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

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	17-03

Document Change History			
Date	**Release**	**Changed by**	**Description**
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

Bibliography

[1] Specification of RTE Software
AUTOSAR_SWS_RTE

[2] Glossary
AUTOSAR_TR_Glossary

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

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

[5] N3857: Improvements to std:: future< T> and Related APIs
https://isocpp.org/files/papers/N3857.pdf

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

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

[8] 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 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:

Binding implementer hint

Some very detailed technical information ...

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?

To fight the impression that it was just one instance of the not-invented-here-syndrome, we have to point out, that existing technologies have been evaluated. 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 justified the introduction of a new API to the readers (and ourselves), we go into the details of the API in the upcoming chapters.

Just to point it out (again):

`ara::com` only defines the API signatures and its behaviour 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_Receive`, `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 that are not included in the [2, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
ctor	C++ constructor
dtor	C++ destructor
RT	Realtime
IPC	Inter Process Communication

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 datatype 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, ...

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. The reader might (correctly) object that the notion "as lean as possible" is slightly fuzzy, so this needs some explanation, what our — admittedly rather subjective — understanding of this term 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 wants to be in 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.

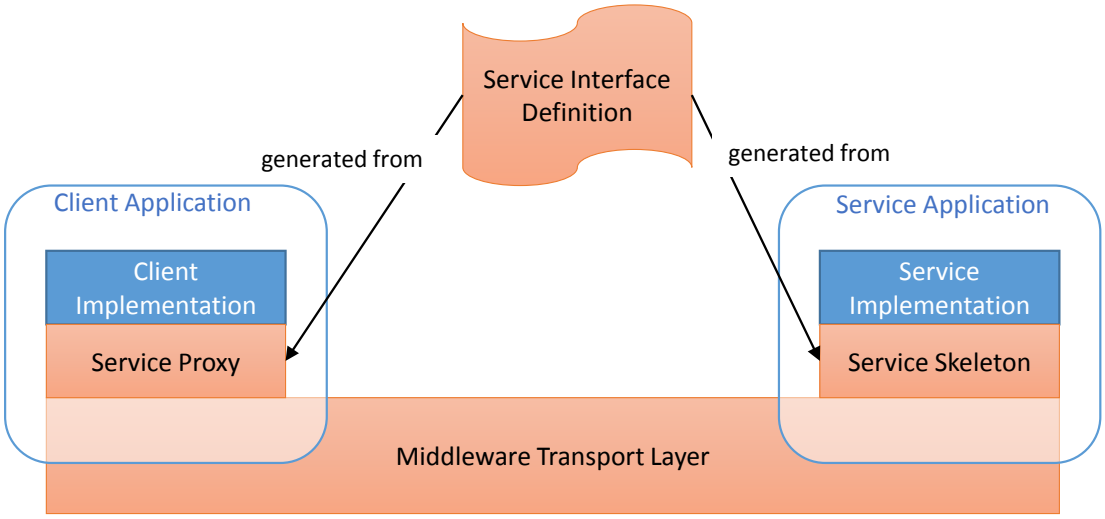

Figure 5.1: Proxy Skeleton Pattern

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 Management 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.

More interesting details regarding the structure of `ara::com` Proxies and Skeletons are shown in section section 6.1 and section 6.2. Regarding this design pattern in general and its role in middleware implementations, see [3, 4].

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 can not 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 6.3).

5.3 Datatype Abstractions

`ara::com` API introduces specific datatypes, 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` 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 behaviour 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.hpp. 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 an product vendor, who wants to implement its own variant: He can validate the functional behaviour of his own implementation against the implementation of the default mapping.

