Document Title	Explanation of ara::com API
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	846

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• Replaced term "(Un)Checked Exception" by proper formulations • Clarified the usage and transferation of "Instance Specifier" • Removed the reference to "AUTOSAR_RS_CPP14Guidelines"
2019-11-28	R19-11	AUTOSAR Release Management	• Added access to current field value from Get/SetHandler • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Changed explanation of Event reception due to new ara::com API
2018-10-31	18-10	AUTOSAR Release Management	• Added InstanceIdentifier and InstanceSpecifier explanation • Restructured chapter structure • Adapt FindService signatures • Added sample code for event usage • Restructured chapter structure • Proxy and skeleton instances are not copyable • Changed certain data types to ara::core namespace. • Adapted to new error handling based on ara::core::ErrorCode

2018-03-29	18-03	AUTOSAR Release Management	• Added Fire&Forget Methods • Minor changes and bugfixes
2017-10-27	17-10	AUTOSAR Release Management	• Added explanation of TLV • Minor changes and bugfixes
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

References

[1] Specification of RTE Software
AUTOSAR_SWS_RTE

[2] Middleware for Real-time and Embedded Systems
http://doi.acm.org/10.1145/508448.508472

[3] Patterns, Frameworks, and Middleware: Their Synergistic Relationships
http://dl.acm.org/citation.cfm?id=776816.776917

[4] Specification of the Adaptive Core
AUTOSAR_SWS_AdaptiveCore

[5] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[6] SOME/IP Protocol Specification
AUTOSAR_PRS_SOMEIPProtocol

[7] Serialization and Unserialization
https://isocpp.org/wiki/faq/serialization

[8] Copying and Comparing: Problems and Solutions
http://dx.doi.org/10.1007/3-540-45102-1_11

[9] SOME/IP Service Discovery Protocol Specification
AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol

1 Preface

Typically, reading formal specifications isn't the easiest way to learn and understand a certain technology. This especially holds true for the Communication Management API (`ara::com`) in the AUTOSAR Adaptive Platform.

Therefore this document shall serve as an entry point not only for the developer of software components for the Adaptive Platform, who will use the `ara::com` API to interact with other application or service components, but also for Adaptive Platform product vendors, who are going to implement an optimized IPC binding for the `ara::com` API on their platform.

We strongly encourage both groups of readers to read this document at hand before going into the formal details of the related SWS.

Since we do address two different groups, it is obvious that parts of the content is more intended for the user of the API (application software developer), while other parts are rather intended for the IPC binding implementer (Adaptive Platform product vendor).

We address this by explicitly marking explanations, which are intended for the IPC binding implementer. So our basic assumption is, that everything which is of interest to the user of the API is also informative/relevant for the IPC binding implementer, while parts explicitly marked as "detailed information for the IPC binding implementer" like this:

AUTOSAR Binding Implementer Hint
Some very detailed technical information

Table 1.1: AUTOSAR Binding Implementer Hint - introduction

are no mandatory knowledge for the user for `ara::com` API. Nevertheless, the interested API user might also benefit from these more detailed explanations, as it will help him to get a good understanding of architectural implications.

2 Introduction

Why did AUTOSAR invent yet another communication middleware API/technology, while there are dozens on the market — the more so as one of the guidelines of Adaptive Platform was to reuse existing and field proven technology?

Before coming up with a new middleware design, we did evaluate existing technologies, which — at first glance — seemed to be valid candidates. Among those were:

- ROS API
- DDS API
- CommonAPI (GENIVI)
- DADDY API (Bosch)

The final decision to come up with a new and AUTOSAR-specific Communication Management API was made due to the fact, that not all of our key requirements were met by existing solutions:

- We need a Communication Management, which is NOT bound to a concrete network communication protocol. It has to support the SOME/IP protocol but there has to be flexibility to exchange that.
- The AUTOSAR service model, which defines services as a collection of provided methods, events and fields shall be supported naturally/straight forward.
- The API shall support an event-driven and a polling model to get access to communicated data equally well. The latter one is typically needed by real-time applications to avoid unnecessary context switches, while the former one is much more convenient for applications without real-time requirements.
- Possibility for seamless integration of end-to-end protection to fulfill ASIL requirements.
- Support for static (preconfigured) and dynamic (runtime) selection of service instances to communicate with.

So in the final `ara::com` API specification, the reader will find concepts (which we will describe in-depth in the upcoming chapters), which might be familiar for him from technologies, we have evaluated or even from the existing Classic Platform:

- Proxy (or Stub)/Skeleton approach (CORBA, Ice, CommonAPI, Java RMI, ...)
- Protocol independent API (CommonAPI, Java RMI)
- Queued communication with configurable receiver-side caches (DDS, DADDY, Classic Platform)
- Zero-copy capable API with possibility to shift memory management to the middleware (DADDY)

- Data reception filtering (DDS, DADDY)

Now that we have established the introduction of a new middleware API, we go into the details of the API in the following chapters.

The following statement is the basis for basically all AUTOSAR AP specifications, but should be explicitly pointed out here again:

`ara::com` only defines the API signatures and its behavior visible to the application developer. Providing an implementation of those APIs and the underlying middleware transport layer is the responsibility of the AUTOSAR AP vendor.

For a rough parallel with the AUTOSAR Classic Platform, `ara::com` can be seen as fulfilling functional requirements in the Adaptive Platform similar to those covered in the Classic Platform by the RTE APIs [1] such as `Rte_Write`, `Rte_Read`, `Rte_Send`, `Rte_Receive`, `Rte_Call`, `Rte_Result`.

3 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the explanation of ara::com API.

Abbreviation / Acronym:	Description:
ctor	C++ constructor
dtor	C++ destructor
IPC	Inter Process Communication
RT	Realtime
SI	Service Interface
WET	Worst Case Execution Time

Terms:	Description:
Binding	This typically describes the realization of some abstract concept with a specific implementation or technology. In AUTOSAR, for instance, we have an abstract data type and interface model described in the methodology. Mapping it to a concrete programming language is called *language binding*. In the AUTOSAR Adaptive Platform for instance we do have a C++ language binding. In this explanatory document we typically use the tech term *binding* to refer to the implementation of the abstract (technology independent) ara::com API to a concrete communication transport technology like for instance sockets, pipes, shared memory, ...
Callable	In the context of C++ a Callable is defined as: A Callable type is a type for which the INVOKE operation (used by, e.g., std::function, std::bind, and std::thread::thread) is applicable. This operation may be performed explicitly using the library function std::invoke. (since C++17)

4 API Design Visions and Guidelines

One goal of the API design was to have it as lean as possible. Meaning, that it should only provide the minimal set of functionality needed to support the service based communication paradigm consisting of the basic mechanisms: methods, events and fields.

Our definition of the notion "as lean as possible" in this context means: Essentially the API shall only deal with the functionality to handle method, field and event communication on service consumer and service provider implementation side.

If we decided to provide a bit more than just that, then the reason generally was *"If solving a certain communication-related problem ABOVE our API could not be done efficiently, we provide the solution as part of* `ara::com` *API layer."*

Consequently, `ara::com` does not provide any kind of component model or framework, which would take care of things like component life cycle, management of program flow or simply setting up `ara::com` API objects according to the formal component description of the respective application.

All this could be easily built on top of the basic `ara::com` API and needs not be standardized to support typical collaboration models.

During the design phase of the API we constantly challenged each part of our drafts, whether it would allow for efficient IPC implementations from AP vendors, since we were aware, that you could easily break it already on the API abstraction level, making it hard or almost impossible to implement a well performing binding.

One of the central design points was — as already stated in the introduction — to support polling and event-driven programming paradigms equally well.

So you will see in the later chapters, that the application developer, when using `ara::com` is free to chose the approach, which fits best to his application design, independent whether he implements the service consumer or service provider side of a communication relation.

This allows for support of strictly real-time scheduled applications, where the application requires total control of what (amount) is done when and where unnecessary context switches are most critical.

On the other hand the more relaxed event based applications, which simply want to get notified whenever the communication layer has data available for them is also fully supported.

The decision within AUTOSAR to genuinely support C++11/C++14 for AP was a very good fit for the `ara::com` API design.

For enhanced usability, comfort and a breeze of elegance `ara::com` API exploits C++ features like smart pointers, template functions and classes, proven concepts for asynchronous operations and reasonable operator overloading.

5 High Level API Structure

5.1 Proxy/Skeleton Architecture

If you've ever had contact with middleware technology from a programmer's perspective, then the approach of a Proxy/Skeleton architecture might be well known to you.

Looking at the number of middleware technologies using the Proxy/Skeleton (sometimes even called Stub/Skeleton) paradigm, it is reasonable to call it the "classic approach".

So with `ara::com` we also decided to use this classical Proxy/Skeleton architectural pattern and also name it accordingly.

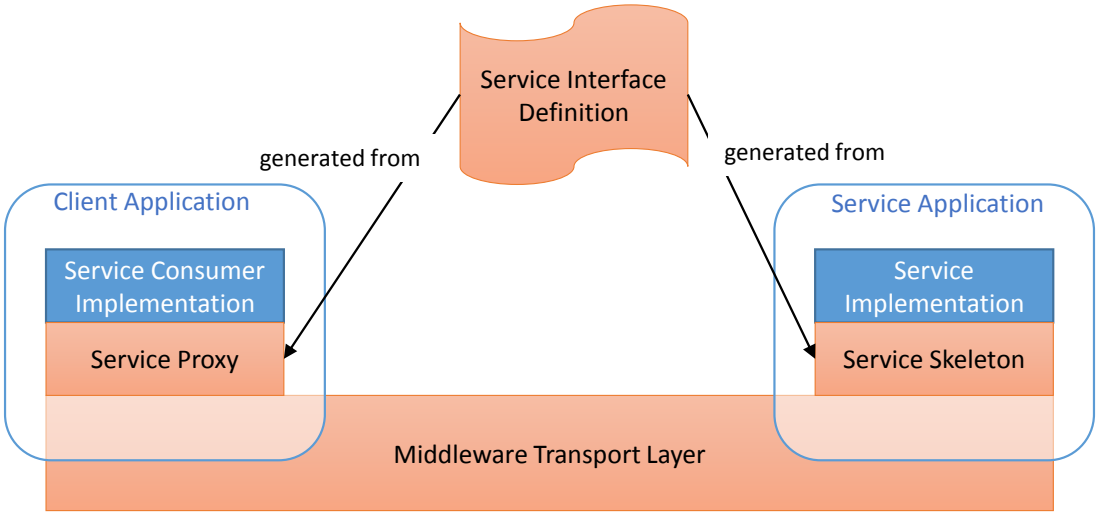

Figure 5.1: Proxy Skeleton Pattern

The basic idea of this pattern is, that from a formal service definition two code artifacts are generated:

- Service Proxy: This code is - from the perspective of the service consumer, which wants to use a possibly remote service - the facade that represents this service on code level.

  In an object-oriented language binding, this typically is an instance of a generated class, which provides methods for all functionalities the service provides. So the service consumer side application code interacts with this local facade, which then knows how to propagate these calls to the remote service implementation and back.

- Service Skeleton: This code is - from the perspective of the service implementation, which provides functionalities according to the service definition - the code, which allows to connect the service implementation to the Communication Man-

agement transport layer, so that the service implementation can be contacted by distributed service consumers.

In an object-oriented language binding, this typically is an instance of a generated class. Usually the service implementation from the application developer is connected with this generated class via a subclass relationship.

So the service side application code interacts with this middleware adapter either by implementing abstract methods of the generated class or by calling methods of that generated class.

Further details regarding the structure of `ara::com` Proxies and Skeletons are shown in section section 6.2 and section 6.3. Regarding this design pattern in general and its role in middleware implementations, see [2] and [3].

5.2 Runtime Interface

Beside the APIs provided by proxies and skeletons, the `ara::com` API contains functionality, which is about crosscutting concerns and therefore cannot really be assigned to proxy/skeleton domain.

The approach in `ara::com` is to assign this kind of functionality to a Runtime singleton class (see 6.4).

5.3 Data Type Abstractions

`ara::com` API introduces specific data types, which are used throughout its various interfaces. They can roughly be divided into the following classes:

- Pointer types: for pointers to data transmitted via middleware
- Collection types: for collections of data transmitted via middleware
- Types for async operation result management: `ara::com` relies on AUTOSAR AP specific data types (see [4]), which are specific versions of C++ `std::future`/`std::promise`
- Function wrappers: for various application side callback or handler functions to be called by the middleware

`ara::com` defines signature and expected behavior of those types, but does not provide an implementation. The idea of this approach is, that platform vendors could easily come up with their own optimized implementation of those types.

This is obvious for collection and pointer types as one of the major jobs of an IPC implementation has to deal with memory allocation for the data which is exchanged between middleware users.

Being able to provide their own implementations allows to optimize for their chosen memory model.

For most of the types `ara::com` provides a default mapping to existing C++ types in ara/com/types.h. This default mapping decision could be reused by an AP product vendor.

The default mapping provided by `ara::com` even has a real benefit for a product vendor, who wants to implement its own variant: He can validate the functional behavior of his own implementation against the implementation of the default mapping.

5.4 Error Notification

`ara::com` API follows the concepts of error handling described in chapter "Error handling" in [4]. `Checked Error`s will be returned via `ara::core::ErrorCode` directly or an `ara::core::ErrorCode` embedded into a `ara::core::Result`, which either holds a valid return value or the `ara::core::ErrorCode`.

The functionality provided with `ara::core::Result` and `ara::core::Future` (details, see subsubsection 6.2.4.2) allow the user of `ara::com` to chose between exception based or return code based error handling to some degree.

5.4.1 Checked Errors/Exceptions

`Checked Error`s within `ara::com` API can only occur in the context of a call of a service interface method and is therefore fully covered in subsection 6.2.4 and subsection 6.3.4.

5.4.2 Unchecked Errors/Exceptions

`Unchecked Error`s within `ara::com` API can occur in the context of **any** `ara::com` API call.
The `ara::com` API does not throw any `Execption`. The only way to have exceptions is calling the get method of `ara::core::Future`, if the user decides to use this approach.

6 API Elements

The following subchapters will guide through the different API elements, which `ara::com` defines. Since we will give code examples for various artifacts and provide sample code how to use those APIs from a developer perspective, it is a good idea to have some uniformity in our examples.

So we will use a virtual service (interface) called "RadarService". The following is a kind of a semi-formal description, which should give you an impression of what this "RadarService" provides/does and might be easier to read than a formal AUTOSAR ARXML service description:

```
RadarService {
  // types used within service
  type RadarObjects {
    active : bool
    objects : array {
      elementtype: uint8
      size: variable
    }
  }

  type Position {
    x: uint32
    y: uint32
    z: uint32
  }

  // events provided by service
  event BrakeEvent {
    type:RadarObjects
  }

  // fields provided by service
  field UpdateRate {
    type:uint32
    get: true
    set: true
  }

  error CalibrationFailed {
    errorCode : 1
    errorContext {
      failureText : string
    }
  }

  error InvalidConfigString {
    errorCode : 2
    errorContext {
      invalidConfig : string
      currentValidConfig : string
    }
  }
```

```
43
44   // methods provided by service
45   method Calibrate {
46     param configuration {
47       type: string
48       direction: in
49     }
50     param result {
51       type: bool
52       direction: out
53     }
54     raises {
55       CalibrationFailed
56       InvalidConfigString
57     }
58   }
59
60   method Adjust {
61     param target_position {
62       type: Position
63       direction: in
64     }
65     param success {
66       type: bool
67       direction: out
68     }
69     param effective_position {
70       type: Position
71       direction: out
72     }
73   }
74
75   oneway method LogCurrentState {}
76 }
```

Listing 6.1: RadarService Definition

So the example service `RadarService` provides an event “BrakeEvent”, which consists of a structure containing a flag and a variable length array of uint8 (as extra payload).

Then it provides a field “UpdateRate”, which is of uint32 type and supports get and set calls and finally it provides three methods.

Method “Adjust”, to position the radar, which contains a target position as in-parameter and two out-parameters. One to signal the success of the positioning and one to report the final (maybe deviating) effective position.

The method “Calibrate” to calibrate the radar, getting an configuration string as in-parameter and returning a success indicator as out-parameter. This method may raise two different application errors, in case the calibration failed: “CalibrationFailed” and “InvalidConfigString”.

The method “LogCurrentState” is a one way method, which means, that no feedback is returned to the caller, if the method is executed at all and with which outcome. It instructs the service `RadarService` to output its current state into its local log files.

6.1 Instance Identifiers

Instance identifiers, which get used at proxy and as well at skeleton side, are such a central concept, that their explanation is drawn here — before the detailed description of `ara::com` proxies and skeletons in upcoming chapters.

Instance identifiers are used within `ara::com`, on client/proxy side, when a specific instance of a service shall be searched for or — at the server/skeleton side — when a specific instance of a service is created.

At `ara::com` API level the instance identifier is generally a technical binding specific identifier.

Therefore the concrete content/structure of which such an instance identifier consists, is totally technology specific: So f.i. SOME/IP is using 16 bit unsigned integer identifiers to distinguish different instances of the same service type, while DDS (DDS-RPC) uses *string<256>* as `service_instance_name`.

Independant of the binding technology the abstract facade of any concrete instance identifier shall apply to this signature at `ara::com` API level in `namespace ara::com`:

```
class InstanceIdentifier {
  public:

    explicit InstanceIdentifier(const ara::core:string_view value);
    const ara::core:string_view toString() const;
    bool operator==(const InstanceIdentifier& other) const;
    bool operator<(const InstanceIdentifier& other) const;
    InstanceIdentifier& operator=(const InstanceIdentifier& other);
};
```

Listing 6.2: InstanceIdentifier class

As you can see the instance identifier interface `ara::com::InstanceIdentifier` provides a `ctor` taking a string, which means it can be constructed from a string representation and it does provide a toString() method, which allows to get a stringified representation of the technology specific `ara::com::InstanceIdentifier`.

This pair of `ctor` taking a string representation and the possibility to write out the string representation makes the `ara::com::InstanceIdentifier` "serializable". This allows it to be transferred, persisted, later re-used, ... (more on potential use cases later).

Introspection into this string (trying to interpret the content) makes no sense for the user of `ara::com`. As mentioned: The content will be highly middleware product/binding specific!

Since it is a core feature, that the technical binding used by an `ara::com` based application is defined/specified by the integrator during deployment any expectations from an `ara::com` software developer regarding its content/structure are typically invalid. Logging it/tracing it out to a log channel might be helpful for debug analysis however.

Then, where does the software-developer get such a highly binding specific `ara::com::InstanceIdentifier` to be used in `ara::com` API calls?

The answer is: By an `ara:com` provided functionality, which translates a logical local name used typically by the software developer in his realm into the technology/binding specific `ara::com::InstanceIdentifier`. This indirection masters both challenges:

- developer using `ara::com` does not need to know anything about bindings and their specifics
- Integrators can adapt bindings in deployments

The local name from which the `ara::com::InstanceIdentifier` is constructed comes basically from AUTOSAR meta-model, describing your software component model.

The requirement for this local name — we will call it "instance specifier" from now on — is, that it is unambiguous within an executable. It has basically the form:

```
<context 0>/<context 1>/.../<context N>/<port name>
```

The C++ representation of such an "instance specifier" is the class `ara::core::InstanceSpecifier`. Structurally it looks similar to the `ara::com::InstanceIdentifier`:

```
1  class InstanceSpecifier {
2    public:
3      // ctor to build specifier from AUTOSAR short name identifier
4      // with '/' as separator between package names
5      explicit InstanceSpecifier(const ara::core:string_view value);
6      const ara::core:string_view toString() const;
7      bool operator==(const InstanceSpecifier& other) const;
8      bool operator<(const InstanceSpecifier& other) const;
9      InstanceSpecifier& operator=(const InstanceSpecifier& other);
10 };
```

Listing 6.3: InstanceSpecifier class

If the unambiguousness is ensured, the integrator/deployer can assign a dedicated technical binding with its specific instance IDs to those "instance specifier" via a "manifest file", which is specifically used for a distinct instantiation/execution of the executable.

This explicitly allows, to start the same executable N times, each time with a different manifest, which maps the same `ara::core::InstanceSpecifier` differently.

Details about the `ara::com` relation to the meta-model and the nature of nested contexts can be read more detailed in section 9.4.

The API `ara:com` provides the following function, to do the translation from the `ara::core::InstanceSpecifier` (local name in the software developers realm) to the technical `ara::com::InstanceIdentifier`:

```
namespace ara {
namespace com {
namespace runtime {
ara::com::InstanceIdentifierContainer ara::com::runtime::ResolveInstanceIDs
    (ara::core::InstanceSpecifier modelName);
}
}
}
```

Listing 6.4: InstanceSpecifier Resolution

Why this API does return an `InstanceIdentifierContainer`, which represents a collection of `ara::com::InstanceIdentifier` is in need of explanation: AUTOSAR supports, that the integrator may configure multiple technical bindings behind one abstract identifier visible to the software component developer.

This feature is called multi-binding and referred to at different parts in this document (you find a more detailed explanation in section 9.3).

Using multi-binding on the skeleton/server side is a common use case, since it simply allows different clients to use their preferred binding, when contacting the server.

Contrary using multi-binding on the proxy/client side is a rather exotic one. E.g. it could be used to support some fail-over approaches (if binding A does not work, fall back on binding B).

So the possible returns for a call of `ResolveInstanceIDs()` are:

- empty list: The integrator failed to provide a mapping for the abstract identifier. This most likely is a configuration error.
- list with one element: The common case. Mapping to one concrete instance id of one concrete technical binding.
- list with more than one element: Mapping to multiple technical instances with possibly multiple technical bindings.

Technically the middleware implementation of `ResolveInstanceIDs()` does a lookup of the `ara::core::InstanceSpecifier` from the `service instance manifest` bundled within the process.

Therefore the `ara::core::InstanceSpecifier` must be unambiguous within the bundled `service instance manifest`.

6.1.1 When to use InstanceIdentifier versus InstanceSpecifier

According to the previous explanations, the impression may have arisen that a software developer always has to resolve `ara::core::InstanceSpecifier` to `ara::com::InstanceIdentifier` manually (by a call to `ResolveInstanceIDs()`) first, before using `ara::com` APIs, which need instance identifier information.

This would be indeed a bit awkward as we already mentioned, that the "typical" approach for a software developer, which implements an Adaptive AUTOSAR SWC, is to use abstract "instance specifiers" from the realm of the software component model.

As you will see in the upcoming chapters, which detail the APIs on the proxy and skeleton side, `ara::com` provides typically function overloads, which either take `ara::com::InstanceIdentifier` OR `ara::core::InstanceSpecifier`, freeing the developer in the most common use cases, where he simply uses `ara::core::InstanceSpecifier` from explicitly calling `ResolveInstanceIDs()`.

This means, that the direct use of `ara::com::InstanceIdentifier` and manual resolution of `ara::core::InstanceSpecifier` is intended more for power users with rather specific/exotic use cases. Some examples will be given in the chapters, where the corresponding `ara::com` API overrides at proxy/skeleton side are discussed.

The fundamental difference between the two variants is this: An `ara::com::InstanceIdentifier` can be exchanged more easily between Adaptive Applications/processes!

As they already exactly contain all the technology specific information and do not need any further resolution via content of a `service instance manifest` such a serialized `ara::com::InstanceIdentifier` can be reconstructed within a different process and be used as long as his process has access to the same binding technology the `ara::com::InstanceIdentifier` is based upon.

6.1.1.1 Transfer of an InstanceIdentifier

As discussed before the `ara::com::InstanceIdentifier` should only be used for "power users" since its format is stack vendor dependent and it contains technology binding information. The transfer or the storage of an `ara::com::InstanceIdentifier` may be very risky, therefore. As the transfer binding may not exist anymore after the transfer or re-storing or the `ara::com::InstanceIdentifier` of stack vendor A may be interpreted by an application using the stack of vendor B.

6.2 Proxy Class

The Proxy class is generated from the service interface description of the AUTOSAR meta model.

`ara::com` does standardize the interface of the generated Proxy class.The toolchain of an AP product vendor will generate a Proxy implementation class exactly implementing this interface.

Note: Since the interfaces the Proxy class has to provide are defined by `ara::com`, a generic (product independent) generator could generate an abstract class or a mock class against which the application developer could implement his service consumer application. This perfectly suits the platform vendor independent development of Adaptive AUTOSAR SWCs.

