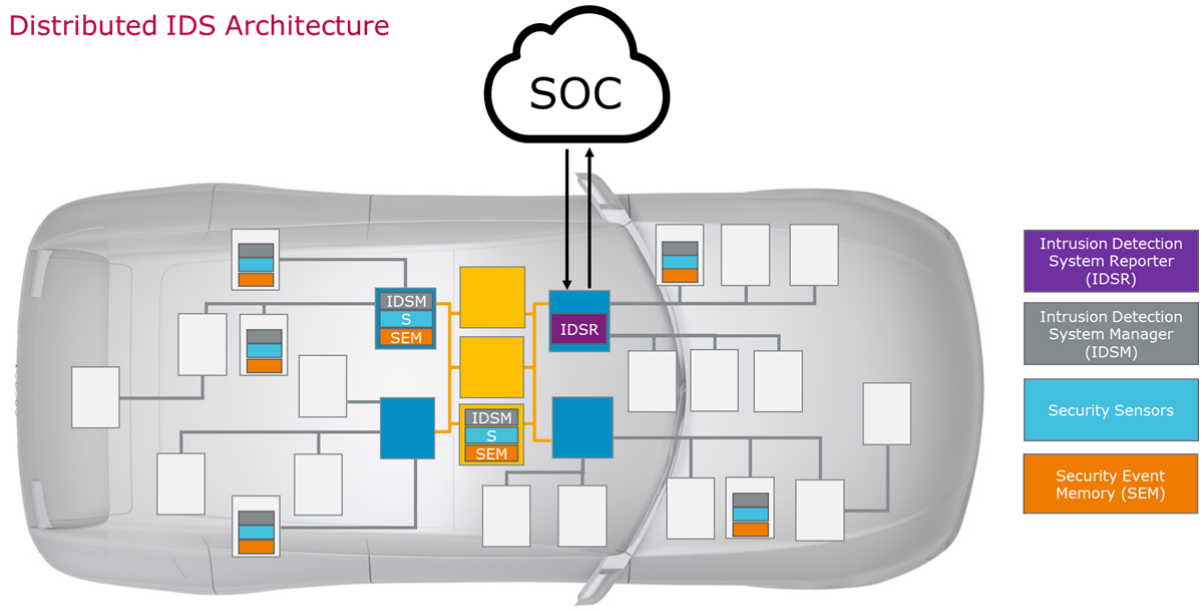

Figure 19.1: Distributed IDS Architecture

20 Firewall

AP supports filtering of Ethernet traffic based on a pattern-matching algorithm based on firewall rules. The functional cluster Firewall is responsible for managing these rules and configuring the firewall engine.

For every AP instance where the firewall is deployed, a set of firewall rules can be configured. Every firewall rule consists of a pattern against which the network packets are matched and an action to be carried out by the firewall in case of a match (i.e., allow or block the network packet). The firewall iterates over this list of firewall rules and performs the associated action in case of a pattern match. If no firewall rule matches, a default action is carried out.