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
  }

  // methods provided by service
  method Calibrate {
    param configuration {
      type: string
      direction: in
    }
    param result {
      type: bool
      direction: out
    }
  }

  method Adjust {
    param target_position {
      type: Position
      direction: in
```

```
    }
    param success {
      type: bool
      direction: out
    }
    param effective_position {
      type: Position
      direction: out
    }
  }
}
```

Figure 6.1: RadarService Definition

So the example service `RadarService` provides an event “BrakeEvent”, which consists of a structure containing a flag and an 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 two 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.

6.1 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 a
     proxy.
7    * This information shall be hidden.
8    * Since the platform vendor is responsible for creation of handles, the
```

```
9     * ctor signature is not given as it is not of interest to the user.
10    */
11   class HandleType {
12     /**
13      * \brief Two ServiceHandles are considered equal if they represent the
       same service instance.
14      *
15      * \param other
16      *
17      * \return
18      */
19     inline bool operator==(const HandleType &other) const;
20     const ara::com::InstanceIdentifier &GetInstanceId() const;
21   };
22
23   /**
24     * StartFindService does not need an explicit version parameter as this
      is internally available in ProxyClass
25     * That means only compatible services are returned.
26     *
27     * \param handler this handler gets called any time the service
      availability of the services matching the given
28     * instance criteria changes. If you use this variant of FindService,
      the Communication Management has to
29     * continuously monitor the availability of the services and call the
      handler on any change.
30     *
31     * \param instance which instance of the service type defined by T shall
       be searched/found. Wildcards may be given.
32     *  Default value is wildcard.
33     *
34     * \return a handle for this search/find request, which shall be used to
       stop the availability monitoring and related
35     * firing of the given handler. (\see StopFindService())
36     **/
37   static ara::com::FindServiceHandle StartFindService(
38       ara::com::FindServiceHandler<RadarServiceProxy::HandleType> handler,
39       ara::com::InstanceIdentifier instance =
40       ara::com::InstanceIdentifier::Any);
41
42   /**
43    * Method to stop finding service request (see above)
44    */
45   static void StopFindService(ara::com::FindServiceHandle handle);
46
47   /**
48    * Opposed to StartFindService(handler, instance) this version is a "one-
      shot" find request, which is
49    * - synchronous, i.e. it returns after the find has been done and a
      result list of matching service instances is
50    * available. (list may be empty, if no matching service instances
      currently exist)
51    * - does reflect the availability at the time of the method call. No
      further (background) checks of availability
52    * are done.
53    *
```

```
54     * \param instance which instance of the service type defined by T shall
       be searched/found. Wildcards may be given.
55     *  Default value is wildcard.
56     */
57    static ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType>
       FindService(
58       ara::com::InstanceIdentifier instance =
59       ara::com::InstanceIdentifier::Any);
60
61    /**
62     * \brief  The proxy can only be created using a specific handle which
       identifies a service.
63     *
64     * This handle can be a known value which is defined at deployment or it
       can be obtained using the ProxyClass::FindService method.
65     *
66     * \param handle The identification of the service the proxy should
       represent.
67     */
68    explicit RadarServiceProxy(HandleType &handle);
69
70    /**
71     * \brief Public member for the BrakeEvent
72     */
73    events::BrakeEvent BrakeEvent;
74
75    /**
76     * \brief Public Field for UpdateRate
77     */
78    fields::UpdateRate UpdateRate;
79
80    /**
81     * \brief Public member for the Calibrate method
82     */
83    methods::Calibrate Calibrate;
84
85    /**
86     * \brief Public member for the Adjust method
87     */
88    methods::Adjust Adjust;
89  };
```

Figure 6.2: RadarService Proxy

6.1.1 Constructor and Handle Concept

As you can see in the figure Figure 6.2 `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 6.1.2. 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.

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 7.2) and to understand what `Multi-Binding` means (see section 7.3). When you have implemented the `Service Discovery` functionality within your AP product and therefore the functionality of subsection 6.1.2 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

To make matters worse: 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 an 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 an 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.

So the question which probably might come up here: Why this indirection, that me as an application developer first have 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 6.1.3). 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` can not know, whether you — in the role as application developer — want always the same proxy instance (explicitly sharing state) or always a new instance each time you trigger 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.