`ara::com` expects proxy related artifacts inside a namespace "proxy". This namespace is typically included in a namespace hierarchy deduced from the service definition and its context.

```
1  class RadarServiceProxy {
2    public:
3      /**
4       * \brief Implementation is platform vendor specific
5       *
6       * A HandleType must contain the information that is needed to create
7       * a proxy.
8       *
9       * This information shall be hidden.
10      * Since the platform vendor is responsible for creation of handles,
      the
11      * ctor signature is not given as it is not of interest to the user.
12      */
13     class HandleType {
14         /**
15          * \brief Two ServiceHandles are considered equal if they represent
16          * the same service instance.
17          *
18          * \param other
19          *
20          * \return bool
21          */
22         inline bool operator==(const HandleType &other) const;
23         const ara::com::InstanceIdentifier &GetInstanceId() const;
24     };
25
26     /**
27      * StartFindService does not need an explicit version parameter as this
28      * is internally available in ProxyClass.
29      * That means only compatible services are returned.
30      *
31      * \param handler this handler gets called any time the service
32      * availability of the services matching the given
33      * instance criteria changes. If you use this variant of
34      * FindService, the Communication Management has to
```

```
35      * continuously monitor the availability of the services
36      * and call the handler on any change.
37      *
38      * \param instanceId which instance of the service type defined
39      * by T shall be searched/found.
40      *
41      * \return a handle for this search/find request, which shall
42      * be used to stop the availability monitoring and related
43      * firing of the given handler. (\see StopFindService())
44      */
45     static ara::com::FindServiceHandle StartFindService(
46     ara::com::FindServiceHandler<RadarServiceProxy::HandleType> handler,
47     ara::com::InstanceIdentifier instanceId);
48
49     /**
50      * This is an overload of the StartFindService method using an
51      * instance specifier, which gets resolved via service instance
52      * manifest.
53      * \param instanceSpec instance specifier
54      */
55     static ara::com::FindServiceHandle StartFindService(
56     ara::com::FindServiceHandler<RadarServiceProxy::HandleType> handler,
57     ara::core::InstanceSpecifier instanceSpec);
58
59     /**
60      * This is an overload of the StartFindService method using neither
61      * instance specifier nor instance identifier.
62      * Semantics is, that ALL instances of the service shall be found, by
63      * using all available/configured technical bindings.
64      *
65      */
66     static ara::com::FindServiceHandle StartFindService(
67     ara::com::FindServiceHandler<RadarServiceProxy::HandleType> handler);
68
69     /**
70      * Method to stop finding service request (see above)
71      */
72     static void StopFindService(ara::com::FindServiceHandle handle);
73
74     /**
75      * Opposed to StartFindService(handler, instance) this version
76      * is a "one-shot" find request, which is:
77      * - synchronous, i.e. it returns after the find has been done
78      *   and a result list of matching service instances is
79      *   available. (list may be empty, if no matching service
80      *   instances currently exist)
81      * - does reflect the availability at the time of the method
82      *   call. No further (background) checks of availability are
83      *   done.
84      *
85      * \param instanceId which instance of the service type defined
86      * by T shall be searched/found.
87      *
88      */
89     static ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType>
    FindService(
```

```
90      ara::com::InstanceIdentifier instanceId);
91
92      /**
93       * This is an overload of the FindService method using an
94       * instance specifier, which gets resolved via service instance
95       * manifest.
96       */
97      static ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType>
        FindService(
98      ara::core::InstanceSpecifier instanceSpec);
99
100     /**
101      * This is an overload of the StartFindService method using neither
102      * instance specifier nor instance identifier.
103      * Semantics is, that ALL instances of the service shall be found, by
104      * using all available/configured technical bindings.
105      */
106     static ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType>
        FindService();
107
108     /**
109      * \brief The proxy can only be created using a specific
110      * handle which identifies a service.
111      *
112      * This handle can be a known value which is defined at
113      * deployment or it can be obtained using the
114      * ProxyClass::FindService method.
115      *
116      * \param handle The identification of the service the
117      * proxy should represent.
118      */
119     explicit RadarServiceProxy(HandleType &handle);
120
121     /**
122      * proxy instances are not copy constructible.
123      */
124     RadarServiceProxy(RadarServiceProxy &other) = delete;
125
126     /**
127      * proxy instances are not copy assignable
128      */
129     RadarServiceProxy& operator=(const RadarServiceProxy &other) = delete;
130
131     /**
132      * \brief Public member for the BrakeEvent
133      */
134     events::BrakeEvent BrakeEvent;
135
136     /**
137      * \brief Public Field for UpdateRate
138      */
139     fields::UpdateRate UpdateRate;
140
141     /**
142      * \brief Public member for the Calibrate method
143      */
```

```
144     methods::Calibrate Calibrate;
145
146     /**
147      * \brief Public member for the Adjust method
148      */
149     methods::Adjust Adjust;
150
151     /**
152      * \brief Public member for the LogCurrentState fire-and-forget method
153      */
154     methods::LogCurrentState LogCurrentState;
155 };
```

Listing 6.5: RadarService Proxy

6.2.1 Constructor and Handle Concept

As you can see in the figure 6.5 `ara::com` prescribes the Proxy class to provide a constructor. This means, that the developer is responsible for creating a proxy instance to communicate with a possibly remote service.

The `ctor` takes a parameter of type `RadarServiceProxy::HandleType` — an inner class of the generated proxy class. Probably the immediate question then is: *"What is this handle and how to create it/where to get it from?"*

What it is, should be straightforward: After the call to the `ctor` you have a proxy instance, which allows you to communicate with the service, therefore the handle has to contain the needed addressing information, so that the Communication Management binding implementation is able to contact the service.

What exactly this address information contains is totally dependent on the binding implementation/technical transport layer!

That already partly answers the question *"how to create/where to get it"*: Really creating is not possible for an application developer as he is — according to AUTOSAR core concepts — implementing his application AP product and therefore Communication Management independent.

The solution is, that `ara::com` provides the application developer with an API to find service instances, which returns such handles.

This part of the API is described in detail here: subsection 6.2.2. The co-benefit from this approach — that proxy instances can only be created from handles, which are the result of a "FindService" API — is, that you are only able to create proxies, which are really backed by an existing service instance.

AUTOSAR Binding Implementer Hint

When implementing an `ara::com` compliant binding, you have to decide what information you embed into the implementation of your handle class and how you react in your implementation of the proxy class `ctor` on the information embedded into your handle implementation.

To get the bigger picture you have to look at the handle-type in the context of the `Service Discovery` mechanism (see section 9.2) and to understand what `Multi-Binding` means (see section 9.3). When you have implemented the `Service Discovery` functionality within your AP product and therefore the functionality of subsection 6.2.2 you may most likely encounter those typical scenarios, when an `ara::com` application calls `ProxyClass::FindService`:

- the found service is located on a different node on the network
- the found service is located within a different application on the same node (within the same AP infrastructure)
- the found service is located within the same process

The possible combinations could increase complexity: For an existing service type any of those cases may apply at the same time — one instance of the service which the applications talks to is locally in the same process (this is not that strange if you think of large application with much code re-use), one on the same ECU in a different process and one on a remote ECU.

We (`ara::com` design team) require that such a setup works seamlessly for the `ara::com` user. By the way: this functionality is called `Multi-Binding` as you have a service abstraction in the form of a proxy class, which is bound to multiple different transport bindings.

In all cases the application developer using `ara::com` interacts with instances of the same Proxy class, where you provided the implementation.

The somewhat obvious expectation from an AP product is now, that it provides ways to communicate in those different cases efficiently.

Meaning that if the developer uses a proxy instance constructed from an instance of `HandleType`, which denotes the instance of the service local to the proxy user, then the Proxy implementation should use a different technical solution (in this case for instance a simple local function call / local in address space copies) than in the case of a proxy constructed from an instance of `HandleType` denoting a remote service instance.

In a nutshell: What the AP product vendor has to provide, is a Proxy class implementation, which is able to delegate to completely different transport layer implementations depending on the information contained in the instance of `HandleType` given in the `ctor`.

Table 6.1: AUTOSAR Binding Implementer Hint - handle class

So the question which probably might come up here: Why this indirection, that an application developer first has to call some `ara::com` provided functionality, to get a handle, which I then have to use in a `ctor` call? `ara::com` could have given back directly a proxy instance instead of a handle from "FindService" functionality.

The reason for that could be better understood, after reading how `ara::com` handles the access to events (subsection 6.2.3). But what is sufficient to say at this point is, that a proxy instance contains certain state.

And because of this there are use cases, where the application developer wants to use different instances of a proxy, all "connected" to the same service instance.

So if you just accept, that there are such cases, the decision for this indirection via handles becomes clear: `ara::com` cannot know, whether an application developer wants always the same proxy instance (explicitly sharing state) or always a new instance each time he triggers some "FindService" functionality, which returns a proxy for exactly the same service instance.

So by providing this indirection/decoupling the decision is in the hands of the `ara::com` user.

Instances of the Proxy class on the other hand are neither copy constructible nor copy assignable! This is an explicit design decision, which complements the idea of forcing the construction via `HandleType`.

The instances of a proxy class might be very resource intensive because of owning event/field caches, registered handlers, complex state,... and so on. Thus, when allowing copy construction/copy assignment, there is a risk that such copies are done unintended.

So — in a nutshell — forcing the user to go the route via `HandleType` for Proxy creation shall sensitize him, that this decision shall be well thought out.

6.2.2 Finding Services

The Proxy class provides class (static) methods to find service instances, which are compatible with the Proxy class.

Since the availability of service instances is dynamic by nature, as they have a life cycle, `ara::com` provides two different ways to do a “FindService” for convenience in general:

- `StartFindService` is a class method, which starts a continuous “FindService” activity in the background, which notifies the caller via a given callback anytime the availability of instances of the service changes.
- `FindService` is a one-off call, which returns available instances at the point in time of the call.

Both of those methods come in three different overrides, depending on the instance identifier approach taken (see section 6.1):

- one taking an `ara::com::InstanceIdentifier`
- one taking an `ara::core::InstanceSpecifier`
- one taking NO argument.

The semantics of no-argument variant is simple: Find all services of the given type, irrespective of their binding and binding specific instance identifier. Note, that only technical bindings will be used for finding/searching, which are configured for the corresponding service interface within the service instance manifest in the form of a `ServiceInterfaceDeployment`.

Note that only technical bindings will be used for finding/searching, which are configured for the corresponding service interface within the service instance manifest in the form of a `service interface deployment`.

The synchronous one-off variant `FindService` returns a container of handles (see subsection 6.2.1) for the matching service instances, which might also be empty, if no matching service instance is currently available.

Opposed to that, the `StartFindService` returns a `FindServiceHandle`, which can be used to stop the ongoing background activity of monitoring service instance availability via call to `StopFindService`.

The first (and specific for this variant) parameter to `StartFindService` is a user provided handler function with the following signature:

```
1    using FindServiceHandler = std::function<void(ServiceHandleContainer<T
    >, FindServiceHandle)>;
```

Any time the binding detects, that the availability of service instances matching the given instance criteria in the call to `StartFindService` has changed, it will call the user provided handler with an updated list of handles of the now available service instances.

Right after being called, `StartFindService` behaves similar to `FindService` in the sense, that it will fire the user provided handler function with the currently available service instances, which might be also an empty handle list.

After that initial callback, it will call the provided handler again in case of changes of this initial service availability.

Note, that it is explicitly allowed, that the `ara::com` user/developer does call `StopFindService` within the user provided handler.

For this purpose, the handler explicitly gets the `FindServiceHandle` argument. The handler needs not to be re-entrant. This means, that the binding implementer has to care for serializing calls to the user provided handler function.

Note, that `ServiceHandleContainer` can be implemented as an allocating or non-allocating container, when used either as a return value of `FindService` or as a parameter to `FindServiceHandler`, as long as it fulfils general and sequence container requirements of the C++ programming language.

6.2.2.1 Auto Update Proxy instance

Regardless whether you use the one-off `FindService` or the `StartFindService` variant, in both cases you get a handle identifying the — possibly remote — service instance, from which you then create your proxy instance.

But what happens if the service instance goes down and later comes up again e.g. due to some life cycle state changes? Can the existing proxy instance at the service consumer side still be re-used later, when the service instance gets available again?

The good news is: The `ara::com` design team decided to require this re-use possibility from the binding implementation as it eases the typical task of implementing service consumers.

In the service based communication universe it is expected, that during the life time of the entire system (e.g. vehicle) service provider and consumer instances are starting up and going down again due to their own life cycle concepts frequently.

To deal with that, there is the service discovery infrastructure, where the life cycle of service providers and consumers is monitored in terms of service offerings and service (re)subscriptions!

If a service consumer application has instantiated a service proxy instance from a handle returned from some of the `Find Service` variants, the sequence which might possibly occur is shown in the figure below.

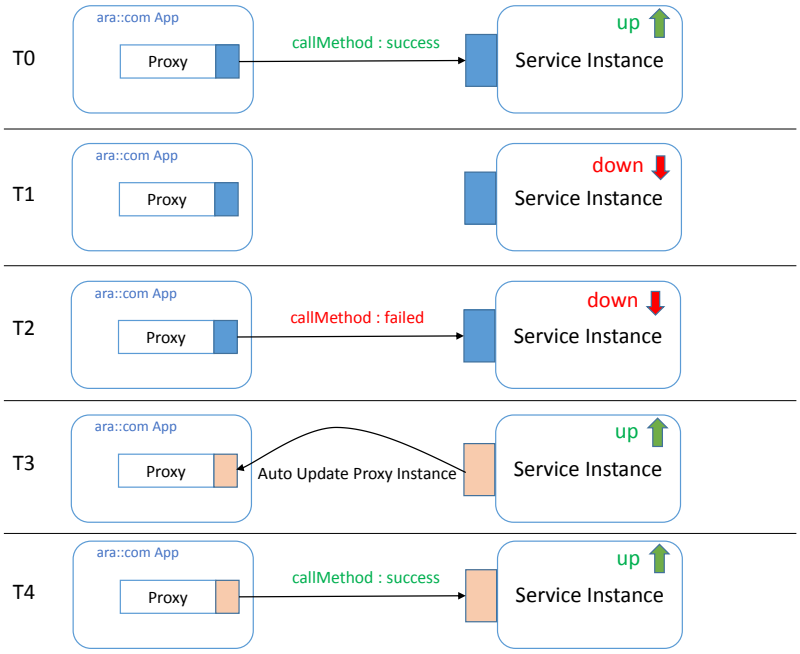

Figure 6.1: Auto Updating of Proxy Instance

Explanation of figure 6.1:

- **T0**: The service consumer may successfully call a service method of that proxy (and `GetSubscriptionState()` on subscribed events will return `kSubscribed` according to 6.2.3.2).
- **T1**: The service instance goes down, correctly notified via service discovery.
- **T2**: A call of a service method on that proxy will lead to a checked exception (`ara::com::ServiceNotAvailableException`), since the targeted service instance of the call does not exist anymore. Correspondingly `GetSubscriptionState()` on any subscribed event will return `kSubscriptionPending` (see also 6.2.3.2) at this point even if the event has been successfully subscribed (`kSubscribed`) before.
- **T3**: The service instance comes up again, notified via service discovery infrastructure. The Communication Management at the proxy side will be notified and will silently update the proxy object instance with a possibly changed transport layer addressing information. This is illustrated in the figure with transport layer part of the proxy, which changed the color from blue to rose. The *Binding implementer hint* part below discusses this topic more detailed.
- **T4**: Consequently service method calls on that proxy instance will succeed again and `GetSubscriptionState()` on events which the service consumer had subscribed before, will return `kSubscribed` again.

This convenience behavior of a proxy instance saves the implementer of a service consumer from either:

- polling via `GetSubscriptionState()` on events, which indicates that service instance has gone down
- re-triggering a one-off `FindService` to get a new handle.

or:

- registering a `FindServiceHandler`, which gets called in case service instance gets down or up with a new handle.

and then to recreate a proxy instance from the new handle (and redo needed event subscribe calls).

Note, in case you have registered a `FindServiceHandler`, then the binding implementation must assure, that it does the "auto updating" of existing proxy instances **before** it calls the registered `FindServiceHandler`!

The reason for this is: It shall be supported, that the application developer can interact successfully with an existing proxy instance within the `FindServiceHandler`, when the handle of the proxy instance is given in the call, signaling, that the service instance is up again.

This expectation is shown in the following code snippet:

```
/**
 * Reference to radar instance, we work with,
 * initialized during startup
 */
RadarServiceProxy *myRadarProxy;

void radarServiceAvailabilityHandler(ServiceHandleContainer<
    RadarServiceProxy::HandleType> curHandles, FindServiceHandle handle) {
     for (RadarServiceProxy::HandleType handle : curHandles) {
         if (handle.GetInstanceId() == myRadarProxy->GetHandle().
    GetInstanceId()) {
             /**
              * This call on the proxy instance shall NOT lead to an
    exception,
              * regarding service instance not reachable, since proxy
    instance
              * should be already auto updated at this point in time.
              */
             ara::core::Future<Calibrate::Output> out =
             myRadarProxy->Calibrate("test");

             // ... do something with out.
         }
     }
}
```

Listing 6.6: Access to proxy instance within FindService handler

▽

△

AUTOSAR Binding Implementer Hint

For the binding implementer it is important to understand, that this “auto updating” of existing proxy instances shall also work, when the low level transport level addressing of the service instance has changed after it went down and up again!

Whether this might happen at all, fully depends on the transport layer binding implementation! For instance, if we have a `SOME/IP` network binding in place between the proxy instance and the service instance implementation, after a service instance restart, the port number under which the service instance can be reached, might indeed have changed. Nevertheless the “auto updating” of the proxy instance shall seamlessly work!

If you recall the discussion (see Table 6.2.1 and section 9.3), where we gave some hints, what a binding implementer could/would embed into the proxy handle instance, then the question might come up, how it interferes with the “auto updating” in place?

At the point in time the handle is generated by the binding/discovery implementation AP product, most likely the initial transport layer addressing information of the service instance will be encoded into the handle, so that the proxy instance created from it, is able to contact the service instance.

Note, that this could be also a performance optimization for setups, where the transport layer addressing information of the service instance remains constant throughout life cycles! In this case you could do a service lookup once in the life time and store the returned handle somewhere persistently.

Anytime the service consumer starts up again — instead of triggering one of the `Find Service` variants — it could directly re-use the persisted handle to create the proxy instance. The optimization lies in the fact, that no — eventually costly — discovery needs to be done first.

In case the proxy instance gets “auto updated” behind the scenes as required by `ara::com`, when the service instance gets re-offered, it might be the case — as we did lay out above — that the transport layer addressing information has changed. This would obviously mean, that the proxy instance after an update uses a different transport layer addressing information than was contained in the handle from which the instance has been formerly constructed!

On the other hand this also means, that the user is allowed to create a proxy instance from an outdated handle (outdated in the sense, that the transport layer addressing information is now invalid). Here two different cases have to be distinguished:

- at the time of construction of the proxy instance with this outdated handle, the binding implementation is NOT aware of the new transport layer address. This would have the effect, that directly AFTER the construction of the proxy with outdated addressing information, calls to the service instance may fail.

  But at the time the service instance gets (re)offered and the new transport layer addressing information of the instance is visible/known to the binding implementation

▽

▽

△

△

of the AP product it shall apply the “auto update” to the proxy instance (updating with new transport layer address).

- at the time of construction of the proxy instance with this outdated handle, the binding implementation is already aware of the new transport layer address and uses this one instead.

The “auto update” mechanism even has to work, if the service instance is changing transport layer mechanism completely.

Table 6.2: AUTOSAR Binding Implementer Hint - auto update

6.2.3 Events

For each event the remote service provides, the proxy class contains a member of a event specific wrapper class. In our example the member has the name `BrakeEvent` and is of type `events::BrakeEvent`.

As you see in 6.5 all the event classes needed for the proxy class are generated inside a specific namespace `events`, which is contained inside the `proxy` namespace.

The member in the proxy is used to access events/event data, which are sent by the service instance our proxy is connected to. Let’s have a look at the generated event class for our example:

```
1  class BrakeEvent {
2      /**
3       * \brief Shortcut for the events data type.
4       */
5      using SampleType = RadarObjects;
6
7      /**
8       * \brief The application expects the CM to subscribe the event.
9       *
10      * The Communication Management shall try to subscribe and resubscribe
11      * until \see Unsubscribe() is called explicitly.
12      * The error handling shall be kept within the Communication Management
      .
13      *
14      * The function returns immediately. If the user wants to get notified,
15      * when subscription has succeeded, he needs to register a handler
16      * via \see SetSubscriptionStateChangeHandler(). This handler gets
17      * then called after subscription was successful.
18      *
19      * \param maxSampleCount maximum number of samples, which can be held.
20      */
21     void Subscribe(size_t maxSampleCount);
22
23     /**
24      * \brief Query current subscription state.
```

```
25      *
26      * \return Current state of the subscription.
27      */
28     ara::com::SubscriptionState GetSubscriptionState() const;
29
30     /**
31      * \brief Unsubscribe from the service.
32      */
33     void Unsubscribe();
34
35     /**
36      * \brief Get the number of currently free/available sample slots.
37      *
38      * \return number from 0 - N (N = count given in call to Subscribe())
39      *         or an ErrorCode in case of number of currently held samples
40      *         already exceeds the max number given in Subscribe().
41      */
42     ara::core::Result<size_t> GetFreeSampleCount() const noexcept;
43
44     /**
45      * Setting a receive handler signals the Communication Management
46      * implementation to use event style mode.
47      * I.e. the registered handler gets called asynchronously by the
48      * Communication Management as soon as new event data arrives for
49      * that event. If the user wants to have strict polling behavior,
50      * where no handler is called, NO handler should be registered.
51      *
52      * Handler may be overwritten anytime during runtime.
53      *
54      * Provided Handler needs not to be re-entrant since the
55      * Communication Management implementation has to serialize calls
56      * to the handler: Handler gets called once by the MW, when new
57      * events arrived since the last call to GetNewSamples().
58      *
59      * When application calls GetNewSamples() again in the context of the
60      * receive handler, MW must - in case new events arrived in the
61      * meantime - defer next call to receive handler until after
62      * the previous call to receive handler has been completed.
63      */
64     void SetReceiveHandler(ara::com::EventReceiveHandler handler);
65
66     /**
67      * Remove handler set by SetReceiveHandler()
68      */
69     void UnsetReceiveHandler();
70
71     /**
72      * Setting a subscription state change handler, which shall get
73      * called by the Communication Management implementation as soon
74      * as the subscription state of this event has changed.
75      *
76      * Communication Management implementation will serialize calls
77      * to the registered handler. If multiple changes of the
78      * subscription state take place during the runtime of a
79      * previous call to a handler, the Communication Management
80      * aggregates all changes to one call with the last/effective
```

```
81          * state.
82          *
83          * Handler may be overwritten during runtime.
84          */
85         void SetSubscriptionStateChangeHandler(
86             ara::com::SubscriptionStateChangeHandler handler);
87
88         /**
89          * Remove handler set by SetSubscriptionStateChangeHandler()
90          */
91         void UnsetSubscriptionStateChangeHandler();
92
93         /**
94          * \brief Get new data from the Communication Management
95          * buffers and provide it in callbacks to the given callable f.
96          *
97          * \pre BrakeEvent::Subscribe has been called before
98          * (and not be withdrawn by BrakeEvent::Unsubscribe)
99          *
100         * \param f
101         * \parblock
102         * callback, which shall be called with new sample.
103         *
104         * This callable has to fulfill signature
105         * void(ara::com::SamplePtr<SampleType const>)
106         * \parblockend
107         *
108         * \param maxNumberOfSamples
109         * \parblock
110         * upper bound of samples to be fetched from middleware buffers.
111         * Default value means "no restriction", i.e. all newly arrived samples
112         * are fetched as long as there are free sample slots.
113         * \parblockend
114         *
115         * \return Result, which contains the number of samples,
116         * which have been fetched and presented to user via calls to f or an
117         * ErrorCode in case of error (e.g. precondition not fullfilled)
118         */
119        template <typename F>
120        ara::core::Result<size_t> GetNewSamples(
121            F&& f,
122            size_t maxNumberOfSamples = std::numeric_limits<size_t>::max());
123 };
```

Listing 6.7: Proxy side BrakeEvent Class

The data type of the event data in our example event is `RadarObjects` (see 6.1). The first you encounter is the using-directive which assigns the generic name `SampleType` to the concrete type, which is then used throughout the interface.

6.2.3.1 Event Subscription and Local Cache

The mere fact, that there exists a member of the event wrapper class inside the proxy instance does not mean, that the user gets instant access to events raised/sent out by service instance.

First you have to “subscribe” for the event, in order to tell the Communication Management, that you are now interested in receiving events.