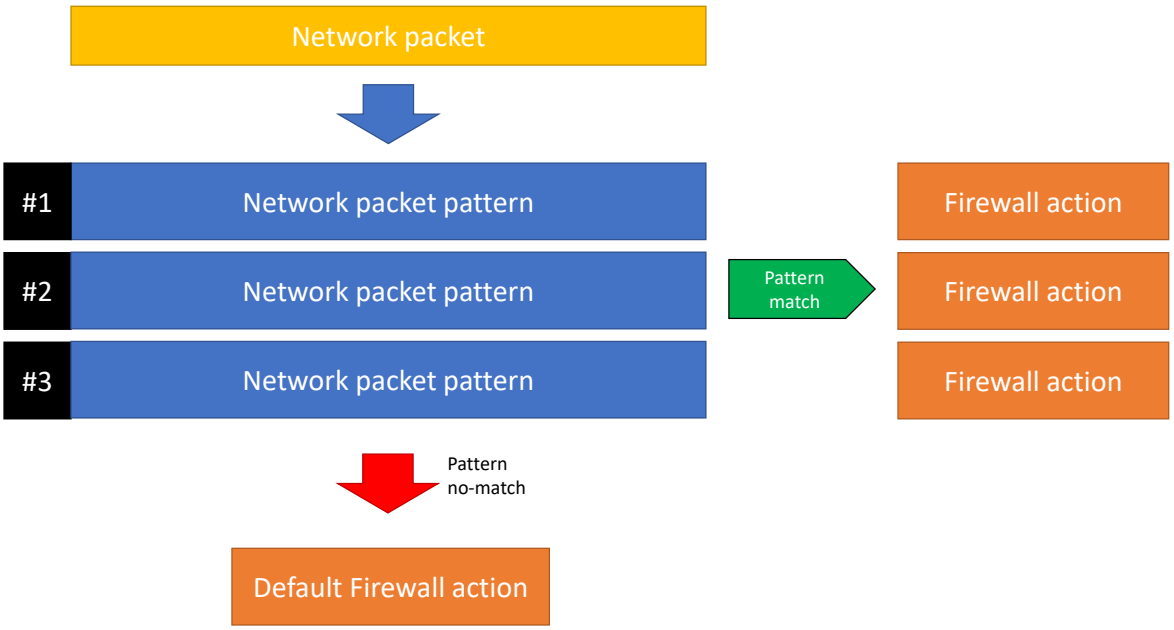

Figure 20.1: Firewall pattern matching algorithm

The firewall rule format is specified within the AUTOSAR MetaModel, hence the rules can be configured and deployed to the AP instances by means of the Machine Manifest. The firewall supports filtering within three categories: stateless network inspection, stateful network inspection (focusing on TCP) and deep packet inspection of application layer protocols.

The network packet inspection is carried out by a firewall engine, which is typically located at a low layer (TCP/IP stack or below). AP specifies a Functional Cluster Firewall that acts as a management cluster: it takes the firewall rules configuration from the manifest and configures the underlying firewall engine accordingly.

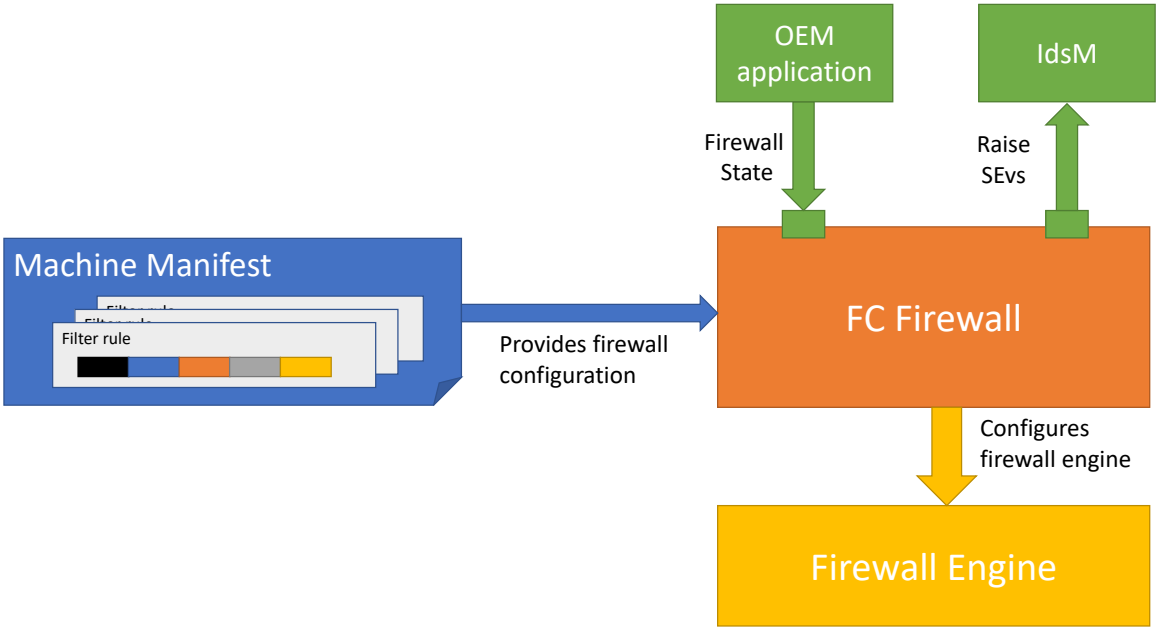

Figure 20.2: Firewall architecture

Additionally, the Firewall supports the following use-cases:

- **State-dependent filtering:** It is possible to define OEM-specific firewall states, where the network traffic is expected to be different (e.g. since the vehicle is currently driving, parked or in a diagnostic session). Firewall rules can be associated with a firewall state and only the rules that are associated with the currently active firewall state are used to inspect and filter network packets. The current firewall state can be set by a user-application by using an API exposed by the FC Firewall.
- **Security Events:** The FC Firewall supports the Intrusion Detection System by specifying a set of security events that can be raised in case of blocked messages. The security events are reported to the AUTOSAR IdsM module, which takes care of event qualification and further handling of the events (passing them to a SOC or storing them persistently on the ECU).

21 Raw Data Stream

Adaptive AUTOSAR also provides a standalone Communication API for processing raw binary data streams towards an external ECU, e.g. a sensor in an ADAS system. The API is static and implements functionality for a client application to establish a communication channel to a server, and for a server application to wait for incoming connections from a client. The API provides functionality for both clients and servers to read and write raw data (a stream of bytes) over the communication channel, and to shutdown the established communication channel. The Raw Data Stream channels can be configured by an integrator by applying deployment information, containing e.g. network endpoint information and selected protocols. Currently, TCP/IP sockets shall be used as a transport layer, but other alternatives can be added in the future. The Raw Data Stream interface is available in the namespace `ara::rds`.