6.1.2 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:

- `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 have the `instance` parameter in common, which allows to either search for an explicit instance of the service or any instance (which is the default parameter value). The synchronous one-off variant `FindService` returns a container of handles (see subsection 6.1.1) 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>)>;
```

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

6.1.2.1 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 following sequence might possibly occur:

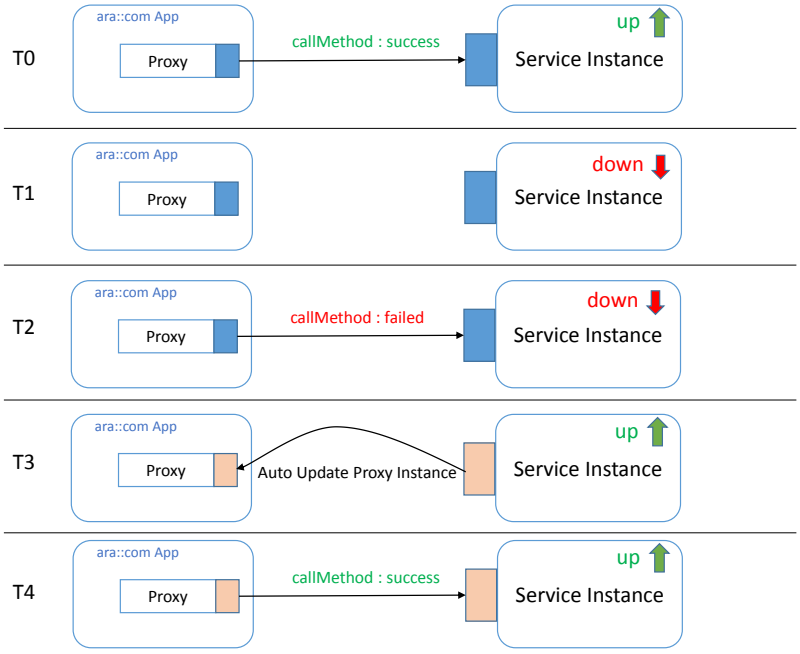

Figure 6.3: Auto Updating of Proxy Instance

Explanation of figure 6.3:

- **T0**: The service consumer may successfully call a service method of that proxy and `GetSubscriptionState()` on subscribed events will return `kSubscribed`.
- **T1**: The service instance goes down, correctly notified via service discovery.

- **T2**: A call of a service method on that proxy will lead to an exception, since the targeted service instance of the call does not exist anymore. Correspondingly `GetSubscriptionState()` on any subscribed event will return `kSubscriptionPending` (see also 6.1.3.2) 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 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:

- checking for Runtime exceptions of service method calls and/or 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:

```
1 /** reference to radar instance, we work with, initialized during startup
      */
2 RadarServiceProxy *myRadarProxy;
3
4 void radarServiceAvailabilityHandler(ServiceHandleContainer<
      RadarServiceProxy::HandleType> curHandles) {
5    for( RadarServiceProxy::HandleType handle : curHandles) {
6       if (handle.GetInstanceId() == myRadarProxy->GetHandle().GetInstanceId
      ()) {
7          /**
```