For that purpose the event wrapper class of `ara::com` provides the method

```
1    /**
2     * \brief The application expects the CM to subscribe the event.
3     *
4     * ....
5     *
6     * \param maxSampleCount maximum number of samples, which can be held.
7     */
8    void Subscribe(size_t maxSampleCount);
```

This method expects a parameter `maxSampleCount`, which basically informs Communication Management implementation, how many event samples the application intends to hold at maximum. Therefore — with calling this method, you not only tell the Communication Management, that you now are interested in receiving event updates, but you are at the same time setting up a "local cache" for those events bound to the event wrapper instance with the given `maxSampleCount`.

This cache is allocated and filled by the Communication Management implementation, which hands out smartpointers to the application for accessing the event sample data. How that works in detail is described in subsubsection 6.2.3.3).

6.2.3.2 Monitoring Event Subscription

The call to the `Subscribe` method is asynchronous by nature. This means that at the point in time `Subscribe` returns, it is just the indication, that the Communication Management has accepted the order to care for subscription.

The subscription process itself may (most likely, but depends on the underlying IPC implementation) involve the event provider side. Contacting the possibly remote service for setting up the subscription might take some time.

So the binding implementation of the subscribe is allowed to return immediately after accepting the subscribe, even if for instance the remote service instance has not yet acknowledged the subscription (in case the underlying IPC would support mechanism like acknowledgment at all). If the user — after having called `Subscribe` — wants to get feedback about the success of the subscription, he might call:

```
1    /**
2     * \brief  query current subscription state.
3     *
4     * \return current state of the subscription.
```

```
     */
    ara::com::SubscriptionState GetSubscriptionState() const;
```

In the case the underlying IPC implementation uses some mechanism like a subscription acknowledge from the service side, then an immediate call to `GetSubscriptionState` after `Subscribe` may return `kSubscriptionPending`, if the acknowledge has not yet arrived.

Otherwise — in case the underlying IPC implementation gets instant feedback, which is very likely for local communication — the call might also already return `kSubscribed`.

If the user needs to monitor the subscription state, he has two possibilities:

- Polling via `GetSubscriptionState`
- Registering a handler, which gets called, when the subscription state changes

The first possibility by using `GetSubscriptionState` we have already described above. The second possibility relies on using the following method on the event wrapper instance:

```
    /**
     * Setting a subscription state change handler, which shall get called by
     * the Communication Management implementation as soon as the subscription
     * state of this event has changed.
     *
     * Handler may be overwritten during runtime.
     */
    void SetSubscriptionStateChangeHandler(ara::com::SubscriptionStateChangeHandler handler);
```

Here the user may register a handler function, which has to fulfill the following signature:

```
    enum class SubscriptionState { kSubscribed, kNotSubscribed, kSubscriptionPending };
    using SubscriptionStateChangeHandler = std::function<void(SubscriptionState)>;
```

Anytime the subscription state changes, the Communication Management implementation calls the registered handler. A typical usage pattern for an application developer, who wants to get notified about latest subscription state, would be to register a handler **before** the first call to `Subscribe`.

After having accepted the “subscribe order” the Communication Management implementation will call the handler first with argument `SubscriptionState.kSubscriptionPending` and later — as it gets acknowledgment from the service side — it will call the handler with argument `SubscriptionState.kSubscribed`.

Again the note: If the underlying implementation does not support a subscribe acknowledgment from the service side, the implementation could also skip the first call

to the handler with argument `SubscriptionState.kSubscriptionPending` and **directly** call it with argument `SubscriptionState.kSubscribed`.

Calls to the registered "subscription state change" handler are done fully asynchronous. That means, they can even happen, while the call to `Subscribe` has not yet returned. The user has to be aware of this!

Once the user has registered such a "subscription state change" handler for a certain event, he may receive multiple calls to this handler. Not only initially, when the state changes from `SubscriptionState.kNotSubscribed` to `SubscriptionState.kSubscribed` (eventually via an intermediate step `SubscriptionState.kSubscriptionPending`), but also anytime later as the service providing this event may have a certain life-cycle (maybe bound to certain vehicle modes).

The service might therefore toggle between availability and (temporarily) unavailability or it might even unexpectedly crash and restart. Those changes of the availability of the service instance providing the event may be visible to the proxy side Communication Management implementation.

The Communication Management therefore will fire the registered "subscription state change" handler, whenever it detects such changes, which have influence on the event subscription state.

Additionally (and maybe even more important) — the Communication Management implementation takes care of renewing/updating event subscriptions done by the user, whenever needed.

This mechanism is closely coupled with the "Auto Update Proxy instance" mechanism already described above (6.2.2.1): Since the Communication Management implementation monitors the availability of the service instances, the service proxies are connected to it automatically once the service is available.

The mechanism does not only "auto-update" its proxies if needed, but also "silently" re-subscribes any event subscription already done by the user, after it has updated a proxy instance.

This can be roughly seen as a very useful comfort feature — without this "re-subscribe after update", the "auto-update" alone seemed to be a halfhearted approach.

With registration of a "subscription state change" handler, the user has now another possibility to monitor the current availability of a service! Beside the possibility to register a `FindServiceHandler` as described in 6.2.2, the user, who has registered a "subscription state change" handler, can monitor the service availability indirectly by calls to his handler.

In case the service instance, the proxy is connected to, goes down, the Communication Management calls the handler with argument `SubscriptionState.kSubscriptionPending`. As soon as the "re-subscribe after update" was successful, the Communication Management calls the handler with argument `SubscriptionState.kSubscribed`.

An `ara::com` compliant Communication Management implementation has to serialize calls to the user registered handler. I.e.: If a new subscription state change happens, while the user provided handler from a previous call of a state change is still running, the Communication Management implementation has to postpone the next call until the previous has returned.

Several subscription state changes, which happen during the runtime of a user registered state change handler, shall be aggregated to one call to the user registered handler with the effective/last state.

AUTOSAR Binding Implementer Hint

Depending on the used IPC or transport layer technology the lifetime/availability of the service as a whole (represented by the proxy instance) and the availability of its subparts (e.g. events, fields methods) may be distinguishable or not.

With SOME/IP f.i., there is the contract, that the service availability as a whole is notified and the expectation/contract is, that then automatically all subparts are available as well.

Here in `ara::com` we do not require this tight coupling! So generally it would be supported/allowed, that a service instance could be found (see subsection 6.2.2) and methods could be called on it (via the proxy), but the "subscription state" switches to `SubscriptionState.kNotSubscribed`, because the service has withdrawn just the event, which the user has subscribed to.

The mechanism of registering the "subscription state change" handler with the expectation to steadily monitor state changes in the background is similar or related to the mechanism of `Proxy::FindService` (see subsection 6.2.2), where the user can also register a handler to monitor availability changes of service instances.

So from implementation view point — depending on the used transport layer technology — those mechanisms may depend on each other or may be tightly coupled implementation-wise.

Table 6.3: AUTOSAR Binding Implementer Hint - availability of service

6.2.3.3 Accessing Event Data — aka Samples

So, after you successfully subscribed to an event according to the previous chapters, how is the access to received event data samples achieved? The event data, which is sent from the event emitter (service provider) to subscribing proxy instances is — in typical IPC implementations — accumulated/queued in some buffers (e.g. kernel buffers, special IPC implementation controlled shared memory regions, ...). So there has to be taken an **explicit** action, to get/fetch those event samples from those buffers, eventually deserialze it and and then put them into the event wrapper class instance specific cache in form of a correct SampleType. The API to trigger this action is `GetNewSamples`.

```
1     /**
2      * \brief Get new data from the Communication Management
3      * buffers and provide it in callbacks to the given callable f.
4      *
5      * ....
6      */
7     template <typename F>
8     ara::core::Result<size_t> GetNewSamples(
9         F&& f,
10        size_t maxNumberOfSamples = std::numeric_limits<size_t>::max());
```

As you can see, the API is a function template, due to the fact, that the first parameter `f` is a very flexible user provided `Callable`, which has to fulfill the following singnature requirement: `void(ara::com::SamplePtr<SampleType const>)`.

The second argument of type `size_t` controls the maximum number of event samples, that shall be fetched/deserialized from the middleware buffers and then presented to the application in form of a call to `f`.

On a call to `GetNewSamples()`, the `ara::com` implementation checks first, whether the number of event samples held by the application already exceeds the maximum number, which it had committed in the previous call to `Subscribe()`. If so, an `ara::core::ErrorCode` is returned. Otherwise `ara::com` implementation checks, whether underlying buffers contain a new event sample and — if it's the case — deserializes it into a sample slot and then calls the application provided `f` with a `SamplePtr` pointing to this new event sample. This processing (checking for further samples in the buffer and calling back the application provided callback `f`) is repeated until either:

- there aren't any new samples in the buffers
- there are further samples in the buffers, but the application provided `maxNumberOfSamples` argument in call to `GetNewSamples()` has been reached.
- there are further samples in the buffers, but the application already exceeds its `maxSampleCount`, which it had committed in `Subscribe()`.

Within the implementation of callback `f`, which the application/user provides, it can be decided, what to do with the passed `SamplePtr` argument (i.e. by eventually doing a deep inspection of the event data): Shall the new sample be "thrown away", because it is not of interest or shall it be kept for later. To get an idea, what keeping/throwing away

of event samples means, the semantics of the `SamplePtr`, which is the access/entry point to the event sample data has to be fully understood.

The following chapter shall clarify this.

The returned `ara::core::Result` contains either an ErrorCode or — in the success case — the number of calls to `f`, which have been done in the context of the `GetNewSamples` call.

6.2.3.4 Event Sample Management via `SamplePtr`s

A `SamplePtr`, which is handed over from the `ara::com` implementation to application/user layer is — from a semantical perspective — a unique-pointer (very similar to a `std::unique_ptr`): When the `ara::com` implementation hands it over an ownership transfer takes place. From now on the application/user is responsible for the lifetime management of the underlying sample. As long as the user doesn't free the sample by destroying the `SamplePtr` or by calling explicit assignment-ops/modifiers on the `SamplePtr` instance, the `ara::com` implementation can not reclaim the memory slot occupied by this sample.

Those memory-slots, in which the event sample data reside, are allocated by the `ara::com` implementation. This typically takes place in the context of the call to `Subscribe()`, where the user/application defines by parameter `maxSampleCount`, what maximum number of event data samples it wants to have concurrently accessible. Within later `GetNewSamples()` calls, the `ara::com` implementation then populates/-fills such a "sample slot" (if one is free) and passes a `SamplePtr` pointing to it in the user/application callback `f`.

In the callback implementation the user/application decides then, what to do with this passed in `SamplePtr`. If it wants to keep the sample for later access (i.e. after the return of the callback, it will make a copy at some outer scope location, where it fits in its software component architecture. The decission, whether to copy it (i.e. keep it) might simply depend on the properties/values of the event sample data. In this case the callback implementation is basically applying a "filter" on the received event samples. Since we stated, that the `SamplePtr` behaves like a `std::unique_ptr`), the above statement has to be slightly corrected: The implementation — when deciding to keep that event sample — is obviously not copying that passed in `SamplePtr`, but moving it to a outer scope location.

The small example in 6.8 shows — beside other things — in method `handleBrakeEventReception()` how such a callback implementation could realize simple filtering and moving of samples to a global storage with a "LastN" semantic for later use/processing.

AUTOSAR Binding Implementer Hint

As laid out in the previous chapters, the memory allocation for samples in the process space of the event receiving proxy instance is the job of the binding implementation. It should be done in the context of the `<event>::Subscribe` call as there might be resource critical applications (like ASIL qualified ones), which explicitly shift their calls to `Subscribe` to an initialization phase, where memory allocation from kernel is allowed. So - a binding implementation, which shifts real memory allocation to a later phase (after execution of the `Subscribe` call), should keep in mind, that there might be applications, which could not live with such an implementation decission as then the application has no control anymore, when memory is allocated from OS/kernel (e.g. via `mmap` or `brk`).

Also — in the previous chapters, we talked about the fact, that an application could exceed its number of occupied samples, which it had committed in the call to `Subscribe`. This could basically happen as a typical binding implementation will allocate an aditional "spare slot" on top of the number anounced by the application in `Subscribe`! For a typical application, which uses a LastN strategy, when working with samples, it will — in the course of `GetNewSamples()` replace an existing/held SamplePtr with a new one. To allow this essential semantics, the binding implementation needs an additional spare slot to serialize event data into, before providing it to the application in the callback. If now the application doesn't behave well and also decides to hold this new sample WITHOUT freeing another one, also the "spare slot" is blocked by the application and it now really exceeds the number of samples it had committed in `Subscribe`.

Table 6.4: AUTOSAR Binding Implementer Hint - memory allocation

6.2.3.5 Event-Driven vs Polling-Based access

As already promised, we fully support event-driven and polling approaches to access new data. For the polling approach no other APIs are needed than those, which we have discussed up to this point. The typical use case is, that you have an application, which is cyclically triggered to do some processing and provide its output to certain deadlines. This is the typical pattern of a regulator/control algorithm — the cyclic activation might additionally be driven by a real-time timer, which assures a minimal jitter.

In such a setup you call `GetNewSamples()` in each activation cycle and then use those updated cache data as input for the current processing iteration. Here it is fully sufficient to get the latest data to process at the time the processing algorithm is scheduled.

It would be counterproductive, if the Communication Management would notify your application anytime new data is available: This would just mean unnecessary context switches to your application process, since at the time you get the notification you do not want to process that new data as it is not time for it.

However, there are other use cases as well. If your application does not have such a cyclical, deadline driven approach, but shall simply react in case certain events occur, then setting up cyclical alarms and poll for new events via calls to `GetNewSamples()` is a bit off and vastly inefficient.

In this case you explicitly want the Communication Management to notify your application thereby issuing asynchronous context switches to your application process. We do support this flavor with the following API mechanism:

```
1    void SetReceiveHandler(ara::com::EventReceiveHandler handler);
```

This API allows you to register a user defined callback, which the Communication Management has to call in case new event data is available since the last call to `GetNewSamples()`. The registered function needs NOT to be re-entrant as the Communication Management has to serialize calls to the registered callback.

It is explicitly allowed to call `GetNewSamples()` from within the registered callback!

AUTOSAR Binding Implementer Hint

In case the binding implementation calls the registered user function and during the execution of this function new events arrive, but application has not yet again called `GetNewSamples()` from within the running user function, no new callback has to be fired!

But the newly arrived data must be visible to the next call to `GetNewSamples()` by the application. If during the execution of this function the user calls `GetNewSamples()` and during or after this `GetNewSamples()` still inside the registered receive handler new event data arrive, the Communication Management implementation has to delay the next call to the receive handler until the running call ends.

So the intuitive binding implementation would be to set a flag, when new event data arrives and the user defined receive handler is currently running. When the user receive handler ends, the Communication Management implementation just checks, whether the flag is set and in case just issues the next call to receive handler.

Table 6.5: AUTOSAR Binding Implementer Hint - calling registered user function

Note, that the user can alter the behavior between event-driven and polling style anytime as he also has the possibility to withdraw the user specific “receive handler” with the `UnsetReceiveHandler()` method provided by the event wrapper.

The following short code snippet is a simple example of how to work with events on proxy/client side. In this sample a proxy instance of type `RadarService` is created within `main` and a reception handler is registered, which gets called by the `ara::com` implementation any time new `BrakeEvent` events get received. This means, that in this example we are using the "Event-Driven" approach.

In our sample receive handler, we update our local cache with newly received events, thereby filtering out all BrakeEvent events, which do not fulfill a certain property. Afterwards we call a processing function, which processes the samples, we have decided to keep.

```
#include "RadarServiceProxy.hpp"
#include <memory>
#include <deque>

using namespace com::mycompany::division::radarservice;
using namespace ara::com;

/**
 * our radar proxy - initially the unique ptr is invalid.
 */
std::unique_ptr<proxy::RadarServiceProxy> myRadarProxy;

/**
 * a storage for BrakeEvent samples in fifo style
 */
std::deque<SamplePtr<const proxy::events::BrakeEvent::SampleType>>
    lastNActiveSamples;

/**
```

```
 * \brief application function, which processes current set of BrakeEvent samples.
 * \param samples
 */
void processLastBrakeEvents(
    std::deque<SamplePtr<const proxy::events::BrakeEvent::SampleType>>& samples) {
    // do whatever with those BrakeEvent samples ...
}

/**
 * \brief event reception handler for BrakeEvent events, which we register to get informed about new events.
 */
void handleBrakeEventReception() {
    /**
     * we get newly arrived BrakeEvent events into our process space.
     * For each sample we get passed in, we check for a certain property
     * "active" and if it fulfills the check, we move it into our Last10-storage.
     * So this few lines basically implement filtering and a LastN policy.
     */
    myRadarProxy->BrakeEvent.GetNewSamples(
    [](SamplePtr<proxy::events::BrakeEvent::SampleType> samplePtr) {
            if(samplePtr->active) {
                lastNActiveSamples.push_back(std::move(samplePtr));
                if (lastNActiveSamples.size() > 10)
                    lastNActiveSamples.pop_front();
            }
        });

    // ... now process those samples ...
    processLastBrakeEvents(lastNActiveSamples);
}

int main(int argc, char** argv) {

    auto handles = proxy::RadarServiceProxy::FindService();

    if (!handles.empty()) {
        /* we have at least one valid handle - we are not very particular here and take the first one to
         * create our proxy */
        myRadarProxy = std::make_unique<proxy::RadarServiceProxy>(handles[0]);

        /* we are interested in receiving the event "BrakeEvent" - so we subscribe for it. We want to access up to 10 events,
         * since our sample algo averages over at most 10.*/
        myRadarProxy->BrakeEvent.Subscribe(10);

        /* whenever new BrakeEvent events come in, we want be called, so we register a callback for it!
         * Note: If the entity we would subscribe to, would be a field instead of an event, it would be crucial, to
```

```
65          * register our reception handler BEFORE subscribing, to avoid race
     conditions. After a field subscription, you
66          * would get instantly so called "initial events" and to be sure
    not to miss them, you should care for that your
67          * reception handler is registered before.*/
68         myRadarProxy->BrakeEvent.SetReceiveHandler(
    handleBrakeEventReception);
69     }
70
71     // ... wait for application shutdown trigger by application exec mgmt.
72 }
```

Listing 6.8: Sample Code how to access Events

6.2.3.6 Buffering Strategies

AUTOSAR Binding Implementer Hint
At this point it surely makes sense to talk about reasonable buffering strategies for binding implementations. So this entire subsection is mainly of interest for an AP product vendor/binding implementer.

Table 6.6: AUTOSAR Binding Implementer Hint - buffering strategies

The following figure sketches a simple deployment, where we have a service providing an event, for which two different local adaptive SWCs have subscribed through their respective `ara::com` proxies/event wrappers.

As you can see in the picture both proxies have a local event cache. This is the cache, which gets filled via `GetNewSamples()`. What this picture also depicts is, that the service implementation sends its event data to a Communication Management buffer, which is apparently outside the process space of the service implementation — the picture here assumes, that this buffer is owned by kernel or it is realized as a shared memory between communicating proxies and skeleton or owned by a separate binding implementation specific “demon” process.

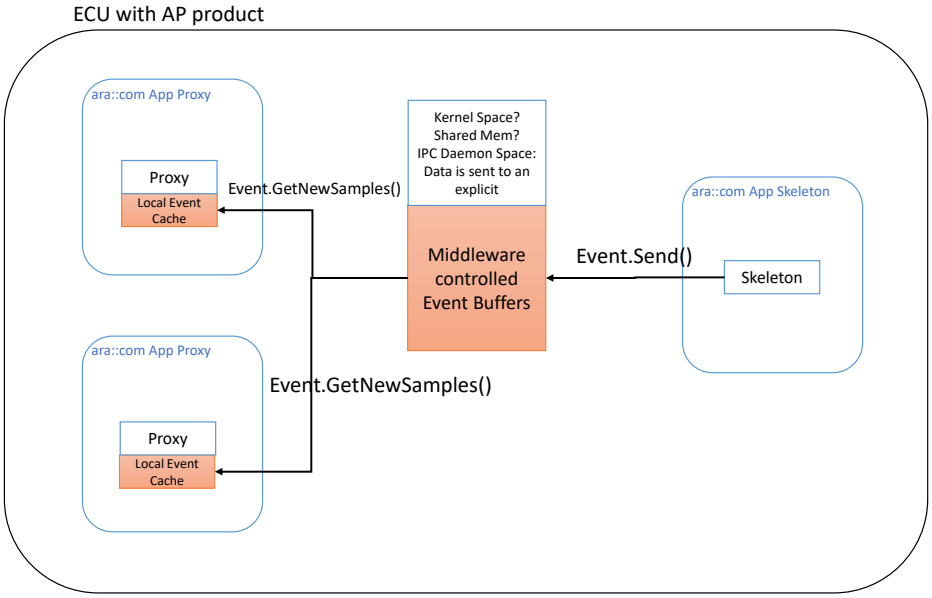

Figure 6.2: Event Buffering Approaches

The background of those assumptions made in the figure is the following: Adaptive applications are realized as processes with separated/protected memory/address spaces.

Event Data sent out by the service implementation (via the skeleton) cannot be buffered inside the service/skeleton process private address space: If that would be the case, event data access by the proxies would typically lead to context switches to the service application process.

Something, which we want to have total control over on service side via the `MethodCallProcessingMode` (see subsection 6.3.3) and should therefore not be triggered by the communication behavior of arbitrary service consumers. Now let's have a rough look at the three different places, where the buffer, which is target for the "send event" might be located:

- Kernel Space: Data is sent to a memory region not mapped directly to an application process. This is typically the case, when binding implementation uses IPC primitives like pipes or sockets, where data written to such a primitive ends up in kernel buffer space.
- Shared Memory: Data is sent to a memory region, which is also directly readable from receivers/proxies. Writing/reading between different parties is synchronized specifically (lightweight with mem barriers or with explicit mutexes).
- IPC-Daemon Space: Data is sent to an explicit non-application process, which acts as a kind of demon for the IPC/binding implementation. Note, that technically this approach might be built on an IPC primitive like communication via kernel space or shared memory to get the data from service process to demon process.

Each of those approaches might have different pros and cons regarding flexibility/size of buffer space, efficiency in terms of access speed/overhead and protection against malicious access/writing of buffers. Therefore consideration of different constraints in an AP product and its use might lead to different solutions.

What shall be emphasized here in this example, is, that the AP product vendor is explicitly encouraged to use a reference based approach to access event data: The `ara::com` API of event wrapper intentionally models the access via `SamplePtr`, which are passed to the callbacks and not the value!

In those rather typical scenarios of 1:N event communication, this would allow to have inside the “Local Event Cache” not the event data values itself but pointers/references to the data contained in a central Communication Management buffer. Updating the local cache via `GetNewSamples()` could then be implemented not as a value copy but as reference updates.

To be honest: This is obviously a coarse grained picture of optimization possibilities regarding buffer usage! As hinted here (section 9.1) data transferred to application processes must typically be de-serialized latest before first application access.

Since de-serialization has to be specific to the alignment of the consuming application the central sharing of an already de-serialized representation might be tricky. But at least you get the point, that the API design for event data access on the proxy/service consumer side gives room to apply event data sharing among consumers.

6.2.4 Methods

For each method the remote service provides, the proxy class contains a member of a method specific wrapper class.

In our example, we have three methods and the corresponding members have the name `Calibrate` (of type `methods::Calibrate`), `Adjust` (of type `methods::Adjust`) and `LogCurrentState` (of type `methods::LogCurrentState`). Just like the event classes the needed method classes of the proxy class are generated inside a specific namespace `methods`, which is contained inside the `proxy` namespace.

The method member in the proxy is used to call a method provided by the possibly remote service instance our proxy is connected to.

Let's have a look at the generated method class for our example — we pick out the `Adjust` method here:

```
class Adjust {
  public:
    /**
     * For all output and non-void return parameters
     * an enclosing struct is generated, which contains
     * non-void return value and/or out parameters.
     */
    struct Output {
        bool success;
        Position effective_position;
    };

    /**
     * \brief Operation will call the method.
     *
     * Using the operator the call will be made by the Communication
     * Management and a future returned, which allows the caller to
     * get access to the method result.
     *
     * \param[in] target_position See service description.
     *
     * \return A future containing Output struct
     */
    ara::core::Future<Output> operator()(const Position &target_position);
};
```

Listing 6.9: Proxy side Adjust Method Class

So the method wrapper class is not that complex. It just consists of two parts: An inner structure definition, which aggregates all OUT-/INOUT-parameters of the method, and a bracket operator, which is used to call the service method.

The operator contains all of the service methods IN-/INOUT-parameters as IN-parameters. That means INOUT-parameters in the abstract service method description are split in a pair of IN and OUT parameters in the `ara::com` API.