```
8           * This call on the proxy instance shall NOT lead to an exception,
     regarding
9           * service instance not reachable, since proxy instance should be
    already
10          * auto updated at this point in time.
11          */
12         ara::com::Future<Calibrate::Output> out = myRadarProxy->Calibrate(
    "test");
13         // ... do something with out.
14       }
15    }
16 }
```

Figure 6.4: Access to proxy instance within FindService handler

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 subsection 6.1.1 and section 7.3), 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 with exceptions. 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.

6.1.3 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 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:

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

  /**
   * \brief  The application expects the Communication Management to subscribe the event.
   *
   * The Communication Management shall try to subscribe and resubscribe until \see
   * Unsubscribe() is called explicitly.
   * The error handling shall be kept within the Communication Management.
   *
   * The function return immediately. If the user wants to get notified, when subscription has succeeded, he needs
   * to register a handler via \see SetSubscriptionStateChangeHandler(). This handler gets then called after
   * subscription was successful.
   *
   * \param policy Defines the update policy for the application local cache.
   * \param cacheSize Defines the size of the application local cache.
   *
   */
  void Subscribe(ara::com::EventCacheUpdatePolicy policy, size_t cacheSize);

  /**
   * \brief  query current subscription state.
   *
   * \return current state of the subscription.
   */
  ara::com::SubscriptionState GetSubscriptionState() const;

  /**
   * \brief  Unsubscribe from the service.
   */
  void Unsubscribe();

  /**
   * Setting a receive handler signals the Communication Management implementation to use event style mode.
```

```
38    * I.e. the registered handler gets called asynchronously by the
      Communication Management as soon as new event data
39    * arrives for that event. If user wants to have strict polling behaviour
      , where you decide when to check for
40    * new data via Update() he should NOT register a handler.
41    * Handler may be overwritten anytime during runtime.
42    *
43    * Provided Handler needs not to be re-entrant since the Communication
      Management implementation has to serialize calls
44    * to the handler: Handler gets called once by the MW, when new events
      arrived since the last call to Update().
45    * When application calls Update() again in the context of the receive
      handler, MW must - in case new events arrived
46    * in the meantime - defer next call to receive handler until after the
      previous call to receive handler has been
47    * completed.
48    *
49    **/
50   void SetReceiveHandler(ara::com::EventReceiveHandler handler);
51
52   /**
53    * Remove handler set by SetReceiveHandler()
54    **/
55   void UnsetReceiveHandler();
56
57   /**
58    * Setting a subscription state change handler, which shall get called by
       the Communication Management
59    * implementation as soon as the subscription state of this event has
      changed.
60    * Communication Management implementation will serialize calls to the
      registered handler. If multiple
61    * changes of the subscription state take place during the runtime of a
      previous call to a handler, the
62    * Communication Management aggregates all changes to one call with the
      last/effective state.
63    * Handler may be overwritten during runtime.
64    *
65    **/
66   void
67       SetSubscriptionStateChangeHandler(ara::com::
      SubscriptionStateChangeHandler
68                                         handler);
69
70   /**
71    * Remove handler set by SetSubscriptionStateChangeHandler()
72    **/
73   void UnsetSubscriptionStateChangeHandler();
74
75   /**
76    * \brief Fetch data from the Communication Management buffers and apply
      filter before
77    * writing the samples into the cache.
78    *
79    * \param filter
80    * \parblock
```

```
81      * FilterFunction for the samples.
82      *
83      * This filter will be applied to the deserialized data within the
        context of
84      * the update this function should return true if the sample shall be
        added to
85      * the cache.
86      * \parblockend
87      *
88      * \return True if new values received and those values are NOT ALL
        filtered out,
89      * else false.
90      *
91      * \note If Update is called and the service is not subscribed the method
         will
92      * raise an \see NotSubscribedException.
93      */
94    bool Update(ara::com::FilterFunction<SampleType> filter = {});
95    /**
96      * \brief  Get the container of the samples in the cache that was updated
         by the last call
97      * of \see update.
98      *
99      * The container and referenced data is expected to be stable until
        update is
100     * called again.
101     *
102     * \return Container of SamplePtr
103     */
104   const ara::com::SampleContainer<ara::com::SamplePtr<const SampleType>> &
        GetCachedSamples() const;
105   /**
106     * \brief Explicitly clean the application local cache.
107     *
108     * This should free the references to the data samples which are owned by
         the
109     * Communication Management.
110     * This method only has an effect if policy in the call to \see Subscribe
         has been
111     * set to kNewestN!
112     */
113   void Cleanup();
114 };
```

Figure 6.5: BrakeEvent Class

The data-type of the event data in our example event is `RadarObjects` (see Figure 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.

6.1.3.1 Event Subscription and Cache Semantics

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

```
/**
 * \brief  The application expects the Communication Management to
 subscribe the event.
 *
 * ....
 *
 * \param policy Defines the update policy for the application local
 cache.
 * \param cacheSize Defines the size of the application local cache.
 *
 */
void Subscribe(ara::com::EventCacheUpdatePolicy policy, size_t cacheSize)
 ;
```

This method expects two parameters, `policy` and `cacheSize`. Let's start with the explanation of the concept with the latter one first: With calling the 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 `cacheSize`. The idea is simple (and also related to the AUTOSAR CP concept of queued S/R communication) and possible uses are:

- stability
- interpolation/averaging between a number of events
- rate adoption

Stability is realized with this concept by having an explicit method (see below) to update event data inside the cache. So this local cache decouples the event-wrapper instance from the Communication Management buffers into which the service instance may send its event updates! Updates of the cache status/content is only done explicitly by the user by calling `Update`. This stability assurance shields the application from situations, where during the processing/computing on event data, suddenly the values change, which would lead to inconsistencies.

The support for (arbitrary) sizes of this local cache stems from various use cases. The most prominent ones we had in mind here, are:

- Application, which needs a certain history of events, e.g. for building an average of the last N values and using it in its computation.

- Application, which is scheduled in a — compared to event emitter — much lower frequency. At the point in time it gets active, it processes all events, arrived in the meantime.

Now let's look at the first parameter `policy`. With this parameter you control how the cache is updated if you call the `Update` method (see below). Currently we support two policies:

- `EventCacheUpdatePolicy.kLastN`: With this policy new available events are put in the cache by each call of `Update`. If they do not fit in the cache, older entries (oldest first) are displaced. With this policy the following applies: If the cache has been filled with a certain amount of events, the amount can only remain constant (if no new event has arrived) or get bigger with upcoming `Update` calls (if new events have been arrived).
- `EventCacheUpdatePolicy.kNewestN`: With this policy in each update the cache gets cleared and then filled with the newest arrived events. Even if NO event has arrived since the last call to `Update`, the cache gets cleared/emptied.

Let us go through an example to clarify the different policy behavior: The cache size has been set to 3 and the cache contains (E1, E2, E3) after the last call of `Update`. Now a new event E4 arrives. After the next call of `Update`, the cache contains (E2, E3, E4) in case of `EventCacheUpdatePolicy.kLastN` and just (E4) in case of `EventCacheUpdatePolicy.kNewestN`.
Both policies will behave identical if the number of new arrived events is equal or greater than the cache size.

6.1.3.2 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.
5   */
6  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 (6.1.2.1): Since the Communication Management implementation monitors the availability of the service instances its proxies are connected to anyways, it 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 6.1.2, 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.

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 6.1.2) 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 6.1.2), 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.

6.1.3.3 Event-Driven vs Polling-Based access

As already stated in the previous chapter, there is an explicit interaction needed with `ara::com` API by calling `Update` to fill the local event wrapper specific cache:

```
1  /**
2   * \brief Fetch data from the Communication Management buffers and apply
    filter before
3   * writing the samples into the cache.
4   *
5   * \param filter
6   * \parblock
7   * FilterFunction for the samples.
8   *
9   * This filter will be applied to the deserialized data within the
    context of
10  * the update this function should return true if the sample shall be
    added to
11  * the cache.
12  * \parblockend
13  *
14  * \return True if new values received and those values are NOT ALL
    filtered out,
15  * else false.
16  *
17  * \note If Update is called and the service is not subscribed the method
     will
18  * raise an \see NotSubscribedException.
19  */
20 bool Update(ara::com::FilterFunction<SampleType> filter = {});
```

Just some notes to the method signature: The method takes an optional parameter of a user defined filter function. The filter function has a simple signature — default type mapping of `ara::com` realizes `FilterFunction` with std::function wrapper:

```
using FilterFunction = std::function<bool(const S& sample)>;
```

So in essence, the user provided filter function gets an event (sample) and has to return `true` after checking, if he wants the sample to be put into the local cache. During `Update()`, the Communication Management calls the filter function for each event, which has arrived since the last call to `Update()`. The user provided filter function may or may NOT be called in the same context/thread as the `Update()` call and it should be reentrant.