The return value of a call to a service method, which is **not** a “one-way method” is an `ara::core::Future`, where the template parameter is of the type of the inner `struct`, which aggregates all OUT-parameters of the method. More about this `ara::core::Future` in the following subsection.

6.2.4.1 One-Way aka Fire-and-Forget Methods

Before proceeding with the functionalities provided for “normal” methods, we briefly introduce “one-way methods” here as we already referred to this term in the previous section. `ara::com` supports a special flavor of a method, which we call “one-way” or “fire-and-forget”. Technically this is a method with only IN-params — no OUT-params and no raising of errors allowed. There is also no hand-shaking/synchronisation possible with the server! The client/caller therefore gets no feedback at all, whether the server/callee has processed a “one-way” call or not.

There are communication patterns, where such a best-effort approach is fully sufficient. In this case such a “one-way/fire-and-forget” semantics is very light-weight from a resource perspective. If we look at the signature of such a method, we see, that it is simpler, than that from a regular method:

```
1  class LogCurrentState {
2    public:
3      /**
4       * \brief Operation will call the method.
5       *
6       * Using the operator the call will be made by the Communication
7       * Management.
8       *
9       * It is a one-way method, so no feedback (return value/out-parameter)
10      * is given.
11      */
12     void operator()();
13 };
```

Listing 6.10: Proxy side LogCurrentState Method Class

AUTOSAR Binding Implementer Hint
To support the notion of a “one-way/fire-and-forget” method perfectly, implementation of such a call should be very asynchronously by nature. I.e. blocking the caller for some time to do a lot of housekeeping/setting up the call, is not expected by the caller of a “one-way/fire-and-forget” method! Since he isn’t even prepared for any error handling in this case, it is explicitly encouraged to do minimal processing in the callers context and shift as much as possible in an asynchronous context.

Table 6.7: AUTOSAR Binding Implementer Hint - fire-and-forget

6.2.4.2 Event-Driven vs Polling access to method results

Similar to the access to event data described in the previous section (subsection 6.2.3), we provide API support for an event-driven and polling-based approach also for accessing the results of a service method call.

The magic of differentiation between both approaches lies in the returned `ara::core::Future`: `ara::core::Future` is basically an extended version of the C++11/C++14 `std::future` class; see [4] for details.

Like in the event data access, event-driven here means, that the caller of the method (the application with the proxy instance) gets notified by the Communication Management implementation as soon as the method call result has arrived.

For a Communication Management implementation of `ara::com` this means, it has to setup some kind of waiting mechanism (WaitEvent) behind the scene, which gets woken up as soon as the method result becomes available, to notify the `ara::com` user. So how do the different usage patterns of the `ara::core::Future` work then?

Let's have a deeper look at our `ara::core::Future` and the interfaces it provides:

```
enum class future_status : uint8_t
{
ready,  ///< the shared state is ready
timeout,  ///< the shared state did not become ready before the specified
    timeout has passed
};

template <typename T, typename E = ErrorCode>
class Future {
  public:

    Future() noexcept = default;
    ~Future();

    Future(Future const&) = delete;
    Future& operator=(Future const&) = delete;

    Future(Future&& other) noexcept;
    Future& operator=(Future&& other) noexcept;

    /**
     * @brief Get the value.
     *
     * This function shall behave the same as the corresponding std::future
     function.
     *
     * @returns value of type T
     * @error Domain:error  the error that has been put into the
    corresponding Promise via Promise::SetError
     *
     */
    T get();

    /**
```

```
32      * @brief Get the result.
33      *
34      * Similar to get(), this call blocks until the value or an error is
    available.
35      * However, this call will never throw an exception.
36      *
37      * @returns a Result with either a value or an error
38      * @error Domain:error  the error that has been put into the
    corresponding Promise via Promise::SetError
39      *
40      */
41     Result<T, E> GetResult() noexcept;
42
43     /**
44      * @brief Checks if the Future is valid, i.e. if it has a shared state.
45      *
46      * This function shall behave the same as the corresponding std::future
     function.
47      *
48      * @returns true if the Future is usable, false otherwise
49      */
50     bool valid() const noexcept;
51
52     /**
53      * @brief Wait for a value or an error to be available.
54      *
55      * This function shall behave the same as the corresponding std::future
     function.
56      */
57     void wait() const;
58
59     /**
60      * @brief Wait for the given period, or until a value or an error is
    available.
61      *
62      * This function shall behave the same as the corresponding std::future
     function.
63      *
64      * @param timeoutDuration  maximal duration to wait for
65      * @returns status that indicates whether the timeout hit or if a value
     is available
66      */
67     template <typename Rep, typename Period>
68     future_status wait_for(std::chrono::duration<Rep, Period> const&
    timeoutDuration) const;
69
70     /**
71      * @brief Wait until the given time, or until a value or an error is
    available.
72      *
73      * This function shall behave the same as the corresponding std::future
     function.
74      *
75      * @param deadline  latest point in time to wait
76      * @returns status that indicates whether the time was reached or if a
    value is available
```

```
77       */
78      template <typename Clock, typename Duration>
79      future_status wait_until(std::chrono::time_point<Clock, Duration> const
        & deadline) const;
80
81      /**
82       * @brief Register a callable that gets called when the Future becomes
        ready.
83       *
84       * When @a func is called, it is guaranteed that get() and GetResult()
        will not block.
85       *
86       * @a func may be called in the context of this call or in the context
        of Promise::set_value()
87       * or Promise::SetError() or somewhere else.
88       *
89       * The return type of @a then depends on the return type of @a func (
        aka continuation).
90       *
91       * Let U be the return type of the continuation (i.e. std::result_of_t<
        std::decay_t<F>(ara::core::Future<T,E>)>).
92       * If U is ara::core::Future<T2,E2> for some types T2, E2, then the
        return type of @a then is ara::core::Future<T2,E2>,
93       * otherwise it is ara::core::Future<U>. This is known as implicit
        unwrapping.
94       *
95       * @param func  a callable to register
96       * @returns a new Future instance for the result of the continuation
97       */
98      template <typename F>
99      auto then(F&& func) -> SEE_COMMENT_ABOVE;
100
101     /**
102      * @brief Return whether the asynchronous operation has finished.
103      *
104      * If this function returns true, get(), GetResult() and the wait calls
         are guaranteed not to block.
105      *
106      * @returns true if the Future contains a value or an error, false
        otherwise
107      */
108     bool is_ready() const;
109 };
```

Listing 6.11: ara::core::Future Class

`GetResult()` returns a Result or an Error and throws no exception. Using `get()` returns the corresponding future and throws exceptions if necessary.

See [4] chapter "Error handling in the Adaptive Platform" for detailed documentation of the error handling approaches in AP.

Below is the sample of using "exception-based" approach to synchronously call a method:

```
1 using namespace ara::com;
```

```
2
3  int main() {
4      // some code to acquire a handle
5      // ...
6      RadarServiceProxy service(handle);
7      Future<Calibrate::Output>  callFuture = service.Calibrate(
       myConfigString);
8
9      /**
10      * Now we do a blocking get(), which will return in case the result
11      * (valid or exception) is received.
12      *
13      * If Calibrate could throw an exception and the service has set one,
14      * it would be thrown by get()
15      */
16     Calibrate::Output callOutput = callFuture.get();
17
18     // process callOutput ...
19     return 0;
20 }
```

Listing 6.12: Synchronous method call sample

In a nutshell: A synchronous call (from the viewpoint of the application developer) to a service method, simply consists of the ()-operator call-syntax with a subsequent blocking `get()` call on the returned future.

There are other ways for the user to get a notification from the Communication Management implementation as soon as the method result is available beside resuming execution from a blocking call to `get()`:

- The variants of “wait” , which the `ara::core::Future` has taken over from `std::future`. They basically provide the functionality of a blocking wait for the fulfillment of the future.
- Registering a callback method via `then()`. This is one of the extensions to the `std::future`; see [4] for details.

The plain parameterless `wait()` variant has the same blocking semantics like `get()` — i.e. blocks till the future has a valid result (value or exception).

The variants of “wait”, where you either give a duration (`wait_for()`) or a target point in time (`wait_until()`) will return either if the future has a valid result or in case the timeout/deadline restriction has been met — therefore they both return `future_status` to allow distinction between those cases.

The last possibility to get notification of the result of the future (valid or exception) is by registering a callback method via `then()`. This is one of the extensions to the `ara::core::Future` over `std::future`.

As you can see, all the possibilities to get access to the future's method result we have discussed (and partly showed in examples) up to now — blocking "get", all "wait" variants and "then" — are **event-driven**. I.e. the event of the arrival of the method result (or an error) leads to either resuming of a blocked user thread or call to a user provided function!

AUTOSAR Binding Implementer Hint

In case `get()` or one of the variants of `wait()` or `then()` is called on a future, there is always the contract with the user of the future, that he gets notified as soon as the method result (valid result or exception) is available (see above). This is our definition of **event-driven** here.

In all of those cases it means, that the binding implementer has to setup a mechanism, which assures that the users callback registered via `Future::then` is called or the blocking wait()/get() call is resumed immediately after the service method result is available or an error during call is detected, respectively.

The notion of "immediately" is obviously a bit fuzzy! The general approach, the `ara::com` design team had in mind, can be best explained with a simple example:

Let's say the underlying transport mechanism used is based on Unix domain sockets (or a comparable fd based I/O). At the point in time the method result is ready, the skeleton side implementation of the service method would write it into a corresponding socket file descriptor.

On the receiving side of the domain socket the proxy instance would have a corresponding descriptor and waiting on it via `select` or `poll`. I.e. writing to the socket on the service side would "immediately" wake up the proxy side Communication Management code waiting in a select/poll call.

As you see "immediately" depends on machine load and latency the used low level mechanism provides.

On the other hand we would not/could not rule out a low level implementation, which favors a polling based mechanism! So instead of propagating OS signals from service instance to proxy instance, which leads to resuming thread execution, the proxy implementation could also cyclically check for data.

If the polling frequency is high enough, this could lead to a rather low and therefore acceptable latency from the pov of the user calling the service method! Such an approach might not make much sense if the underlying transport mechanism is something like file descriptor based read/write I/O, where you then issue reads per descriptor.

But in the realms of shared memory based implementation or async I/O support which allows submitting multiple I/O operations per syscall this might be a valid use case! If you have have an adaptive application with extreme communication load, such a polling based solution on Communication Management implementation level even to fulfill **event-driven** application behavior might make sense, if your platform/chosen transport mechanism provides effective bulk operations, which you can apply with just some acceptable latency costs.

Table 6.8: AUTOSAR Binding Implementer Hint - event-driven implementation

There are of course cases, where the `ara::com` users does not want his application (process) getting activated by some method-call return event at all! Think for a typical RT (real time) application, which must be in total control of its execution. We discussed this RT/polling use case already in the context of event data access already (subsubsection 6.2.3.3). For method calls the same approach applies!

So we did foresee the following usage pattern with regards to `ara::core::Future`: After you have called the service method via the ()-operator, you just use `ara::core::Future::is_ready()` to poll, whether the method call has been finished. This call is defined to be **non-blocking**. Sure, it might involve some syscall/context-switch (for instance to look into some kernel buffers), which is not for free, but it does not block!

After `ara::core::Future::is_ready()` has returned `true`, it is guaranteed that the next call to `ara::core::Future::get()` will NOT block, but immediately return either the valid value or throw an exception in case of error.

6.2.4.3 Canceling Method Result

There may be cases, where you already have called a service method via the ()-operator, which returned you an `ara::core::Future`, but you are not interested in the result anymore.

It could even be the case, that you already have registered a callback via `ara::core::Future::then()` for it. Instead of just let things go and “ignore” the callback, you should tell the Communication Management explicitly.

This might free resources and avoid unnecessary processing load on the binding implementation level. Telling that you are not interested in the method call result anymore is simply done by letting the `ara::core::Future` go out of scope, so that its destructor gets called.

Call of the `dtor` of the `ara::core::Future` is a signal to the binding implementation, that any registered callback for this future shall not be called anymore, reserved/allocated memory for the method call result might be freed and event waiting mechanisms for the method result shall be stopped.

AUTOSAR Binding Implementer Hint
If the user signals, that he is not interested in the service method call result anymore by triggering the `dtor` of the `ara::core::Future`, it obviously makes sense to skip the work to be done for that method call entirely. At the extreme this would mean to propagate the cancellation of the method call up to the service side, which implements the service method. We do intentionally NOT require this, as it might have great influence on the application level implementation side! If we would require/foresee, that an application level service method could be aborted anytime, we would be in the realms of high level application protocols (something like transactional systems) and would put a lot of burden to the service side application developer. This is totally out of scope! But a binding implementer is free to propagate the cancellation up to the service side skeleton, so that the returned method call result from the application level method implementation might directly be discarded on the service/skeleton side! Of course such an efficient implementation would need a proper control channel/protocol to propagate the cancellation from the proxy to the skeleton. SOME/IP protocol f.i. does **not** provide such a mechanism protocol-wise , therefore the method result cannot be discarded already on the skeleton side, in case SOME/IP transport is used. Whether this does really hurt is questionable anyways. The cancellation notification would impose additional network traffic, which would only pay measurably if the saved transmission from skeleton to proxy would have been much more resource intensive.

Table 6.9: AUTOSAR Binding Implementer Hint - cancellation of method call

To trigger the call to the `dtor` you could obviously let the future go out of scope. Depending on the application architecture this might not be feasible, as you already might have assigned the returned `ara::core::Future` to some variable with greater scope.

To solve this, the `ara::core::Future` is default-constructible. Therefore you simply overwrite the returned `ara::core::Future` in the variable with a default constructed instance as is shown in the example below:

```
1  using namespace ara::com;
2
3  Future<Calibrate::Output> calibrateFuture;
4
5  int main() {
6      // some code to acquire handle
7      // ...
8      RadarServiceProxy service(handle);
9      calibrateFuture = service.Calibrate(myConfigString);
10
11     /** ....
12      * Some state changes happened, which render the calibrate method
13      * result superfluous ...
14      *
```

```
15      * We force deletion by resetting our variable to a new default
16      * constructed Future.
17      */
18     calibrateFuture = Future<Calibrate::Output>();
19
20     // go on doing something  ...
21     return 0;
22 }
```

Listing 6.13: Example of discarding a future

6.2.5 Fields

Conceptually a field has — unlike an event — a certain value at any time. That results in the following additions compared to an event:

- if a subscription to a field has been done, “immediately” initial values are sent back to the subscriber in an event-like notification pattern.
- the current field value can be queried via a call to a `Get()` method or could be updated via a `Set()` method.

Note, that all the features a field provides are optionally: In the configuration (IDL) of your field, you decide, whether it has “on-change-notification”, `Get()` or `Set()`. In our example field (see below), we have all three mechanisms configured.

For each field the remote service provides, the proxy class contains a member of a field specific wrapper class. In our example the member has the name `UpdateRate` (of type `fields::UpdateRate`).

Just like the event and method classes the needed field classes of the proxy class are generated inside a specific namespace `fields`, which is contained inside the `proxy` namespace.

The explanation of fields has been intentionally put after the explanation of events and methods, since the field concept is roughly an aggregation of an event with correlated get()/set() methods. Therefore technically we also implement the `ara:com` field representation as a combination of `ara:com` event and method.

Consequently the field member in the proxy is used to

- call `Get()` or `Set()` methods of the field with exactly the same mechanism as regular methods
- access field update notifications in the form of events/event data, which are sent by the service instance our proxy is connected to with exactly the same mechanism as regular events

Let's have a look at the generated field class for our example `UpdateRate` field here:

```
class UpdateRate {
    /**
     * \brief Shortcut for the events data type.
     */
    using FieldType = uint32_t;

    /**
     * \brief See Events for details, as a field contains the possibility for
     * notifications the details of the interfaces described there.
     */
    void Subscribe(size_t maxSampleCount);
    ara::core::Result<size_t> GetFreeSampleCount() const noexcept;
    ara::com::SubscriptionState GetSubscriptionState() const;
    void Unsubscribe();
    void SetReceiveHandler(ara::com::EventReceiveHandler handler);
    void UnsetReceiveHandler();
    void SetSubscriptionStateChangeHandler(ara::com::
    SubscriptionStateChangeHandler handler);
    void UnsetSubscriptionStateChangeHandler();
    template <typename F>
    ara::core::Result<size_t> GetNewSamples(
        F&& f,
        size_t maxNumberOfSamples = std::numeric_limits<size_t>::max());

    /**
     * The getter allows to request the actual value of the service provider.
     *
     * For a description of the future, see the method.
     * It should behave like a Method.
     */
    ara::core::Future<FieldType> Get();

    /**
     * The setter allows to request the setting of a new value.
     * It is up to the Service Provider ro accept the request or modify it.
     * The new value shall be sent back to the requester as response.
     *
     * For a description of the future, see the method.
     * It should behave like a Method.
     */
    ara::core::Future<FieldType> Set(const FieldType& value);
};
```

Listing 6.14: Proxy side UpdateRate Field Class

There is nothing more to be described here. For documentation of the mechanisms of event-like part of the field have a look at subsection 6.2.3 and for documentation of the method-like part of the field have a look at subsection 6.2.4.

6.3 Skeleton Class

The Skeleton class is generated from the service interface description of the AUTOSAR meta model. `ara::com` does standardize the interface of the generated Skeleton class. The toolchain of an AP product vendor will generate a Skeleton implementation class exactly implementing this interface.

The generated Skeleton class is an abstract class. It cannot be instantiated directly, because it does not contain implementations of the service methods, which the service shall provide. Therefore the service implementer has to subclass the skeleton and provide the service method implementation within the subclass.

Note: Equal to the Proxy class the interfaces the Skeleton class has to provide are defined by `ara::com`, a generic (product independent) generator could generate an abstract class or a mock class against which the application developer could implement his service provider application. This perfectly suits the platform vendor independent development of Adaptive AUTOSAR SWCs.

`ara::com` expects skeleton related artifacts inside a namespace "skeleton". This namespace is typically included in a namespace hierarchy deduced from the service definition and its context.

```
class RadarServiceSkeleton {
  public:
    /**
     * Ctor taking instance identifier as parameter and having default
     * request processing mode kEvent.
     */
    RadarServiceSkeleton(ara::com::InstanceIdentifier instanceId,
    ara::com::MethodCallProcessingMode mode =
    ara::com::MethodCallProcessingMode::kEvent);

    /**
     * Ctor taking instance identifier container as parameter and having
     * default request processing mode kEvent.
     * This specifically supports multi-binding.
     */
    RadarServiceSkeleton(ara::com::InstanceIdentifierContainer instanceIds,
    ara::com::MethodCallProcessingMode mode =
    ara::com::MethodCallProcessingMode::kEvent);

    /**
     * Ctor taking instance specifier as parameter and having default
     * request processing mode kEvent.
     */
    RadarServiceSkeleton(ara::core::InstanceSpecifier instanceSpec,
    ara::com::MethodCallProcessingMode mode =
    ara::com::MethodCallProcessingMode::kEvent);

    /**
     * skeleton instances are nor copy constructible.
     */
    RadarServiceSkeleton(const RadarServiceSkeleton& other) = delete;

```

```
33      /**
34       * skeleton instances are nor copy assignable.
35       */
36      RadarServiceSkeleton& operator=(const RadarServiceSkeleton& other) =
        delete;
37
38      /**
39       * The Communication Management implementer should care in his dtor
40       * implementation, that the functionality of StopOfferService()
41       * is internally triggered in case this service instance has
42       * been offered before. This is a convenient cleanup functionality.
43       */
44      ~RadarServiceSkeleton();
45
46      /**
47       * Offer the service instance.
48       * method is idempotent - could be called repeatedly.
49       */
50      void OfferService();
51
52      /**
53       * Stop Offering the service instance.
54       * method is idempotent - could be called repeatedly.
55       *
56       * If service instance gets destroyed - it is expected that the
57       * Communication Management implementation calls StopOfferService()
58       * internally.
59       */
60      void StopOfferService();
61
62      /**
63       * For all output and non-void return parameters
64       * an enclosing struct is generated, which contains
65       * non-void return value and/or out parameters.
66       */
67      struct CalibrateOutput {
68              bool result;
69          };
70
71      /**
72       * For all output and non-void return parameters
73       * an enclosing struct is generated, which contains
74       * non-void return value and/or out parameters.
75       */
76      struct AdjustOutput {
77              bool success;
78              Position effective_position;
79          };
80
81      /**
82       * This fetches the next call from the Communication Management
83       * and executes it. The return value is a ara::core::Future.
84       * In case of an Application Error, an ara::core::ErrorCode is stored
85       * in the ara::core::Promise from which the ara::core::Future
86       * is returned to the caller.
87       * Only available in polling mode.
```

```
88       */
89      ara::core::Future<bool> ProcessNextMethodCall();
90
91      /**
92       * \brief Public member for the BrakeEvent
93       */
94      events::BrakeEvent BrakeEvent;
95
96      /**
97       * \brief Public member for the UpdateRate
98       */
99      fields::UpdateRate UpdateRate;
100
101     /**
102      * The following methods are pure virtual and have to be implemented
103      */
104     virtual ara::core::Future<CalibrateOutput> Calibrate(
105     std::string configuration) = 0;
106     virtual ara::core::Future<AdjustOutput> Adjust(
107     const Position& position) = 0;
108     virtual void LogCurrentState() = 0;
109 };
```

Listing 6.15: RadarService Skeleton

6.3.1 Instantiation

As you see in the example code of the `RadarServiceSkeleton` above, the skeleton class, from which the service implementer has to subclass his service implementation, provides three different `ctor` variants, which basically differ in the way, how the instance identifier to be used is determined.

Since you could deploy many different instances of the same type (and therefore same skeleton class) it is straightforward, that you have to give an instance identifier upon creation. This identifier has to be unique. In the exception-less creation of a service skeleton a static member function Preconstruct checks the provided identifier. The construction token is embedded in the returned `ara::core::Result` if the identifier was unique. Otherwise it returns `ara::core:.ErrorCode`.

If a new instance shall be created with the same identifier, the existing instance needs to be destroyed before.

Exactly for this reason the skeleton class (just like the proxy class) does neither support copy construction nor copy assignment! Otherwise two "identical" instances would exist for some time with the same instance identifier and routing of method calls would be non-deterministic.

The different variants of `ctor`s regarding instance identifier definition reflect their different natures, which are described in section 6.1.

- variant with `ara::com::InstanceIdentifier`: Service instance will be created with exactly one binding specific instance identifier.
- variant with `ara::com::InstanceIdentifierContainer`: Service instance will be created with bindings to multiple distinct instance identifiers. This is mentioned as "multi-binding" throughout this document and also explained in more detail in section 9.3
- variant with `ara::core::InstanceSpecifier`: Service instance will be created with bindings to the instance identifier(s) found after "service manifest" lookup with the given `ara::core::InstanceSpecifier`. Note, that this could also imply a "multi-binding" as the integrator could have mapped the given `ara::core::InstanceSpecifier` to multiple technical/binding specific instance identifiers within the "service manifest".

The second parameter of the `ctor`s of type `ara::com::MethodCallProcessingMode` has a default value and is explained in detail in subsection 6.3.3.

Note: Directly after creation of an instance of the subclass implementing the skeleton, this instance will not be visible to potential consumers and therefore no method will be called on it. This is only possible after the service instance has been made visible with the `OfferService API` (see below).

6.3.2 Offering Service instance

The skeleton provides the method `OfferService()`. After you — as application developer for the service provider side — have instantiated your custom service implementation class and initialized/set up your instance to a state, where it is now able to serve requests (method calls) and provide events to subscribing consumers, you will call this `OfferService()` method on your instance.

From this point in time, where you call it, method calls might be dispatched to your service instance — even if the call to `OfferService()` has not yet returned.

If you decide at a certain point (maybe due to some state changes), that you do not want to provide the service anymore, you call `StopOfferService()` on your instance. The contract here is: After `StopOfferService()` has returned no further method calls will be dispatched to your service instance.

For sanity reasons `ara::com` has the requirement for the AP vendors implementation of the skeleton `dtor`, that it internally does a `StopOfferService()` too, if the instance is currently offered.