Binding implementer hint

There are some (optimization) variants, which might be used/considered by the binding implementer: If a lot of newly arrived events exist, which have to be transferred to local cache, binding implementation might decide to spawn multiple threads to check samples with user filter calls in parallel. This might be straightforward as implementation might use thread pools inside receiver process anyway. In case the job of checking 50 new events could be split between two different worker threads and this would speed up things, it is allowed to do. The other most trivial (and expected) optimization is, that no events shall be checked via calls to user provided filter, if the binding implementation will later decide due to configured combination of `EventCacheUpdatePolicy` and `cacheSize`, that it will rule it out anyway. So the user defined filter check shall be the last step before putting event into local cache.

`Update` returns `true` in case new events have been placed into the local cache by the call, `false` otherwise. Note: In case we have the event wrapper configured to `EventCacheUpdatePolicy.kNewestN` and the cache does contain some entries and then `Update()` is called, which deletes the cache and does in this case NOT add any new events, as none have been received in the meantime, the return would be `false`! I.e. the cache content has been changed (deletion) but nothing new has been added — so the returned flag really does not indicate cache-changes but only new additions to the cache!

After you have filled your event specific local cache with event-data via `Update` you typically want access those events. This is done with the following API:

```
  /**
   * \brief  Get the container of the samples in the cache that was updated
    by the last call
   * of \see update.
   *
   * The container and referenced data is expected to be stable until
   update is
   * called again.
   *
   * \return Container of SamplePtr
   */
```

```
10  const ara::com::SampleContainer<ara::com::SamplePtr<const SampleType>> &
      GetCachedSamples() const;
```

You can call this method as many times as you want — as long as you do not call `Update()` in between, the returned collection will always be the same/stable, even if the service side has send out several new events.

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 `Update()` 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 `Update()` is a bit off and vastly inefficient. In this case you explicitly want the Communication Management to notify you 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 `Update()`. 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 `Update()` from within the registered callback!

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 `Update()` 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 `Update()` by the application. If during the execution of this function the user calls `Update()` and during or after this `Update()` 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.

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.

6.1.3.4 Buffering Strategies

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.

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 `Update()`. 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.

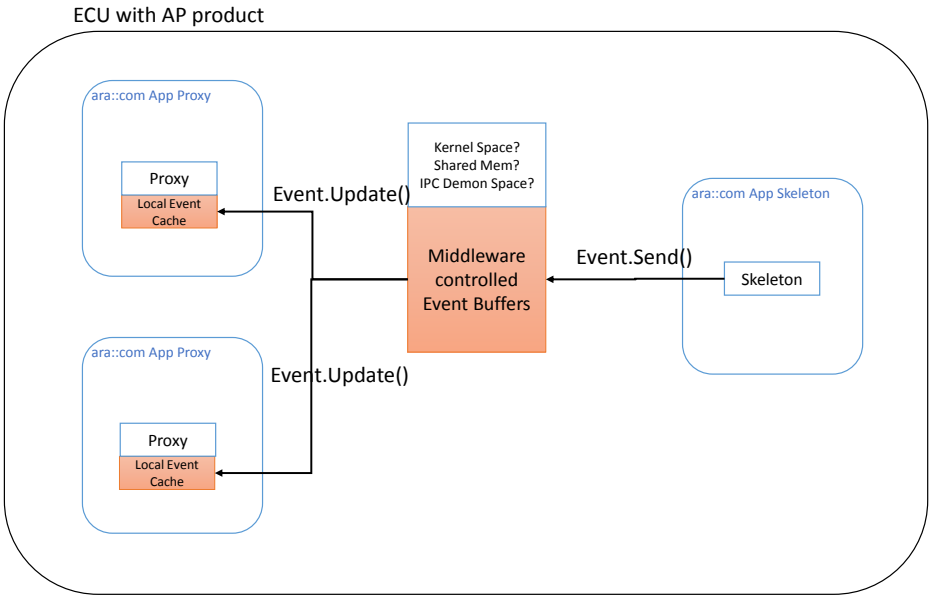

Figure 6.6: Event Buffering Approaches

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) can not 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 6.2.3) 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-Demon 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 (`GetCachedSamples()`) to return a pointer to event data 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 `Update()` 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 7.1) data transfered 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.