So — “stop offer” needs only be called on an instance which lives on and during its lifetime it switches between states, where it is visible and provides its service, and states, where it does not provide the service.

```
1 using namespace ara::com;
2
3 /**
```

```
 * Our implementation of the RadarService -
 * subclass of RadarServiceSkeleton
 */
class RadarServiceImpl;

int main(int argc, char** argv) {
    // read instanceId from commandline
    ara::core::string_view instanceIdStr(argv[1]);
    RadarServiceImpl myRadarService(InstanceIdentifier(instanceIdStr));

    // do some service specific initialization here ....
    myRadarService.init();

    // now service instance is ready -> make it visible/available
    myRadarService.OfferService();

    // go into some wait state in main thread - waiting for AppExecMgr
    // signals or the like ....

    return 0;
}
```

Listing 6.16: Example of RadarService Init and Offer

6.3.3 Polling and event-driven processing modes

Now let's come to the point, where we deliver on the promise to support event-driven and polling behavior also on the service providing side. From the viewpoint of the service providing instance — here our skeleton/skeleton subclass instance — requests (service method or field getter/setter calls) from service consumers may come in at arbitrary points in time.

In a purely event-driven setup, this would mean, that the Communication Management generates corresponding call events and transforms those events to concrete method calls to the service methods provided by the service implementation.

The consequences of this setup are clear:

- general reaction to a service method call might be fast, since the latency is only restricted by general machine load and intrinsic IPC mechanism latency.
- rate of context switches to the OS process containing the service instance might be high and non-deterministic, decreasing overall throughput.

As you see — there are pros and cons for an event-driven processing mode at the service provider side. However, we do support such a processing mode with `ara::com`. The other bookend we do support, is a pure polling style approach. Here the application developer on the service provider side explicitly calls an `ara::com` provided API to process explicitly **one** call event.

With this approach we again support the typical RT-application developer. His application gets typically activated due to a low jitter cyclical alarm.

When his application is active, it checks event queues in a non-blocking manner and decides explicitly how many of those accumulated (since last activation time) events it is willing to process. Again: Context switches/activations of the application process are only accepted by specific (RT) timers. Asynchronous communication events shall **not** lead to an application process activation.

So how does `ara::com` allow the application developer to differentiate between those processing modes? The behavior of a skeleton instance is controlled by the second parameter of its `ctor`, which is of type `ara::com::MethodCallProcessingMode`.

```
/**
 * Request processing modes for the service implementation side
 * (skeleton).
 *
 * \note Should be provided by platform vendor exactly like this.
 */
enum class MethodCallProcessingMode { kPoll, kEvent, kEventSingleThread };
```

That means the processing mode is set for the entire service instance (i.e. all its provided methods are affected) and is fix for the whole lifetime of the skeleton instance. The default value in the `ctor` is set to `kEvent`, which is explained below.

6.3.3.1 Polling Mode

If you set it to `kPoll`, the Communication Management implementation will not call any of the provided service methods asynchronously!

If you want to process the next (assume that there is a queue behind the scenes, where incoming service method calls are stored) pending service-call, you have to call the following method on your service instance:

```
/**
 * This fetches the next call from the Communication Management
 * and executes it. The return value is a ara::core::Future.
 * In case of an Application Error, an ara::core::ErrorCode is stored
 * in the ara::core::Promise from which the ara::core::Future
 * is returned to the caller.
 * Only available in polling mode.
 */
ara::core::Future<bool> ProcessNextMethodCall();
```

We are using the mechanism of `ara::core::Future` again to return a result, which will be fulfilled in the future. What purpose does this returned `ara::core::Future` serve? It allows you to get notified, when the “next request” has been processed. That might be helpful to chain service method calls one after the other. A simple use case for a typical RT application could be:

- RT application gets scheduled.
- it calls `ProcessNextMethodCall` and registers a callback with `ara::core::Future::then()`

- the callback is invoked after the service method called by the middleware corresponding to the outstanding request has finished.
- in the callback the RT application decides, if there is enough time left for serving a subsequent service method. If so, it calls another `ProcessNextMethodCall`.

Sure - this simple example assumes, that the RT application knows worst case runtime of its service methods (and its overall time slice), but this is not that unlikely!

The bool value of the returned `ara::core::Future` is set to `true` by the Communication Management in case there really was an outstanding request in the queue, which has been dispatched, otherwise it is set to `false`.

This is a somewhat comfortable indicator to the application developer, not to call repeatedly `ProcessNextMethodCall` although the request queue is empty. So calling `ProcessNextMethodCall` directly after a previous call returned an `ara::core::Future` with the result set to `false` might most likely do nothing (except that incidentally in this minimal time frame a new request came in).

Note that the binding implementation is free to decide, whether it dispatches the method call event to your service method implementation within the thread context in which you called `ProcessNextMethodCall`, or whether it does spawn a separate thread for this method call.

AUTOSAR Binding Implementer Hint

The explanation up to this point regarding the request processing mode `MethodCallProcessingMode.kPoll` will have a huge impact on the binding implementation!

The fundamental idea of this mode to rule out context switches to a process containing a service implementation caused by Communication Management events (incoming service method calls) has some consequences for AP products based on typical operating systems: There are constraints for the location of the queue, which has to collect the service method call requests until they are consumed by the polling service implementation.

The queue must be realized either outside of the address space of the service provider application or it must be located in a shared memory like location, so that the sending part is able to write directly into the queue. Typical solutions of placing the queue outside of the service provider address space would be

- Kernel space. If the binding implementation would use socket or pipe mechanisms, the kernel buffers being the target of the write-call would resemble the queue. Adapting/configuring maximal sizes of those buffers might in typical OS mean recompiling the kernel.
- User address space of a different binding/Communication Management demon-application. Buffer space allocation for queues allocated within user space could typically be done more dynamic/flexible.

In comparison to a shared memory solution the access from the polling service provider to those queue location might come with higher costs/latency.

Table 6.10: AUTOSAR Binding Implementer Hint - service method call queue

6.3.3.2 Event-Driven Mode

If you set the processing mode to `kEvent` or `kEventSingleThread`, the Communication Management implementation will dispatch events asynchronously to the service method implementations at the time the service call from the service consumer comes in.

Opposed to the `kPoll` mode, here the service consumer implicitly controls/triggers service provider process activations with their method calls!

What is then the difference between `kEvent` and `kEventSingleThread`? `kEvent` means, that the Communication Management implementation may call the service method implementations concurrently.

That means for our example: If — at the same point in time — one call to method `Calibrate` and two calls to method `Adjust` arrive from different service consumers, the Communication Management implementation is allowed to take three threads from its internal thread-pool and do those three calls for the two service methods concurrently.

On the contrary the mode `kEventSingleThread` assures, that on the service instance only one service method at a time will be called by the Communication Management implementation.

That means, Communication Management implementation has to queue incoming service method call events for the same service instance and dispatch them one after the other.

Why did we provide those two variants? From a functional viewpoint only `kEvent` would have been enough! A service implementation, where certain service methods could not run concurrently, because of shared data/consistency needs, could simply do its synchronization (e.g. via `std::mutex`) on its own!

The reason is “efficiency”. If you have a service instance implementation, which has extensive synchronization needs, i.e. would synchronize almost all service method calls anyways, it would be a total waste of resources, if the Communication Management would “spend” N threads from its thread-pool resources, which directly after get a hard sync, sending N-1 of it to sleep.

For service implementations which lie in between — i.e. some methods can be called concurrently without any sync needs, some methods need at least partially synchronization — the service implementer has to decide, whether he uses `kEvent` and does synchronization on top on his own (possibly optimizing latency, responsiveness of his service instance) or whether he uses `kEventSingleThread`, which frees him from synchronizing on his own (possibly optimizing ECU overall throughput).

6.3.4 Methods

Service methods on the skeleton side are abstract methods, which have to be overwritten by the service implementation sub-classing the skeleton. Let's have a look at the `Adjust` method of our service example:

```
/**
 * For all output and non-void return parameters
 * an enclosing struct is generated, which contains
 * non-void return value and/or out parameters.
 */
struct AdjustOutput {
    bool success;
    Position effective_position;
};

virtual ara::core::Future<AdjustOutput> Adjust(
    const Position& position) = 0;
```

Listing 6.17: Skeleton side Adjust method

The IN-parameters from the abstract definition of the service method are directly mapped to method parameters of the skeletons abstract method signature.

In this case it's the `position` argument from type Position, which is — as it is a non-primitive type — modeled as a "const ref"[1].

The interesting part of the method signature is the return type. The implementation of the service method has to return our extensively discussed `ara::core::Future`.

The idea is simple: We do not want to force the service method implementer to signal the finalization of the service method with the simple return of this "entry point" method!

Maybe the service implementer decides to dispatch the real processing of the service call to a central worker-thread pool! This would then be really ugly, when the "entry point" methods return would signal the completion of the service call to the Communication Management.

Then — in our worker thread pool scenario — we would have to block into some kind of wait point inside the service method and wait for some notification from the worker thread, that he has finished and only then we would return from the service method.

In this scenario we would have a blocked thread inside the service-method! From the viewpoint of efficient usage of modern multi-core CPUs this is not acceptable.

The returned `ara::core::Future` contains a structure as template parameter, which aggregates all the OUT-parameters of the service call.

The following two code examples show two variants of an implementation of `Adjust`. In the first variant the service method is directly processed synchronously in the method

[1]The referenced object is provided by the Communication Management implementation until the service method call has set its promise (valid result or error). If the service implementer needs the referenced object beyond that, he has to make a copy.

body, so that an `ara::core::Future` with an already set result is returned, while in the second example, the work is dispatched to an asynchronous worker, so that the returned `ara::core::Future` may not have a set result at return.

```
using namespace ara::com;

/**
 * Our implementation of RadarService
 */
class RadarServiceImpl : public RadarServiceSkeleton {
  public:

    Future<AdjustOutput> Adjust(const Position& position)
    {
        ara::core::Promise<AdjustOutput> promise;

        // calling synchronous internal adjust function, which delivers
    results
        struct AdjustOutput out = doAdjustInternal(
        position,
        &out.effective_position);
        promise.set_value(out);

        // we return a future from an already set promise...
        return promise.get_future();
    }

  private:

    AdjustOutput doAdjustInternal(const Position& position) {
        // ... implementation
    }
}
```

Listing 6.18: Example of returning Future with already set result

As you see in the example above: Inside the body of the service method an internal method is called, which does the work synchronously. I.e. after the return of “doAdjustInternal” in `out` the attributes, which resemble the service methods out-params are set. Then this `out` value is set at the `ara::core::Promise` and then the `Future` created from the `Promise` is returned.

This has the effect that the caller, who gets this `Future` as return, can immediately call `Future::get()`, which would not block, but immediately return the `AdjustOutput`.

Now let's have a look at the asynchronous worker thread variant:

```
using namespace ara::com;

/**
 * Our implementation of the RadarService
 */
class RadarServiceImpl : public RadarServiceSkeleton {

  public:
    Future<AdjustOutput> Adjust(const Position& position)
    {
        ara::core::Promise<AdjustOutput> promise;
        auto future = promise.get_future();

        // asynchronous call to internal adjust function in a new Thread
        std::thread t(
            [this] (const Position& pos, ara::core::Promise prom) {
                prom.set_value(doAdjustInternal(pos));
            },
            std::cref(position), std::move(promise)).detach();

        // we return a future, which might be set or not at this point...
        return future;
    }

  private:
    AdjustOutput doAdjustInternal(const Position& position) {
        // ... implementation
    }
}
```

Listing 6.19: Example of returning Future with possibly unset result

In this example, "doAdjustInternal" is called within a different asynchronous thread. In this case we wrapped the call to "doAdjustInternal" inside a small lambda, which does the job of setting the value to the `Promise`.

6.3.4.1 One-Way aka Fire-and-Forget Methods

"One-way/fire-and-forget" methods on the server/skeleton side do have (like on the proxy side) a simpler signature compared to normal methods. Since there is no feedback possible/needed towards the caller it is a simple void method:

```
    virtual void LogCurrentState() = 0;
```

6.3.4.2 Raising Application Errors

Whenever on the implementation side of a service method, an `ApplicationError` — according to the interface description — is detected, the `Checked Exception`

representing this `ApplicationError` simply has to be stored into the `Promise`, from which the `Future` is returned to the caller:

```
using namespace ara::com;
using namespace com::mycompany::division::radarservice;

/**

    Our implementation of the RadarService
    */
    class RadarServiceImpl : public RadarServiceSkeleton {

public:
Future<CalibrateOutput> Calibrate(const std::string& configuration)
{
ara::core::Promise<CalibrateOutput> promise;
auto future = promise.get_future();

// we check the given configuration arg
if (!checkConfigString(configuration))
{ // given arg is invalid: // assume that in ARXMLs we have ErrorDomain
    with name SpecificErrors // which contains InvalidConfigString error. //
     Note that numeric error code will be casted to ara::core::ErrorCode //
    implicitly. promise.SetError(SpecificErrorsErrc::InvalidConfigString); }

else
{ ... }

// we return a future with a potentially set exception
return future;
}

private:
bool checkConfigString(const std::string& config);

std::string curValidConfig_;
}
```

Listing 6.20: Returning Future with possibly set exception

In this example, the implementation of “Calibrate” detects, that the given configuration string argument is invalid and sets the corresponding exception to the `Promise`.

6.3.5 Events

On the skeleton side the service implementation is in charge of notifying about occurrence of an event. As shown in 6.15 the skeleton provides a member of an event wrapper class per each provided event. The event wrapper class on the skeleton/event provider side looks obviously different than on the proxy/event consumer side.

On the service provider/skeleton side the service specific event wrapper classes are defined within the namespace `event` directly beneath the namespace `skeleton`. Let's have a deeper look at the event wrapper in case of our example event `BrakeEvent`:

```
class BrakeEvent {
  public:
    /**
     * Shortcut for the events data type.
     */
    using SampleType = RadarObjects;

    void Send(const SampleType &data);

    ara::com::SampleAllocateePtr<SampleType> Allocate();

    /**
     * After sending data you loose ownership and can't access
     * the data through the SampleAllocateePtr anymore.
     * Implementation of SampleAllocateePtr will be with the
     * semantics of std::unique_ptr (see types.h)
     */
    void Send(ara::com::SampleAllocateePtr<SampleType> data);
};
```

Listing 6.21: Skeleton side of BrakeEvent class

The using directive — analogue to the Proxy side — just introduces the common name `SampleType` for the concrete data type of the event. We provide two different variants of a "Send" method, which is used to send out new event data. The first one takes a reference to a `SampleType`.

This variant is straight forward: The event data has been allocated somewhere by the service application developer and is given via reference to the binding implementation of `Send()`.

After the call to send returns, the data might be removed/altered on the caller side. The binding implementation will make a copy in the call.

The second variant of 'Send' also has a parameter named "data", but this is now of a different type `ara::com::SampleAllocateePtr<SampleType>`. According to our general approach to only provide abstract interfaces and eventually provide a proposed mapping to existing C++ types (see section 5.3) this pointer type, we introduced here, shall behave like a `std::unique_ptr<T>`.

That roughly means: Only one party can hold the pointer - if the owner wants to give it away, he has to explicitly do it via `std::move`. So what does this mean here? Why do we want to have `std::unique_ptr<T>` semantics here?

To understand the concept, we have to look at the third method within the event wrapper class first:

```
1    ara::com::SampleAllocateePtr<SampleType> Allocate();
```

The event wrapper class provides us here with a method to allocate memory for one sample of event data. It returns a smart pointer `ara::com::SampleAllocateePtr <SampleType>`, which points to the allocated memory, where we then can write an event data sample to. And this returned smart pointer we can then give into an upcoming call to the second version of “Send”.

So — the obvious question would be — why should I let the binding implementation do the memory allocation for event data, which I want to notify/send to potential consumers? The answer simply is: Possibility for optimization of data copies.

The following “over-simplified” example makes things clearer: Let’s say the event, which we talk about here (of type `RadarObjects`), could be quite big, i.e. it contains a vector, which can grow very large (say hundreds of kilobytes). In the first variant of “Send”, you would allocate the memory for this event on your own on the heap of your own application process.

Then — during the call to the first variant of “Send” — the binding implementation has to copy this event data from the (private) process heap to a memory location, where it would be accessible for the consumer. If the event data to copy is very large and the frequency of such event occurrences is high, the sheer runtime of the data copying might hurt.

The idea of the combination of `Allocate()` and the second variant to send event data (`Send(SampleAllocateePtr<SampleType> data)`) is to eventually avoid this copy!

A smart binding implementation might implement the `Allocate()` in a way, that it allocates memory at a location, where writer (service/event provider) and reader (service/event consumer) can both directly access it! So an `ara::com::SampleAllocateePtr<SampleType>` is a pointer, which points to memory nearby the receiver.

Such locations, where two parties can both have direct access to, are typically called “shared memory”. The access to such regions should — for the sake of data consistency — be synchronized between readers and writers.

This is the reason, that the `Allocate()` method returns such a smart pointer with the aspects of single/solely user of the data, which it points to: After the potential writer (service/event provider side) has called `Allocate()`, he can access/write the data pointed to as long as he hands it over to the second send variant, where he explicitly gives away ownership!

This is needed, because after the call, the readers will access the data and need a consistent view of it.

```
using namespace ara::com;

// our implementation of RadarService - subclass of RadarServiceSkeleton
RadarServiceImpl myRadarService;

/**
 * Handler called at occurrence of a BrakeEvent
 */
void BrakeEventHandler() {

    // let the binding allocate memory for event data...
    SampleAllocateePtr<BrakeEvent::SampleType> curSamplePtr =
    myRadarService.BrakeEvent.Allocate();

    // fill the event data ...
    curSamplePtr->active = true;
    fillVector(curSamplePtr->objects);

    // Now notify event to consumers ...
    myRadarService.BrakeEvent.Send(std::move(curSamplePtr));

    // Now any access to data via curSamplePtr would fail -
    // we've given up ownership!
}
```

Listing 6.22: Event Allocate/Send sample

AUTOSAR Binding Implementer Hint

The idea behind the concept of providing a binding specific “Allocate” functionality was greatly driven by the “zero-copy” buzzword.

Having a shared memory based IPC transport mechanism the “zero-copy” axiom might be easily fulfill-able at first glance. The `ara::com::SampleAllocateePtr<SampleType>` mechanism foresees/assumes a hard synchronization between readers/writers anyway, so the challenge in implementation isn’t that big, if the platform provides shared memory concepts anyway.

But in reality you have to be aware of serialization needs (section 9.1), which can ruin any “zero-copy” attempts, which we did also hint at explicitly in subsection 9.1.1.

The work-around would be to either rule out serialization needs between `ara::com` communication partners in an deployment by prescribing compile settings in a way, that exchanged data types are binary compatible or at least to implement some smart checking logic to detect, between which `ara::com` communication partners in fact serialization is not needed.

Table 6.11: AUTOSAR Binding Implementer Hint - zero copy

6.3.6 Fields

On the skeleton side the service implementation is in charge of

- updating and notifying about changes of the value of a field.
- serving incoming `Get()` calls.
- serving incoming `Set()` calls.

As shown in 6.15 the skeleton provides a member of a field wrapper class per each provided field. The field wrapper class on the skeleton/field provider side looks obviously different than on the proxy/field consumer side.

On the service provider/skeleton side the service specific field wrapper classes are defined within the namespace `fields` directly beneath the namespace `skeleton`. Let's have a deeper look at the field wrapper in case of our example event `UpdateRate`:

```
1  class UpdateRate {
2    public:
3
4      using FieldType = uint32_t;
5
6      /**
7       * Update equals the send method of the event. This triggers the
8       * transmission of the notify (if configured) to
9       * the subscribed clients.
10      *
11      * In case of a configured Getter, this has to be called at least
12      * once to set the initial value.
13      */
14     void Update(const FieldType& data);
15
16     /**
17      * Registering a GetHandler is optional. If registered the function
18      * is called whenever a get request is received.
19      *
20      * If no Getter is registered ara::com is responsible for responding
21      * to the request using the last value set by update.
22      *
23      * This implicitly requires at least one call to update after
24      * initialization of the Service, before the service
25      * is offered. This is up to the implementer of the service.
26      *
27      * The get handler shall return a future.
28      */
29     void RegisterGetHandler(std::function<ara::core::Future<FieldType>()>
       getHandler);
30
31     /**
32      * Registering a SetHandler is mandatory, if the field supports it.
33      * The handler gets the data the sender requested to be set.
34      * It has to validate the settings and perform an update of its
35      * internal data. The new value of the field should than be set
36      * in the future.
37      *
```

```
38      * The returned value is sent to the requester and is sent via
    notification to all subscribed entities.
39      */
40     void RegisterSetHandler(std::function<ara::core::Future<FieldType>(
    const FieldType& data)> setHandler);
41 };
```

Listing 6.23: Skeleton side UpdateRate Class

The using directive — again as in the Event Class and on the Proxy side — just introduces the common name `FieldType` for the concrete data type of the field.

We provide an `Update` method by which the service implementer can update the current value of the field.

It is very similar to the simple/first variant of the `Send` method of the event class: The field data has been allocated somewhere by the service application developer and is given via reference to the binding implementation of `Update`. After the call to `Update` returns, the data might be removed/altered on the caller side.

The binding implementation will make a (typically serialized) copy in the call.

In case “on-change-notification” is configured for the field, notifications to subscribers of this field will be triggered by the binding implementation in the course of the `Update` call.

6.3.6.1 Registering Getters

The `RegisterGetHandler` method provides the possibility to register a method implementation by the service implementer, which gets then called by the binding implementation on an incoming `Get()` call from any proxy instance.

The `RegisterGetHandler` method in the generated skeleton does **only** exist in case availability of “field getter” has been configured for the field in the IDL!

Registration of such a “GetHandler” is fully optional! Typically there is no need for a service implementer to provide such a handler. The binding implementation always has access to the latest value, which has been set via `Update`. So any incoming `Get()` call can be served by the Communication Management implementation standalone.

A theoretical reason for a service implementer to still provide a “GetHandler” could be: Calculating the new/current value of a field is costly/time consuming. Therefore the service implementer/field provider wants to defer this process until there is really need for that value (indicated by a getter call). In this case he could calculate the new field value within its “GetHandler” implementation and give it back via the known `ara:com` promise/future pattern.

If you look at the bigger picture, then such a setup with the discussed intention, where the service implementer provides and registers a “GetHandler” will not really make sense, if the field is configured with “on-change-notification”, too.

In this case, new subscribers will get potentially outdated field values on subscription, since updating of the field value is deferred to the explicit call of a “GetHandler”.

You also have to keep in mind: In such a setup, with enabled “on-change-notification” together with a registered “GetHandler” the Communication Management implementation will **not** automatically care for, that the value the developer returns from the “GetHandler” will be synchronized with value, which subscribers get via “on-change-notification” event!

If the implementation of “GetHandler” does not internally call `Update()` with the same value, which it will deliver back via `ara:com` promise, then the field value delivered via “on-change-notification” event will differ from the value returned to the `Get()` call. I.e. the Communication Management implementation will not automatically/internally call `Update()` with the value the “GetHandler” returned.

Bottom line: Using `RegisterGetHandler` is rather an exotic use case and developers should be aware of the intrinsic effect.

Additionally a user provided “GetHandler”, which only returns the current value, which has already been updated by the service implementation via `Update()`, is typically very inefficient! The Communication Management then has to call to user space and to additionally apply field serialization of the returned value at any incoming `Get()` call.

Both things could be totally “optimized away” if the developer does not register a “GetHandler” and leaves the handling of `Get()` calls entirely to the Communication Management implementation.

6.3.6.2 Registering Setters

Opposed to the `RegisterGetHandler` the `RegisterSetHandler` API has to be called by the service implementer in case it exists (i.e. field has been configured with setter support).

The reason, that we decided to make the registration of a “GetHandler” mandatory is simple: We expect, that the server implementation will always need to check the validity of a new/updated field values set by any anonymous client.

A look at the signature of the “SetHandler” `std::function<ara::core::Future<FieldType>(const FieldType& data)>` reveals that the registered handler does get the new value as input argument and is expected to return also a value. The semantic behind this is: In case the “SetHandler” always has to return the effective (eventually replaced/corrected) value. This allows the service side implementer to validate/overrule the new field value provided by a client.

The effective field value returned by the “SetHandler” is implicitly taken over by the Communication Management implementation as if the service implementer had called `Update()` explicitly with the effective value on its own. That means: An explicit `Update()` call within the “SetHandler” is superfluous as the Communication Management would update the field value with the returned value of the “SetHandler” anyways.

6.3.6.3 Ensuring existence of "SetHandler"

The existence of a registered "SetHandler" is ensured by an `ara:com` compliant implementation by returning an unchecked error: If a developer calls `OfferService()` on a skeleton implementation and had not yet registered a "SetHandler" for each of its fields, which has setter enabled, the Communication Management implementation shall return an unchecked error indicating this programming error.