6.1.4 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 two methods and the corresponding members have the name `Calibrate` (of type `methods::Calibrate`) and `Adjust` (of type `methods::Adjust`). 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:

```
1  class Adjust {
2   public:
3
4    /**
5     * For all output and non-void return parameters
6     * an enclosing struct is generated, which contains
7     * non-void return value and/or out parameters.
8     */
9    struct Output {
10     bool success;
11     Position effective_position;
12   };
13   /**
14    * \brief Operation will call the method.
15    *
16    * Using the operator the call will be made by the Communication
      Management and a
17    * future returned, which allows the caller to get access to the method
      result.
```

```
18     *
19     * \param[in] target_position See service description.
20     *
21     * \return A future with out-params success(bool) and effective_position(
       Position).
22     */
23   ara::com::Future<Output> operator()(const Position &target_position);
24 };
```

Figure 6.7: 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 is an `ara::com::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::com::Future` in the following subsection.

6.1.4.1 Event-Driven vs Polling access to method results

Similar to the access to event data described in the previous section (subsection 6.1.3), we provide API support for a 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::com::Future`: It is basically an extended version of the C++11/C++14 `std::future` with some extensions borrowed from the C++ proposal N3857 (see [5]). 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::com::Future` work then? Let's have a deeper look at our `ara::com::Future` and the interfaces it provides:

```
1 enum class FutureStatus {
2    ready,
3    timeout
4 };
5
6 template<typename T>
7 class Future {
8   Future();
9   ~Future();
```

```
10
11    /** not copyable */
12    Future(const Future&) = delete;
13    Future& operator=(const Future&) = delete;
14
15    /** movable */
16    Future(Future&&) noexcept;
17    Future& operator=(Future&&) noexcept;
18
19    /**
20     * Following methods are taken from std::future and shall behave
       identically ...
21     */
22    T get();
23    bool valid() const;
24    void wait() const;
25    template< class Rep, class Period >
26    FutureStatus wait_for(const std::chrono::duration<Rep,Period>&
       timeout_duration) const;
27    template <class Clock, class Duration>
28    FutureStatus wait_until(const std::chrono::time_point<Clock,Duration>&
       abs_time) const;
29
30    /**
31     * those methods are borrowed from C++ proposal N3857 and have same
       semantics...
32     */
33
34    /**
35     * Gives the ability to register a function, which gets called in case
36     * the future has a valid result (or exception).
37     *
38     * The function func will get *this future as parameter.
39     *
40     * When func is called, get() will not block.
41     *
42     * func may be called in the context of this call or in the context of
43     * Promise::set_value or set_exception or somewhere else.
44     */
45    template <typename F>
46    auto then(F&& func) -> Future<decltype(func(*this))>;
47
48
49    /** True when the result (or exception) is ready. */
50    bool is_ready() const;
51
52  };
```

Figure 6.8: Future

The standard result-access functions of the future is `get()`, which is blocking as per definition of `std::future::get`. If you as the application developer want to call a service method in a synchronous fashion you simply use the `ara::com::Future`

like this:

```
1
2  using namespace ara::com;
3
4  int main() {
5     // some code to acquire handle
6     // ...
7     RadarServiceProxy service(handle);
8     Future<Calibrate::Output>  callFuture = service.Calibrate(myConfigString
      );
9
10    /* now we do a blocking get(), which will return in case the result (
      valid or exception) is received.
11        if Calibrate could throw an exception and the service has set one,
      it would be thrown by get() */
12    Calibrate::Output callOutput = callFuture.get();
13
14    // process callOutput ...
15    return 0;
16 }
```

Figure 6.9: 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::com::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` according to proposal N3857.