6.3.6.4 Ensuring existence of valid Field values

Since the most basic guarantee of a field is, that it has a valid value at any time, `ara:-com` has to somehow ensure, that a service implementation providing a field has to provide a value **before** the service (and therefore its field) becomes visible to potential consumers, which — after subscription to the field — expect to get initial value notification event (if field is configured with notification) or a valid value on a `Get` call (if getter is enabled for the field).

An `ara::com` Communication Management implementation needs therefore behave in the following way: If a developer calls `OfferService()` on a skeleton implementation and had not yet called `Update()` on any field, which

- has notification enabled
- or has getter enabled but not yet a "GetHandler" registered

the Communication Management implementation shall return an unchecked error indicating this programming error.

Note: The AUTOSAR meta-model supports the definition of such initial values for a field in terms of a so called `FieldSenderComSpec` of a `PPortPrototype`. So this model element should be considered by the application code calling `Update()`.

6.3.6.5 Access to current field value from Get/SetHandler

Since the underlying `field` value is only known to the middleware, the current `field` value is not accessible from the "Get/SetHandler" implementation, which are on application level. If the "Get/SetHandler" needs to read the current `field` value, the skeleton implementation must provide a `field` value replica accessible from application level.

6.4 Runtime

Note: A singleton called `Runtime` may be needed to collect cross-cutting functionalities. Currently there are no requirements for such functionalities, so this chapter is empty. This might change until the 1st release.

7 Data Types on Service Interface level

The following chapter describes the C++ language mapping in `ara::com` of the service interface specific ("user defined") data types. "user defined" here means, that those data types aren't defined/mandated by `ara::com` API itself like e.g. `InstanceIdentifier`, `FindServiceHandle`, `ServiceHandleContainer` or any other data type defined by `ara::com` in its own namespace, but are specifically provided by the user defined service interface description (IDL).

In the AUTOSAR Meta-Model ([5]) `CppImplementationDataType`s have been introduced to support the specifics of the C++14 data type system appropriately.

7.1 Optional data elements

Record elements inside a `StructureImplementationDataType` can be defined as optional inside the meta-model, see [5].

This optionality is represented in the `ara::com` API by the template class `ara::core::Optional`. The serialization of such record elements is based on the Tag-Length-Value principle whereas `StructureImplementationDataType`s without optional record elements do not have to make use of tags.

Details on how this serialization works is specified in [6].

The `ara::core::Optional` template parameter has the `ImplementationDataType` (also `ApplicationDataTypes` are possible) of the record element e.g. `uint32`.

Optional record elements can be used in structures for every service interface element (e.g. Fields, Events and Methods). This optionality is defined on the service interface level.

The structure in 7.1 has the optional declared elements `current` and `health`. These elements are not mandatory present.

The consuming application has to check whether the optional elements contain a value or not during runtime. If an optional element contains a value or not depends on the providing application.

The providing application may set the value or not for this specific instance. The feature of optional contained elements provides forward and backward compatibility of the service interface because new added record elements can just be ignored by old applications.

```
1 /**
2  * \brief Data structure with optional contained values.
3  */
4 struct BatteryState {
5     Voltage_t voltage;
6     Temperature_t temperature;
```

```
7     ara::core::Optional<Current_t> current;
8     ara::core::Optional<Health> health;
9 };
```

Listing 7.1: Definition of BatteryState

The `Skeleton` implementation in 7.2 provides the BatteryState structure defined in 7.1.

The implementation is aware of the optional labeled element `current` but not of the optional labeled element `health` due to a new version of the service interface. Therefore `health` is not set by the `Skeleton` implementation.

```
1  using namespace ara::com;
2
3  class BatteryStateImpl : public BatteryStateSkeleton {
4    public:
5      Future<BatteryState> GetBatteryState() {
6          // no asynchronous call for simplicity
7          ara::core::Promise<BatteryState> promise;
8
9          // fill the data structure
10         BatteryState state;
11         state.voltage = 14;
12         state.temperature = 35;
13         state.current = 0;
14         // state.health is not set and therefore it is not transmitted
15
16         promise.set_value(state);
17         auto future = promise.get_future();
18         return future;
19     }
20 }
```

Listing 7.2: Handling of optional data elements on Skeleton side

The `Proxy` in 7.3 consumes the BatteryState structure defined in 7.1.

The implementation is aware of both optional labeled elements `current` and `health`. Before accessing the value of the optional elements the implementation has to check whether there is really a value contained. Therefore the `optional` API provides two methods: The `operator bool` and the `has_value` method.

```
using namespace ara::com;

int main() {
  // some code to acquire handle
  // ...
  BatteryStateProxy bms_service(handle);
  Future<BatteryState> stateFuture = bms_service.GetBatteryState();
  // Receive BatteryState
  BatteryState state = stateFuture.get();

  // Check the optional contained elements for presence
  if(state.current) {
    //Access the value of the optional element with the optional::operator*
    if(*state.current >= MAX_CURRENT) {
      // do something with this information
    }
  }

  // Check with optional::has_value() method
  if(state.health.has_value()){
    // Access the value of the optional element with the optional::value()
     method
    if(state.health.value() >= BAD_HEALTH) {
      // do something with this information
    }
  }
}
```

Listing 7.3: Handling of optional data elements on Proxy side

8 Raw Data Streaming Interface

8.1 Introduction

The Adaptive AUTOSAR Communication Management is based on Service Oriented communication. This is good for implementing platform independent and dynamic applications with a service-oriented design.

For ADAS applications, it is important to be able to transfer raw binary data streams over Ethernet efficiently between applications and sensors, where service oriented communication (e.g. SOME/IP, DDS) either creates unnecessary overhead for efficient communication, or the sensors do not even have the possibility to send anything but raw binary data.

The Raw Data Binary Stream API provides a way to send and receive Raw Binary Data Streams, which are sequences of bytes, without any data type. They enable efficient communication with external sensors in a vehicle (e.g. sensor delivers video and map data in "Raw data" format). The communication is performed over a network using sockets.

From the ara::com architecture point of view, Raw Data Streaming API is static, i.e. its is not generated. It is part of the ara::com namespace, but is independent of the ara::com middleware services.

8.1.1 Functional description

The Raw Data Binary Stream API can be used in both the client or the server side. The functionality of both client and server allow to send and receive. The only difference is that the server can wait for connections but cannot actively connect to a client. On the other side, the client can connect to a server (that is already waiting for connections) but the client cannot wait for connections.

The usage of the Raw Data Binary Streams API from Adaptive Autosar must follow this sequence:

- As client
  1. Connect: Establishes connection to sensor
  2. ReadData/WriteData: Receives or sends data
  3. Shutdown: Connection is closed.
- As server
  1. WaitForConnection: Waits for incoming connections from clients
  2. ReadData/WriteData: Receives or sends data
  3. Shutdown: Connection is closed and stops waiting for connections.

8.2 Class and Model

8.2.1 Class and signatures

The class `ara::com::raw` defines a RawDataStream class for reading and writing binary data streams over a network connection using sockets. The client side is an object of the class class `ara::com::raw::RawDataStreamClient` and the server side is `ara::com::raw::RawDataStreamServer`

8.2.1.1 Constructor

The constructor takes as input the instance specifier qualifying the network binding and parameters for the instance.

```
RawDataStreamClient(const ara::com::InstanceSpecifier\& instance);
RawDataStreamServer(const ara::com::InstanceSpecifier\& instance);
```

8.2.1.2 Destructor

Destructor of RawDataStream. If the connection is still open, it will be shut down before destroying the RawDataStream object.Destructor of RawDataStream. If the connection is still open, it will be shut down before destroying the RawDataStream object.

```
~RawDataStreamClient();
~RawDataStreamServer();
```

8.2.2 Manifest Model

The manifest defines the parameters of the Raw Data Stream deployment.

The RawDataStreamMapping defines the actual transport that raw data uses in the sub-classes of EthernetRawDataStreamMapping.

The IP address is defined in the attribute communicationConnector (type Ethernet-CommunicationConnector).

The socketOption attribute allows to specify non-formal socket options that might only be valid for specific platforms.

In principle, Raw Data Streaming can use any transport layer but currently only TCP and UDP are supported. The following attributes of the sub-class EthernetRawDataS-treamMapping with type PositiveInteger allow choosing it:

- multicastUdpPort
- tcpPort

- udpPort

At least one of the three previous attributes has to be defined.

The EthernetRawDataStreamMapping also has an attribute regarding security:

- tlsSecureComProps

8.3 Methods of class RawDataStream

Detailed information about the methods of `ara::com::raw::RawDataStream` can be found in document Specification of Communication Management chapter 8.1.3.19 Raw Data Stream API

8.3.1 Timeout parameter

All the methods of RawDataStream have an optional input parameter for the timeout. This argument defines the timeout of the method in milliseconds. The type is `std::chrono::milliseconds`.

If timeout is 0 or not specified the operation will block until it returns.

If timeout is specified is > 0 the method call will return a timeout error if the time to perform the operation exceeds the timeout limit.

8.3.2 Methods

The API methods are synchronous, so they will block until the method returns or until timeout is reached.

8.3.2.1 WaitForConnection

This method is available only in the server side of the Raw Data Stream.

The server side of the Raw Data Stream is ready to be connected from a client. No connection from clients can be established until this method is called in the server.

8.3.2.2 Connect

This method is available only in the client side of the Raw Data Stream.

This method initializes the socket and establishes a connection to the TCP server. In the case of UDP, no connection is established. Incoming and outgoing packets are restricted to the specified address.

The sockets are specified in the manifest which is accessed through the Instance-Specifer provided in the constructor.

```
ara::core:Result<void> Connect();
ara::core:Result<void> Connect(std::chrono::milliseconds timeout);
```

8.3.2.3 Shutdown

This method shuts down communication. It is available from both client and server sides of the Raw Data Stream.

```
ara::core:Result<void> Shutdown();
ara::core:Result<void> Shutdown(std::chrono::milliseconds timeout);
```

8.3.2.4 ReadData

This method reads bytes from the socket connection. The maximum number of bytes to read is provided with the parameter length. The timeout parameter is optional.

```
ara::core::Result<ReadDataResult> ReadData(size_t length);

ara::core::Result<ReadDataResult> ReadData(
        size_t length,
        std::chrono::milliseconds timeout);
```

If the operation worked, it returns a struct with a pointer to the memory containing the read data and the actual number of read bytes.

```
struct ReadDataResult{
        ara::com::SamplePtr<uint8_t> data;
        size_t numberOfBytes;
}
```

In case of an error it retuns an `ara::core::ErrorCode` from `ara::com::RawErrorDomain`:

- Stream Not Connected: If the connection is not yet established
- Communication Timeout: No data was read until the timeout expiration.

8.3.2.5 WriteData

This method writes bytes to the socket connection. The data is provided as a buffer with the data parameter. The number of bytes to write is provided in the length parameter. An optional timeout parameter can also be used.

```
ara::core:Result<size_t> WriteData(
        ara::com::SamplePtr<uint8_t> data,
        size_t length);

ara::core:Result<size_t> WriteData(
        ara::com::SamplePtr<uint8_t> data,
        size_t length,
        std::chrono::milliseconds timeout);
```

If the operation worked, it will return the actual number of bytes written. In case of an error, it will return a `ara::core::ErrorCode`:

- Stream Not Connected: If the connection is not yet established.
- Communication Timeout: No data was written until the timeout expiration.

8.4 Overview

8.4.1 Sequence diagrams

The diagram 8.1 shows the sequence when using the Raw Data Streaming API on the client side.

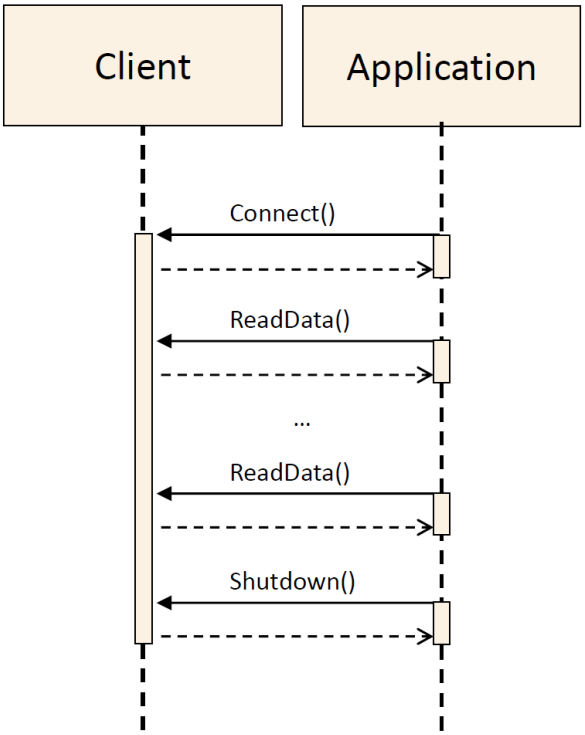

Figure 8.1: Client sequence diagram

The diagram 8.2 shows the sequence when using the Raw Data Streaming API on the server side.

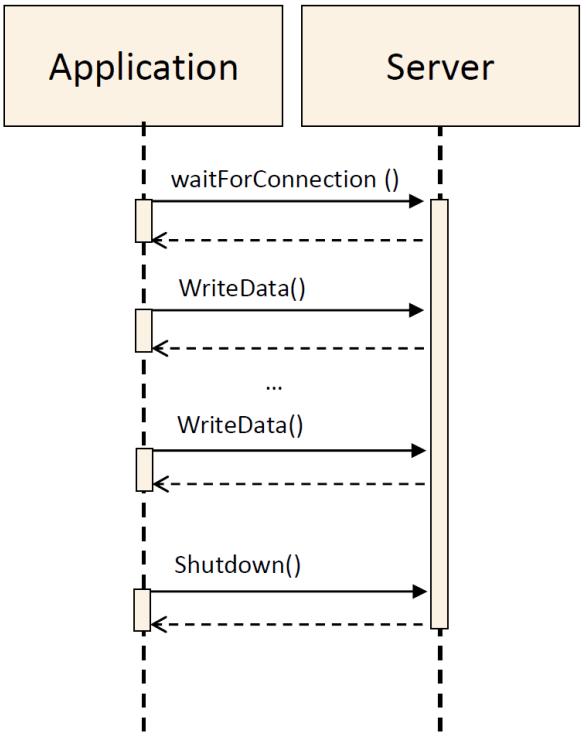

Figure 8.2: Client sequence diagram

Note that the sequences with a client that sends data and a server that reads data are also valid.

8.4.2 Usage

Since the Raw Data Streaming provides an API it is required to have the instances of the RawDataStreamServer or RawDataStreamClient and call the methods according to the sequences described in 8.4.1

8.4.2.1 Example of usage as server

The code 8.1 shows how to use the RawDataStreamServer for sending and receiving data.

```
// NOTE! For simplicity the example does not use ara::core::Result.

#include "ara/core/instance_specifier.h"
#include "raw_data_stream.h"
int main() {
    size_t rval;
    ara::com::raw::RawDataStream::ReadDataResult result;

    // Instance Specifier from model
    ara::core::InstanceSpecifier instspec
    {...}

    // Create RawDataStream Server instance
    ara::com::raw::RawDataStreamServer server{instspec};

    // Wait for incoming connections
    server.WaitForConnection();

    // Read data from the RawData stream in chunks of 10 bytes
    do{
        result = server.ReadData(10);
        rval = result.numberOfBytes;
        if (rval > 0) {
            // assumes the data is printable
            std::cout << "-->" << result.data.get() << std::endl;
        }
    } while (rval > 0);

    // Write data to the RawData stream in chunks of 16 bytes
    int i=0;
    do{
        std::unique_ptr<uint8_t> write_buf (new uint8_t[1024] \{....});
        rval = server.WriteData(std::move(write_buf), 16);
        ++i;
    }while (i<1000);

    // Shutdown RawDataStream connection
```

```
38     server.Shutdown(); return 0;
39 }
```

Listing 8.1: Example of usage as server

8.4.2.2 Example of usage as client

The code 8.2 shows how to use the RawDataStreamClient for sending and receiving data.

```
1  // NOTE! For simplicity the example does not use ara::core::Result.
2
3  #include "ara/core/instance_specifier.h"
4  #include "raw_data_stream.h"
5  int main() {
6      size_t rval;
7      ara::com::raw::RawDataStream::ReadDataResult result;
8
9      // Instance Specifier from model
10     ara::core::InstanceSpecifier instspec
11     {...}
12
13     // Create a RawDataStreamClient instance
14     ara::com::raw::RawDataStreamClient client {instspec};
15
16     // Connect to RawDataStream Server
17     client.Connect();
18
19     // Write data to RawData stream in chunks of 40 bytes
20     int i=0;
21     do {
22         std::unique_ptr<uint8_t> write_buf (new uint8_t[1024]{.....});
23         rval = client.WriteData(std::move(write_buf), 40);
24         ++i;
25     } while (i<1000);
26
27     // Read data from the RawData stream in chunks of 4 bytes
28     do {
29         result = client.ReadData(4);
30         rval = result.numberOfBytes;
31         if (rval > 0){
32             // assumes the data is printable
33             std::cout << "-->" << result.data.get() << std::endl;
34         }
35     } while (rval > 0);
36
37     // Shutdown RawDataStream connection
38     client.Shutdown(); return 0;
39 }
```

Listing 8.2: Example of usage as client

8.4.3 Security

Raw Data Stream communication can be transported using TCP and UDP. Therefore different security mechanisms have to be available to secure the stream communication. Currently the security protocols TLS, DTLS and IPSec are available.

Access control to Raw Data Streams can also be defined by the IAM.

All security functions are configurable in the deployment and mapping model of Raw Data Streaming Interface.

If sensor data must fulfil security requirements, security extensions have to be used.

8.4.4 Safety

The RawDataStream interface only transmits raw data without any data type information. Therefore Raw Data Stream interface cannot provide any data protection, such as E2E protection. If it is required it must be implemented in the application that uses the RawDataStream interface.

8.4.5 Hints for implementers

Implementation of Raw Data Streaming interface should be independent from the underlying Sockets API (e.g. POSIX Sockets).

9 Appendix

AUTOSAR Binding Implementer Hint
This whole section is mainly intended for `ara::com` binding implementers respectively AP product vendors. So instead of enclosing everything in a box, we state it in a preceding comment. However, `ara::com` API users are of course welcomed reading this section, too.

Table 9.1: AUTOSAR Binding Implementer Hint - AP product vendors section

9.1 Serialization

`Serialization` (see [7]) is the process of transforming certain data structures into a standardized format for exchange between a sender and a (possibly different) receiver. You typically have this notion if you transfer data from one network node to another. When putting data on the wire and reading it back, you have to follow exact, agreed-on rules to be able to correctly interpret the data on the receiver side. For the network communication use case the need for a defined approach to convert an in-process data representation into a wire-format and back is very obvious: The boxes doing the communication might be based on different micro-controllers with different endianness and different data-word sizes (16-bit, 32-bit, 64-bit) and therefore employing totally different alignments. In the AUTOSAR CP `serialization` typically plays no role for platform internal/node internal communication! Here the internal in-memory data representation can be directly copied from a sender to a receiver. This is possible, because three assumptions are made in the typical CP product:

- Endianness is identical among all local SWCs.
- Alignment of certain data structures is homogeneous among all local SWCs.
- Data structures exchanged are contiguous in memory.

The first point is maybe a bit pathological as it is most common, that “internal” communication generally means communication on a single- or multi-core MCU or even a multi-processor system, where endianness is identical everywhere. Only if we look at a system/node composed of CPUs made of different micro-controller families this assumption may be invalid, but then you are already in the discussion, whether this communication is still “internal” in the typical sense. The second assumption is valid/acceptable for CP as here a static image for the entire single address space system is built from sources and/or object files, which demands that compiler settings among the different parts of the image are somewhat aligned anyway. The third one is also assured in CP. It is not allowed/possible to model non contiguous data types, which get used in inter-SWC communication.

For the AP things look indeed different. Here the loading of executables during runtime, which have been built independently at different times and have been uploaded to an AP ECU at different times, is definitely a supported use case. The chance, that compiler settings for different `ara::com` applications were different regarding alignment decisions is consequently high. Therefore an AP product (more concrete its IPC bind-

ing implementation) has to use/support serialization of exchanged event/field/method data. How serialization for AP internal IPC is done (i.e. to what generalized format) is fully up to the AP vendor. Also regarding the 3rd point, the AP is less restrictive. So for example the AP supports exchange of `std::map` data types or record like datatypes, which contain variable-length members. These datatypes are generally NOT contiguous in-memory (depending on the allocation strategy). So even if the data contained in the map or records is compatible with the receiver layout wise, a deep copy (meaning collecting contained elements and their references from various memory regions — see [8]) must be done during transfer. Of course the product vendor could apply optimization strategies to get rid of the serialization and de-serialization stages within a communication path:

- Regarding alignment issues, the most simple one could be to allow the integrator of the system to configure, that alignment for certain communication relations can be considered compatible (because he has the needed knowledge about the involved components).
- Another approach common to middleware technology is to verify, whether alignment settings on both sides are equal by exchanging a check-pattern as kind of a init-sequence before first `ara::com` communication call.
- The problem regarding need for deep-copying because of non-contiguous memory allocation could be circumvented by providing vector implementations which care for continuity.

9.1.1 Zero-Copy implications

One thing which typically is at the top of the list of performance optimizations in IPC/middleware implementations is the avoidance of unnecessary copies between sender and the receiver of data. So the buzzword “zero-copy” is widely used to describe this pattern. When we talk about AP, where we have architectural expectations like applications running in separate processes providing memory protection, the typical communication approach needs at least ONE copy of the data from source address space to target address space. Highly optimizing middleware/IPC implementations could even get rid of this single copy step by setting up shared memory regions between communicating `ara::com` components. If you look at 6.22, you see, that we directly encourage such implementation approaches in the API design. But the not so good news is, that if the product vendor does NOT solve the serialization problem, he barely gets benefit from the shared memory approach: If conversions (aka de/serialization) have to be done between communication partners, copying must be done anyhow — so tricky shared memory approaches to aim for “zero-copy” do not pay.

9.2 Service Discovery Implementation Strategies

AUTOSAR Binding Implementer Hint
This whole section is intended for `ara::com` binding implementers respectively AP product vendors.

Table 9.2: AUTOSAR Binding Implementer Hint - AP product vendors section

As laid out in the preceding chapters, `ara::com` expects the functionality of a service discovery being implemented by the product vendor. As the service discovery functionality is basically defined at the API level (see section 6.4) with the methods for `FindService`, `OfferService` and `StopOfferService`, the protocol and implementation details are partially open.

When an AP node (more concretely an AP SWC) offers a service over the network or requires a service from another network node, then service discovery/service registry obviously takes place over the wire. The protocol for service discovery over the wire needs to be completely specified by the used communication protocol. For SOME/IP, this is done in the SOME/IP Service Discovery Protocol Specification [9]. But if an `ara::com` application wants to communicate with another `ara::com` application on the same node within the AP of the same vendor there has to be a local variant of a service discovery available. Here the only difference is, that the protocol implementation for service discovery taking place locally is totally up to the AP product vendor.

9.2.1 Central vs Distributed approach

From an abstract perspective a AP product vendor could choose between two approaches: The first one is a centralist approach, where the vendor decides to have one central entity (f.i. a daemon process), which:

- maintains a registry of all service instances together with their location information
- serves all `FindService`, `OfferService` and `StopOfferService` requests from local `ara::com` applications, thereby either updating the registry (`OfferService`, `StopOfferService`) or querying the registry (`FindService`)
- serves all `SOME/IP SD` messages from the network either updating its registry (`SOME/IP Offer Service` received) or querying the registry (`SOME/IP Find Service` received)
- propagates local updates to its registry to the network by sending out `SOME/IP SD` messages.

The following figure roughly sketches this approach.

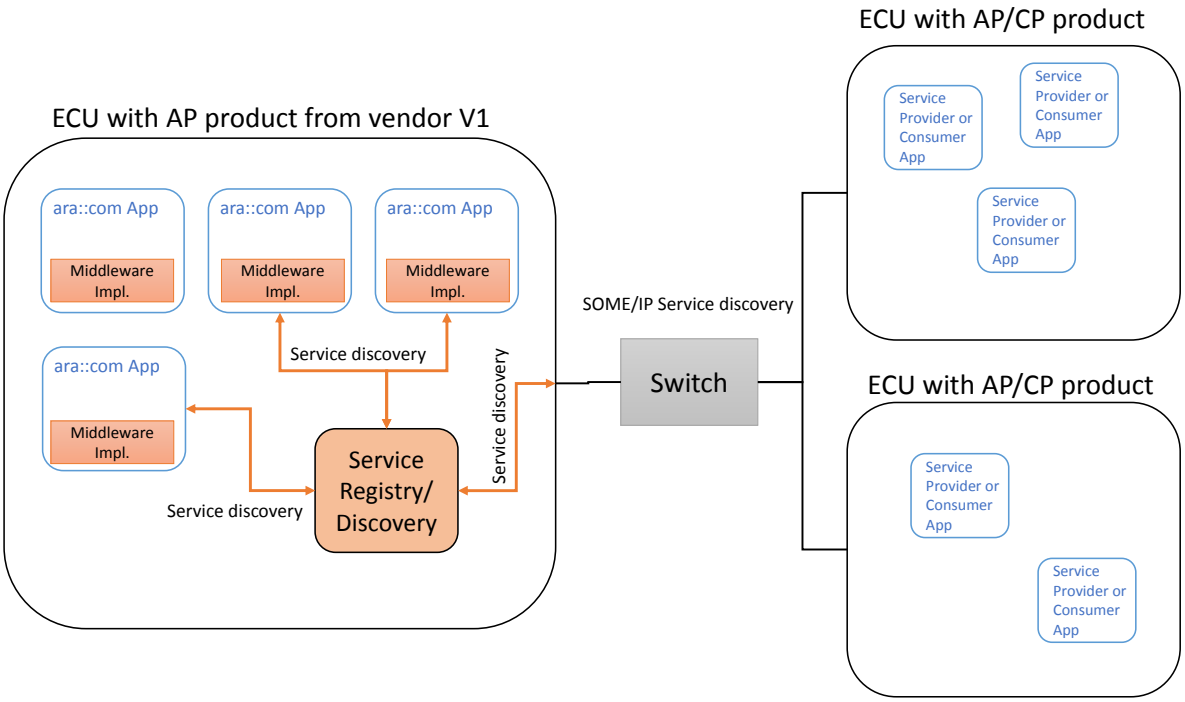

Figure 9.1: Centralized discovery approach

A slightly different — more distributed — approach would be, to distribute the service registry information (availability and location information) among the `ara::com` applications within the node. So for the node local communication use case no prominent discovery demon would be needed. That could be technically reached by having a broadcast-like communication. That means any service offering and finding is propagated to all local `ara::com` applications, so that each application has a local (in process) view of the service registry. There might be a benefit with this approach as local communication might be more flexible/stable as it is not dependent from a single registry demon. However, for the service discovery communication to/from the network a single responsible instance is needed anyhow. Here the distributed approach is not feasible as `SOME/IP SD` requires a fixed/defined set of ports, which just can be provided (in typical operating systems / with typical network stacks) by a single application process.

At the end we also do have a singleton/central instance, with the slight difference, that it is responsible for taking the role as a service discovery protocol bridge between node local discovery protocol and network `SOME/IP SD` protocol. On top of that — since registry is duplicated/distributed among all `ara::com` applications within the node — this bridge also holds a local registry.

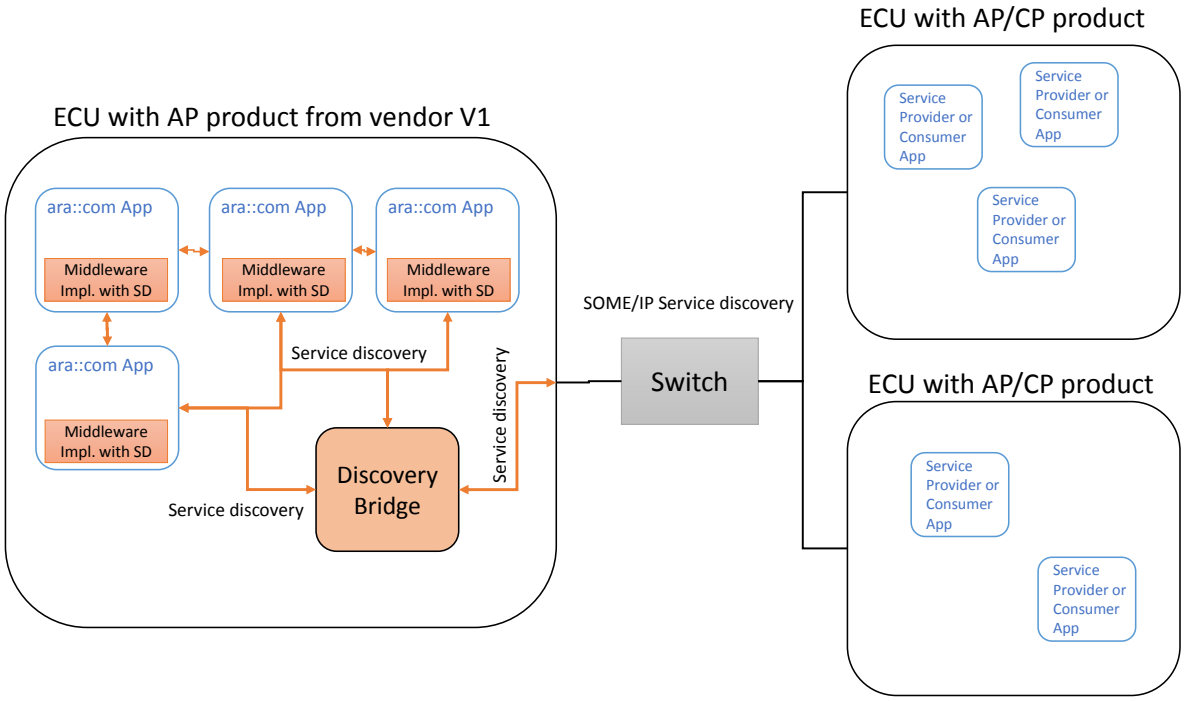

Figure 9.2: Distributed discovery approach

9.3 Multi-Binding implications

As shortly discussed in subsection 6.2.1 `Multi-Binding` describes the solution to support setups, where the technical transport/connection between different instances of a certain proxy class/skeleton class are different. There might be various technical reasons for that:

- proxy class uses different transport/IPC to communicate with different skeleton instances. Reason: Different service instances support different transport mechanisms because of deployment decisions.
- symmetrically it may also be the case, that different proxy instances for the same skeleton instance uses different transport/IPC to communicate with this instance: The skeleton instance supports multiple transport mechanisms to get contacted.

9.3.1 Simple Multi-Binding use case

The following figure depicts an obvious and/or rather simple case. In this example, which only deals with node local (inside one AP product/ECU) communication between service consumers (proxy) and service providers (skeleton), there are two instances of the same proxy class on the service consumer side. You see in the picture, that the service consumer application has triggered a “FindService” first, which returned two handles for two different service instances of the searched service type. The service consumer application has instantiated a proxy instance for each of those handles. Now in this example the instance 1 of the service is located inside the same adaptive application (same process/address space) as the service consumer (proxy instance 1), while the service instance 2 is located in a different adaptive application (different process/address space).

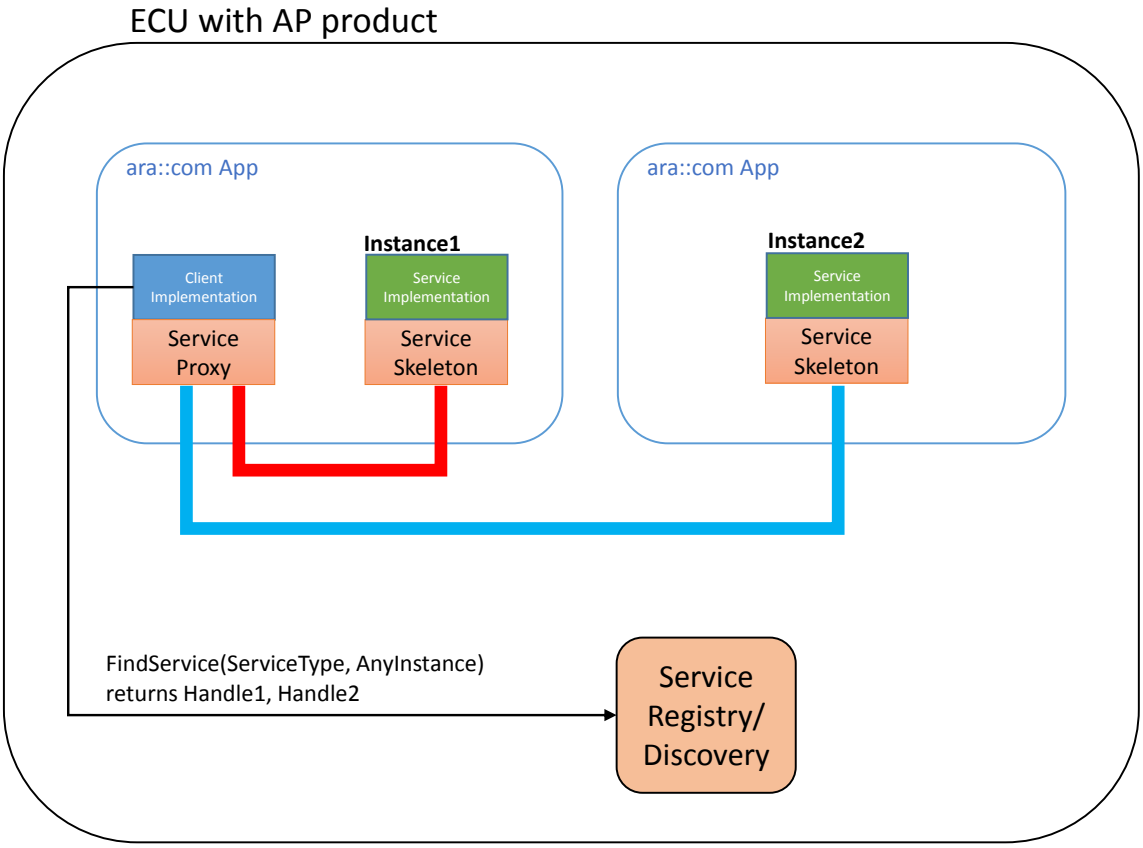

Figure 9.3: Simple Multi-Binding intra AP example

The line symbolizing the transport layer between proxies and skeletons are colored differently in this picture: The instance of the proxy class for instance 1 has a red colored transport layer (binding implementation), while the transport layer for instance 2 is colored blue. They are colored differently because the used technology will be different already on the level of the proxy implementation. At least if you expect that the AP product vendor (in the role as IPC binding implementer) strives for a well performing product!

The communication between proxy instance 1 and the service instance 1 (red) should in this case be optimized to a plain method call, since proxy instance and skeleton instance 1 are contained in ONE process.

The communication between proxy instance 2 and the service instance 2 (blue) is a real IPC. So the actions taken here are of much higher costs involving most likely a variety of syscalls/kernel context switches to transfer calls/data from process of service consumer application to service application (typically using basic technologies like pipes, sockets or shared mem with some signaling on top for control).

So from the service consumer side application developer it is totally transparent: From the vendors `ProxyClass::FindService` implementation he gets two opaque handles for the two service instances, from which he creates two instances of the same proxy class. But “by magic” both proxies behave totally different in the way, they con-

tact their respective service instances. So — somehow there must be some information contained inside this handle, from which the proxy class instance knows which technical transport to choose. Although this use case looks simple at the first look it isn't on the second ... The question is: *Who* writes *When* into the handle, that the proxy instance created from it shall use a direct method/function call instead of a more complex IPC mechanism or vice versa?

At the point in time when instance 1 of the service does register itself via `SkeletonClass::OfferService` at the registry/service discovery, this cannot be decided! Since it depends on the service consumer which uses it later on. So most likely the `SkeletonClass::OfferService` implementation of the AP vendor takes the needed information from the argument (skeleton generated by the AP vendor) and notifies via AP vendor specific IPC the registry/service discovery implementation of the AP vendor. The many "AP vendor" in the preceding sentence were intentional. Just showing, that all those mechanisms going on here are not standardized and can therefore deliberately designed and optimized by the AP vendors. However, the basic steps will remain. So what typically will be communicated from the service instance side to the registry/discovery in the course of `SkeletonClass::OfferService` is the technical addressing information, how the instance could be reached via the AP products local IPC implementation.

Normally there will be only ONE IPC-mechanism used inside one AP product/AP node! If the product vendor already has implemented a highly optimized/efficient local IPC implementation between adaptive applications, which will then be generally used. So — in our example let"s say the underlying IPC-mechanism is unix domain sockets — the skeleton instance 1 would get/create some file descriptor to which its socket endpoint is connected and would communicate this descriptor to the registry/service discovery during `SkeletonClass::OfferService`. Same goes for the skeleton instance 2, just the descriptor is different. When later on the service consumer application part does a `ProxyClass::FindService`, the registry will send the addressing information for both service instances to the service consumer, where they are visible as two opaque handles.

So in this example obviously the handles look exactly the same — with the small difference, that the contained filedescriptor values would be different as they reference distinctive unix domain sockets. So in this case it somehow has to be detected inside the proxy for instance 1, that there is the possibility to optimize for direct method/function calls. One possible trivial trick could be, that inside the addressing information, which skeleton instance 1 gives to the registry/discovery, also the ID of the process (pid) is contained; either explicitly or by including it into the socket descriptor filename. So the service consumer side proxy instance 1 could simply check, whether the PID inside the handle denotes the same process as itself and could then use the optimized path. By the way: Detection of process local optimization potential is a triviality, which almost every existing middleware implementation does today — so no further need to stress this topic.

Now, if we step back, we have to realize, that our simple example here does NOT fully reflect what `Multi-Binding` means. It does indeed describe the case, where two

instances of the same proxy class use different transport layers to contact the service instance, but as the example shows, this is NOT reflected in the handles denoting the different instances, but is simply an optimization! In our concrete example, the service consumer using the proxy instance 1 to communicate with the service instance 1 could have used also the Unix domain socket transport like the proxy instance 2 without any functional losings — only from a non-functional performance viewpoint it would be obviously bad. Nonetheless this simple scenario was worth being mentioned here as it is a real-world scenario, which is very likely to happen in many deployments and therefore must be well supported!

9.3.2 Local/Network Multi-Binding use case

After we have seen a special variant of `Multi-Binding` in the preceding section, we now look at a variant, which can also be considered as being a real-world case. Let"s suppose, we have have a setup quite similar to the one of the preceding chapter. The only difference is now, that the instance 2 of the service is located on a different ECU attached to the same Ethernet network as our ECU with the AP product, where the service consumer (with its proxies for instance 1 and 2) resides. As the standard protocol on Ethernet for AP is SOME/IP, it is expected, that the communication between both ECUs is based on SOME/IP. For our concrete example this means, that proxy 1 talks to service 1 via unix domain sockets (which might be optimized for process local communication to direct method calls, if the AP vendor/IPC implementer did his homework), while the proxy 2 talks to service 2 via network sockets in a SOME/IP compliant message format.

Before someone notes, that this is not true for the typical SOME/IP deployment, because there adaptive SWCs will not directly open network socket connections to remote nodes: We will cover this in more detail here (subsection 9.3.3*), but for now suppose, that this is a realistic scenario. (For other network protocols it might indeed be realistic)*

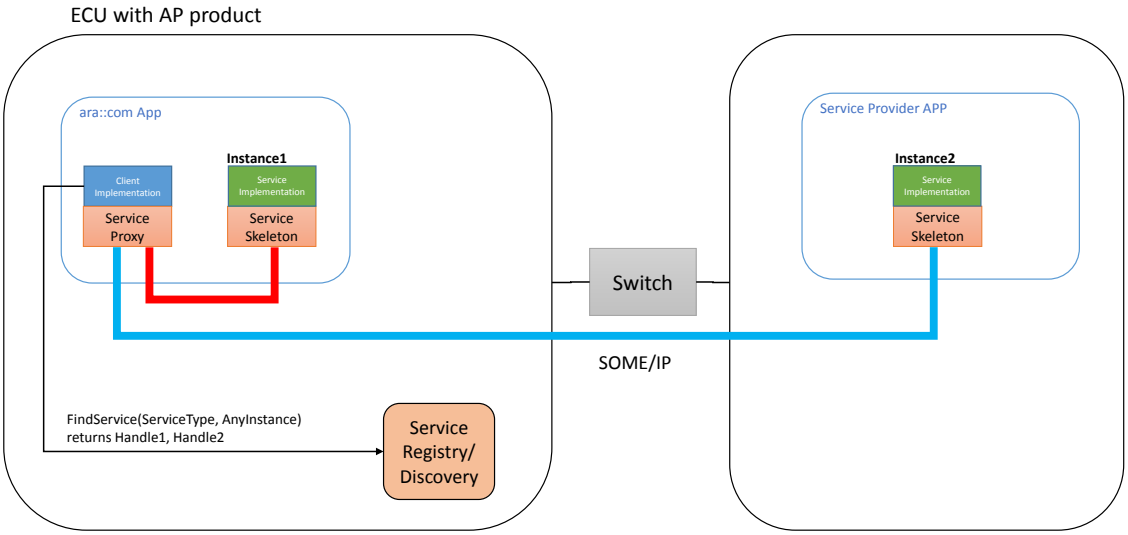

Figure 9.4: Multi-Binding local and network example

So in this scenario the registry/service discovery demon on our AP ECU has seen a service offer of instance 2 and this offer contained the addressing information on IP network endpoint basis. Regarding the service offer of the instance 1 nothing changed: This offer is still connected with some Unix domain socket name, which is essentially a filename. In this example the two handles for instance 1 and 2 returned from `ProxyClass::FindService` internally look very different: Handle of instance 1 contains the information, that it is a Unix domain socket and a name, while handle 2 contains the information, that it is a network socket and an IP address and port number. So — in contrast to our first example (subsection 9.3.1) here we do really have a full blown `Multi-Binding`, where our proxy class `ctor` instantiates/creates two completely different transport mechanisms from handle 1 and handle 2! How this dynamic decision, which transport mechanism to use, made during call of the `ctor`, is technically solved is — again — up to the middleware implementer: The generated proxy class implementation could already contain any supported mechanism and the information contained in the handle is just used to switch between different behavior or the needed transport functionality aka binding could be loaded during runtime after a certain need is detected from the given handle via shared library mechanisms.

9.3.3 Typical SOME/IP Multi-Binding use case

In the previous section we briefly mentioned, that in a typical deployment scenario with SOME/IP as network protocol, it is highly unlikely that an adaptive SWC (i.e. the language and network binding which runs in its context) opens socket connections itself to communicate with a remote service. Why is it unlikely? Because SOME/IP was explicitly designed to use as few ports as possible. The reason for that requirement

comes from low power/low resources embedded ECUs: Managing a huge amount of IP sockets in parallel means huge costs in terms of memory (and runtime) resources. So somehow our AUTOSAR CP siblings which will be main communication partner in an inside vehicle network demand this approach, which is uncommon, compared to non-automotive IT usage pattern for ports.

Typically this requirement leads to an architecture, where the entire SOME/IP traffic of an ECU / network endpoint is routed through one IP port! That means SOME/IP messages originating from/dispatched to many different local applications (service providers or service consumers) are (de)multiplexed to/from one socket connection. In Classic AUTOSAR (CP) this is a straight forward concept, since there is already a shared communication stack through which the entire communication flows. The multiplexing of different upper layer PDUs through one socket is core functionality integrated in CPs SoAd basic software module. For a typical POSIX compatible OS with POSIX socket API, multiplexing SOME/IP communication of many applications to/from one port means the introduction of a separate/central (demon) process, which manages the corresponding port. The task of this process is to bridge between SOME/IP network communication and local communication and vice versa.

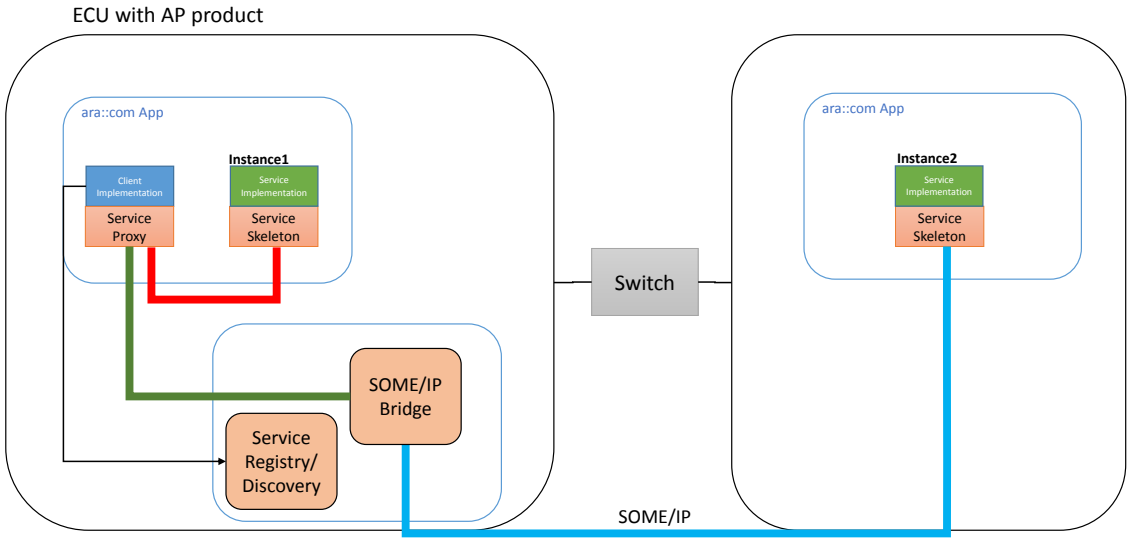

Figure 9.5: SOME/IP Bridge

In the above figure you see, that the service proxy within our `ara::com` enabled application communicates through (green line) a SOME/IP Bridge with the remote service instance 2. Two points which may pop out in this figure:

- we intentionally colored the part of the communication route from app to bridge (green) differently than the part from the bridge to the service instance 2 (blue).
- we intentionally drew a box around the function block service discovery and SOME/IP bridge.

The reason for coloring first part of the route differently from the second one is simple: Both parts use a different transport mechanism. While the first one (green) between the proxy and the bridge uses a fully vendor specific implementation, the second one (blue) has to comply with the SOME/IP specification. “Fully vendor specific” here means, that the vendor not only decides which technology he uses (pipes, sockets, shared mem, ...), but also which serialization format (see section 9.1) he employs on that path. Here we obviously dive into the realm of optimizations: In an optimized AP product, the vendor would not apply a different (proprietary) serialization format for the path denoted with the green line. Otherwise it would lead to an inefficient runtime behavior. First the proxy within the service consumer app would employ a proprietary serialization of the data before transferring it to the bridge node and then the bridge would have to de-serialize and re-serialize it to SOME/IP serialization format! So even if the AP product vendor has a much more efficient/refined serialization approach for local communication, using it here does not pay, since then the bridge is not able to simply copy the data through between internal and external side. The result is, that for our example scenario we eventually do have a `Multi-Binding` setup. So even if the technical transport (pipes, unix domain sockets, shared mem, ...) for communication to other local `ara::com` applications and to the bridge node is the same, the serialization part of the binding differs.

Regarding the second noticeable point in the figure: We drew a box around the service discovery and SOME/IP bridge functionality since in product implementations it is very likely, that it is integrated into one component/running within one (demon) process. Both functionalities are highly related: The discovery/registry part also consists of parts local to the ECU (receiving local registrations/offers and serving local FindService requests) and network related functions (SOME/IP service discovery based offers/finds) , where the registry has to arbitrate. This arbitration in its core is also a bridging functionality.

9.4 `ara::com` and AUTOSAR meta-model relationship

Throughout this document we paid attention to explain `ara::com API` ideas and mechanisms **without** relating to the concrete/specific AP meta-model (the manifest parts of it), which is the basis to formally describe the `SI` signature (and partially the behavior) from which the `ara::com API` artifacts like `ProxyClass` and `SkeletonClass` and data types used in the communication are generated/created. In 6.1 we even introduced an oversimplified/synthetic IDL, just to shield the reader from complexities of the real meta-model/IDL, which wouln’t have added any value at that point.

This chapter shall by no means serve as a thorough explanation of the AUTOSAR meta-model, which is fully described in its own document, but it shall shed some light on the relation between `ara::com` and the meta-model parts described in [5]. So bear in mind, that the following parts are still somewhat high level and try to give a basic understanding of the relationship.

9.4.1 Service Interface

The most important meta-model element from the `ara::com` perspective is the `ServiceInterface`. Most important, because it defines everything signaturewise of an `ara::com` proxy or skeleton. The `ServiceInterface` describes the methods, fields and the methods a service interface consists of and how the signatures of those elements (arguments and data types) look like. So the 6.1 is basically a simplification of meta-model `ServiceInterface` and the real meta-model data type system.

The relationship between the meta-model element `ServiceInterface` and `ara::com` is therefore clear: `ara::com` proxy and skeleton classes get generated from `ServiceInterface`.

9.4.2 Software Component

With software components, the AUTOSAR methodology defines a higher order element than just interfaces. The idea of a software component is to describe a reusable part of software with well defined interfaces. For this the AUTOSAR manifest specification defines a model element `SoftwareComponentType`, which is an abstract element with several concrete subtypes, of which the subtype `AdaptiveApplicationSwComponentType` is the most important one for Adaptive Application software developers. A `SoftwareComponentType` model element is realized by C++ code. Which service interfaces such a component "provides to" or "requires from" the outside is expressed by `ports`. Ports are typed by `ServiceInterface`s. P-ports express that the `ServiceInterface`, which types the port, is provided, while R-ports express, that the `ServiceInterface`, which types the port, is required by the `SoftwareComponentType`.

The figure Figure 9.6 gives a coarse idea, how the model view relates to the code implementation.

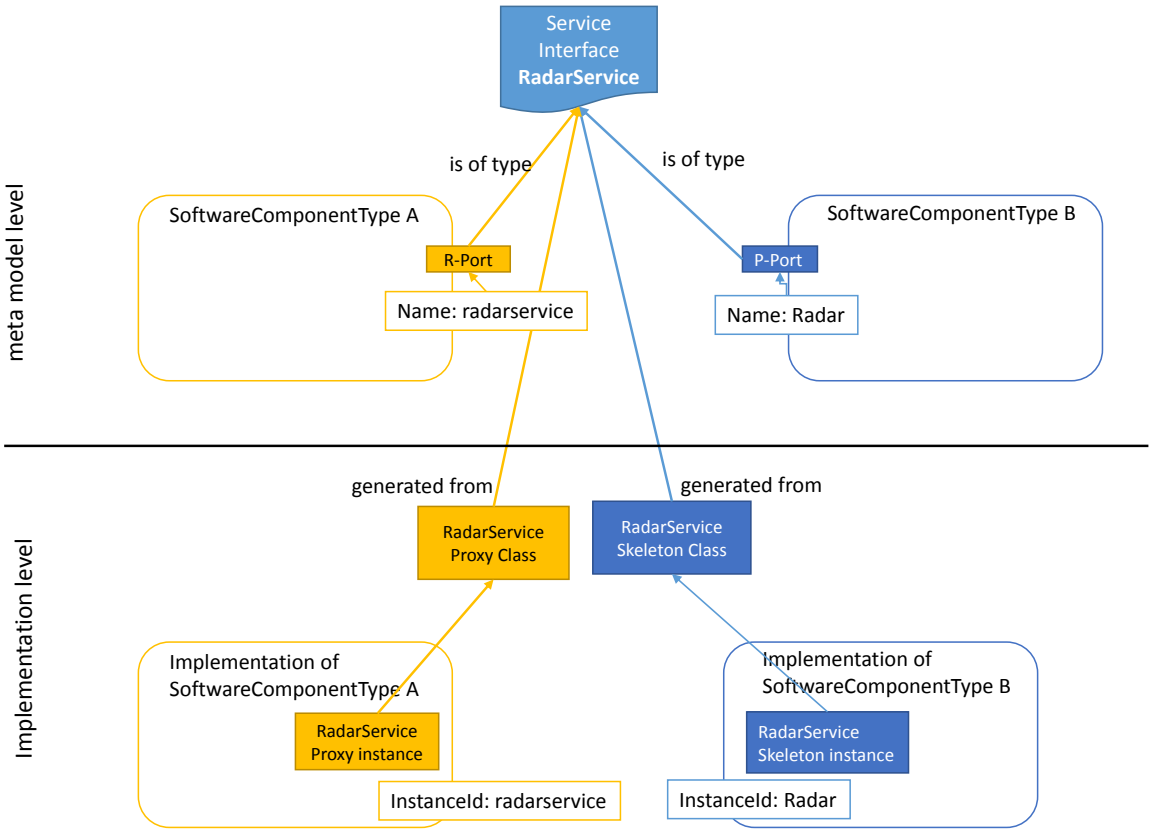

Figure 9.6: meta-model to Implementation

For both of the different `SoftwareComponentType`s A and B from the example in the upper part (meta-model level) a concrete implementation exists on implementation level (lower part in the figure). The realization/implementation of `R-Port` of `SoftwareComponentType` A is based on an instance of `ara::com` proxy class on implementation level, while the `P-Port` implementation of `SoftwareComponentType` B is using an instance of `ara::com` skeleton class. Proxy and skeleton class are generated from the service interface definition `ServiceInterface`, which is referenced by the corresponding ports. In this example it is the `ServiceInterface` "RadarService", which we already use throughout the document.

Such a code fragment, which realizes a `SoftwareComponentType` can obviously be re-used. On C++ implementation level an implementation of an `AdaptiveApplicationSwComponentType` typically boils down to one or several C++ classes. So re-use simply means instantiating this class/those classes in different contexts multiple times. Here we can basically distinguish the following cases:

- Explicit multiple instantiation of the C++ class(es) within Code.
- Implicit multiple instantiation by starting/running the same executable multiple times.

The first case still belongs to the realm of "implementation level".

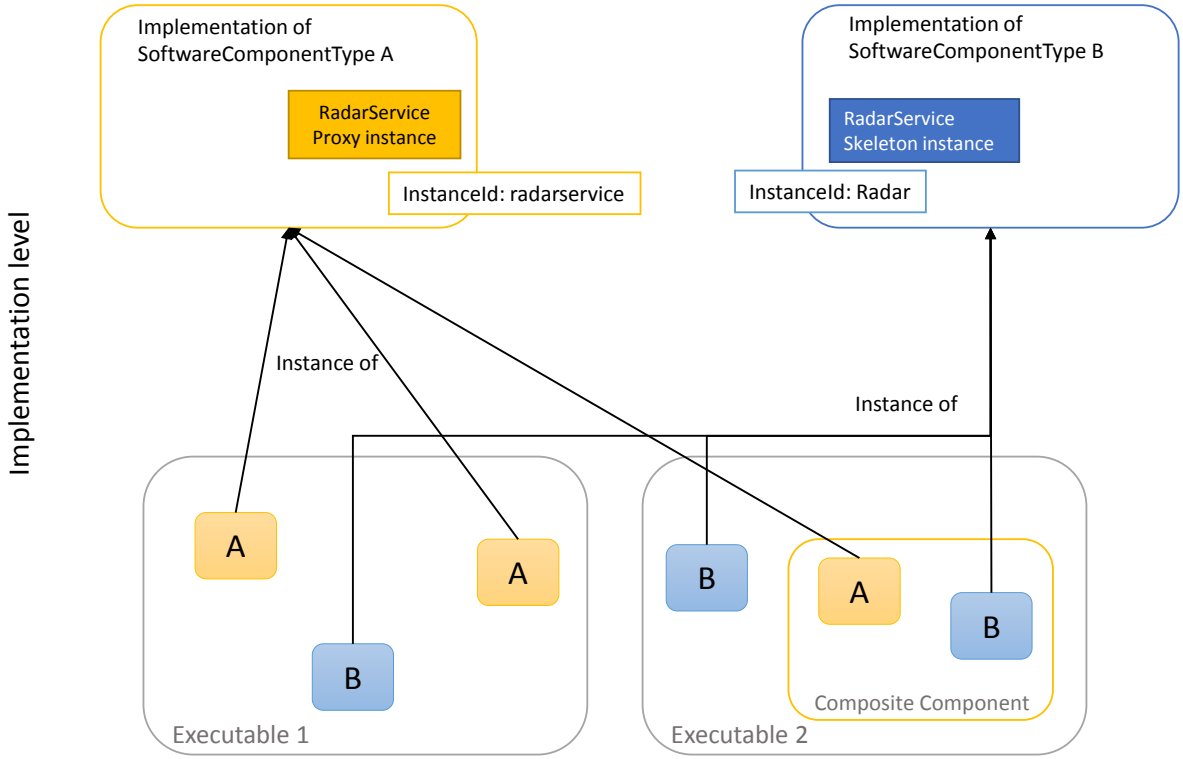

Figure 9.7: Multiple Instantiation in Implementation Contexts

The figure above shows an arbitrary example, where the implementations of A and B are instantiated in different contexts. On the lower left side there is an Executable 1, which directly uses two instances of As impl and one instance of Bs impl. Opposed to that, the right side shows an Executable 2, which "directly" (i.e. on its top most level) uses one instance of Bs impl and an instance of a composite software component, which itself "in its body" again instantiates one instance of As and Bs impl. Note: This natural implementation concept of composing software components from other components to a bigger/composite artefact is fully reflected in the AUTOSAR meta-model in the form of a `CompositionSwComponentType`, which itself is a `SoftwareCompositionType` and allows arbitrary recursive nesting/compositing of software components.

The second case on the other hand belongs to the realm of "deployment level" and shall be clarified in the following sub-chapter.

9.4.3 Adaptive Application/Executables and Processes

Deployable software units within AP are so called Adaptive Applications (the corresponding meta-model element is `AdaptiveAutosarApplication`). Such an Adaptive Application consists of 1..n executeables, which are in turn built up by instantiating `CompositionSwComponentType` (with arbitrary nesting) as described in the previous chapter. Typically integrators then decide, which Adaptive Applications in the form of its 1..n executables they start at all and how many times they start a certain Adaptive

Application/its associated executables. That means for those kind of implicit instantiation no specific code has to be written! Integrators rather have to deal with machine configuration, to configure how many times Applications get started. A started Adaptive Application then turns into 1..n processes (depending on the number of executables it is made of). We call this then the "deployment level".

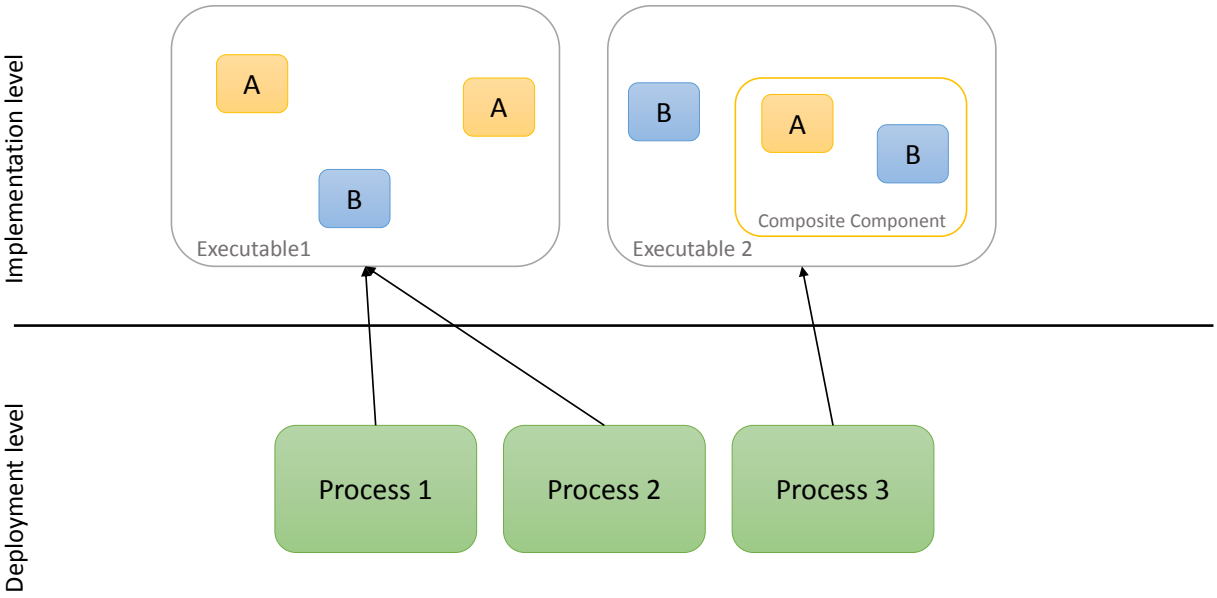

Figure 9.8: Instantiation of Adaptive Applications in Deployment

The figure above shows a simple example, where we have two Adaptive Applications, where each of those exactly consists of one executable. Adaptive Application 1 with Executable 1 is deployed twice, leading to Process 1 and Process 2 after executable start, where Application 2, which consists of Executable 2 is deployed once leading to Process 3 after start.

9.4.4 Usage of meta-model identifiers within `ara::com` based application code

The explanations of meta-model/`ara::com` relation up to this point should help to understand the structure of `instance specifier`s used in `ResolveInstanceIDs` described in 6.4. As described in the previous chapter and depicted in Figure 9.6 the `instance specifier`s relate in a certain way to the corresponding port in the model of the `SoftwareComponentType`. If you followed the previous chapters the **port name** of the model alone isn't sufficient to clearly identify it in its final instantiation, where the same component implementation might be instantiated multiple times in the code and then eventually started multiple times in different processes. Instance IDs obviously have to be assigned to objects, which finally have a distinct identity in an deployment.

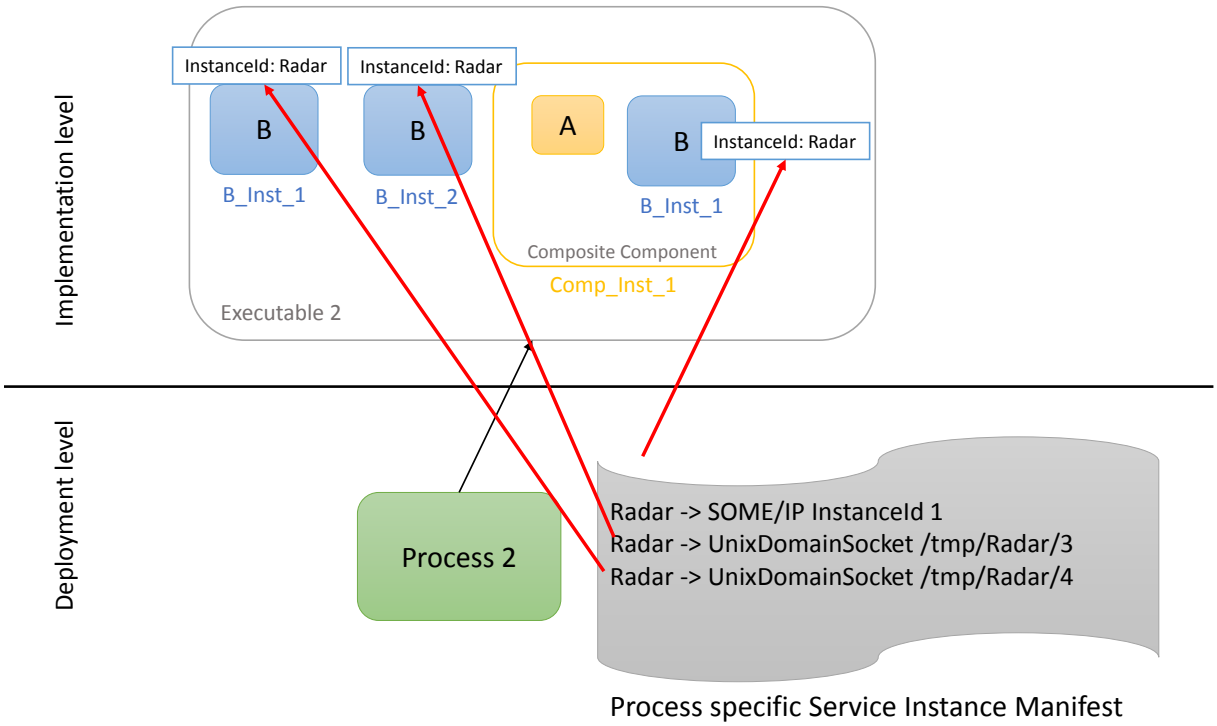

Figure 9.9: InstanceIds in Deployment

The figure above outlines the "problem" with a simple example. Within Executable 2 there are three instantiations of `SoftwareComponentType` B implementation in different contexts (nesting levels). All instances do provide a specific instance of SI RadarService. The integrator, who applies the `Service Instance Manifest` for Process 2 has to do the technical mapping on `ara::com` level. I.e. he has to decide, which technical transport binding is to be used in each of the B instantiations and subsequently also, which technical transport binding specific instance ID. In our example, the integrator wants to provide the SI RadarService via SOME/IP binding and an SOME/IP specific instance ID "1" in the context of the B instantiation, which is nested inside the composite component on the right side, while he decides to provide the SI RadarService via local IPC (Unix domain socket) binding and a Unix domain socket specific instance ID "/tmp/Radar/3" and "/tmp/Radar/4" in the context of the B instantiations on the left side, which are not nested (they are instantiated at "top-level" of the executable). Here it gets obvious, that within the Service Instance Manifest, which allows to specify the mapping of port instantiations within a Process to technical bindings and their concrete instance IDs, the sole usage of the **port name** from the model isn't sufficient to differentiate. To get unique identifiers within an executable (and therefore a process), the nature of nested instantiation and re-use of `SoftwareComponentType`s has to be considered. Every time a `SoftwareComponentType` gets instantiated, its instantiation gets a unique name within its instantiation context. This concept applies to both: C++ implementation level and AUTOSAR meta-model level! In our concrete example this means:

- B instantiations on top level get unique names on their level: "B_Inst_1" and "B_Inst_2"

- B instantiation within the Composite Component Type gets unique name on this level: "B_Inst_1"
- Composite Component instantiation on top level gets unique name on its level: "Comp_Inst_1"
- From the perspective of the executable/process, we therefore have unique identifiers for all instances of B:
  - "B_Inst_1"
  - "B_Inst_2"
  - "Comp_Inst_1::B_Inst_1"

For an Adaptive Software Component developer this then means in a nutshell:

If you construct an `instance specifier` to be transormed via `ResolveInstanceIDs()` into an `ara::com::InstanceIdentifier` or used directly with `FindService()` (R-port side from model perspective) or as `ctor` parameter for a skeleton (P-port side from model perspective), it shall look like:

`<context identifier>/<port name>`

Port name is to be taken from the model, which describes the `AdaptiveApplicationSwComponentType` to be developed. Since you are not necessarily the person who decides where and how often your component gets deployed, you should foresee, that your `AdaptiveApplicationSwComponentType` implementation can be handed over a stringified `<context identifier>`, which you

- either use directly, when constructing `ara::core::InstanceSpecifier` to instantiate proxies/skeleton, which reflect your own component ports.
- "hand over" to other `AdaptiveApplicationSwComponentType` implementations, which you instantiate from your own `AdaptiveApplicationSwComponentType` implementation (that is creating a new nesting level)

Note: Since AUTOSAR AP does **not** prescribe, how the component model on meta-model level shall be translated to (C++) implementation level, component instantiation (nesting of components) and "handing over" of the `<context identifier>` is up to the implementer! It might be a "natural" solution, to solve this by a `<context identifier> ctor` parameter for multi instantiable `AdaptiveApplicationSwComponentType`s.

9.5 Abstract Protocol Network Binding Examples

This chapter presents Abstract Protocol Network Bindings expamples using an `InstanceSpecifier`.

Proxy “FindService” Code Examples :

Figure 9.10: Find Service using abstract network binding

Proxy “ANY” Code Examples :

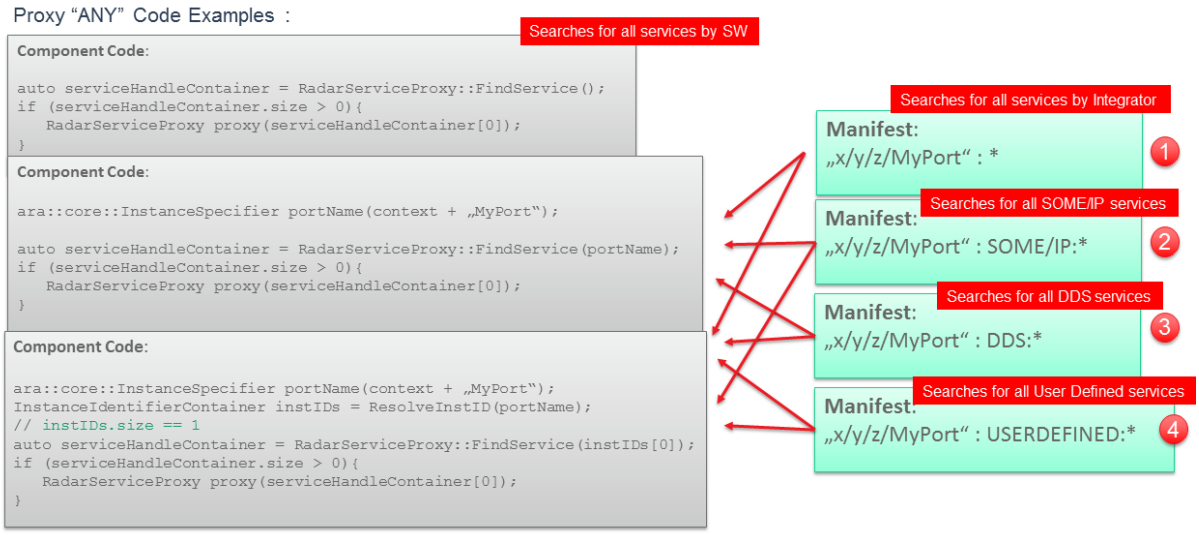

Figure 9.11: Find Service using abstract network binding - ANY

Skeleton “ctor” Code Examples :

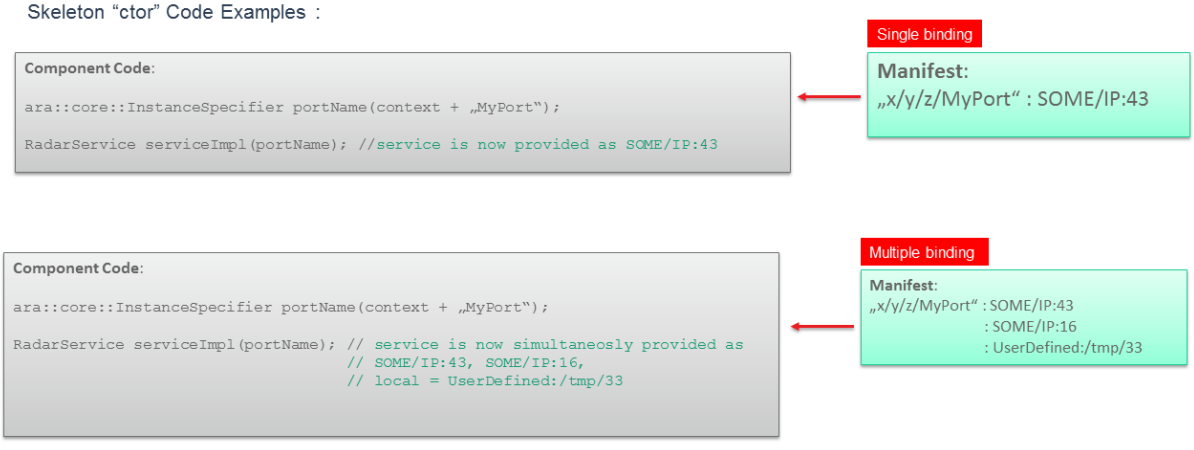

Figure 9.12: Skeleton creation using abstract network bindings

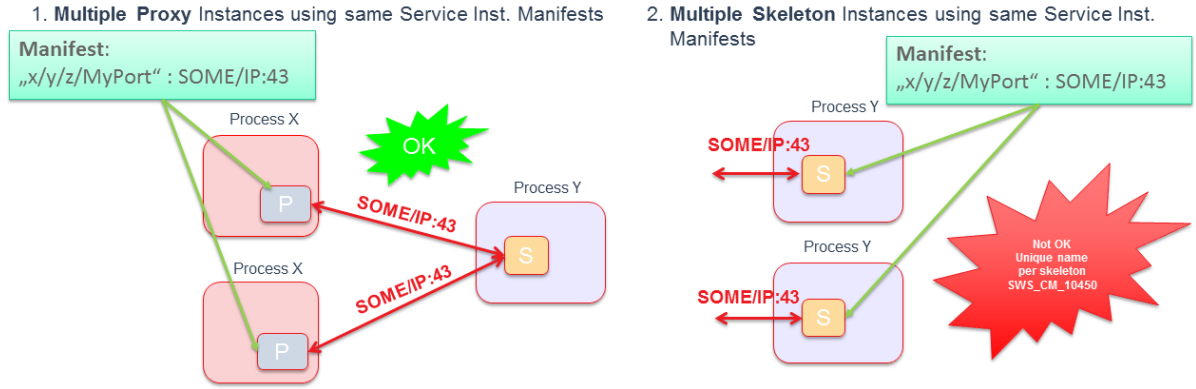

Figure 9.13: Multiple usage of the same service instance manifest for an abstract binding