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 time-out/deadline restriction has been met — therefore they both return `FutureStatus` 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 std::future according to proposal N3857, which is shown in the following example:

```
1
2  using namespace ara::com;
3
4  void myCalibrateOKHandler(Calibrate::Output out) {
5     // ... do something
6  }
7
8  void myCalibrateErrorHandler() {
9     // ... do something
10 }
11
12 int main() {
13    // some code to acquire handle
14    // ...
15    RadarServiceProxy service(handle);
16    Future<Calibrate::Output>  callFuture = service.Calibrate(myConfigString
      );
17
18    /* now we register a lambda, which calls our OK or Error handler,
      depending on the
19        outcome of the Calibrate service method call. */
20    callFuture.then([] (Future<Calibrate::Output> f) {
21       try{
22         // get() is NOT blocking here as per definition!
23         myCalibrateOKHandler(f.get());
24       } catch (... ) {
25          myCalibrateErrorHandler();
26       });
27
28    // go on doing something in parallel  ...
29    return 0;
30 }
```

Figure 6.10: Future::then() usage sample

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 an user provided function!

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.

There are of course cases, where the `ara::com` users does not want his application (process) getting activated by some asynchronous 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 6.1.3.3). For method calls the same approach applies! So we did foresee the following usage pattern with regards to `ara::com::Future`: After you have called the service method via the ()-operator, you just use `ara::com::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::com::Future::is_ready()` has returned `true`, it is guaranteed that the next call to `ara::com::Future::get()` will NOT

block, but immediately return either the valid value or throw an exception in case of error.

6.1.4.2 Canceling Method Result

There may be cases, where you already have called a service method via the ()-operator, which returned you an `ara::com::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::com::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::com::Future` go out of scope, so that its destructor gets called. Call of the `dtor` of the `ara::com::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.

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::com::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 can not 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.

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::com::Future` to some variable with greater scope. To solve this, the `ara::com::Future` is default-constructible. Therefore you simply

overwrite the returned `ara::com::Future` in the variable with a default constructed instance as is shown in the example below:

```
1
2  using namespace ara::com;
3
4  Future<Calibrate::Output> calibrateFuture;
5
6  int main() {
7     // some code to acquire handle
8     // ...
9     RadarServiceProxy service(handle);
10    calibrateFuture = service.Calibrate(myConfigString);
11
12    /* ....
13        Some state changes happened, which render the
14        calibrate method result superfluous ...
15        We force deletion by resetting our variable to a
16        new default constructed Future. */
17
18    calibrateFuture = Future<Calibrate::Output>();
19
20    // go on doing something  ...
21    return 0;
22 }
```

Figure 6.11: Example of discarding a future

6.1.5 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 subscribtion 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(ara::com::EventCacheUpdatePolicy policy, size_t cacheSize);
  ara::com::SubscriptionState GetSubscriptionState() const;
  void Unsubscribe();
  void SetReceiveHandler(ara::com::EventReceiveHandler handler);
  void UnsetReceiveHandler();
  void SetSubscriptionStateChangeHandler(ara::com::SubscriptionStateChangeHandler handler);
  void UnsetSubscriptionStateChangeHandler();
  bool Update(ara::com::FilterFunction<FieldType> filter = {});
  const ara::com::SampleContainer<ara::com::SamplePtr<const FieldType>> & GetCachedSamples() const;
  void Cleanup();
  /**
   * 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::com::Future<FieldType> Get();
  /**
   * The getter 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::com::Future<FieldType> Set(const FieldType& value);
};
```

Figure 6.12: 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 6.1.3 and for documentation of the method-like part of the field have a look at subsection 6.1.4.

6.2 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 instance,
                       ara::com::MethodCallProcessingMode mode = ara::com::
   MethodCallProcessingMode::kEvent);

  /**
   * The Communication Management implementer should care in his dtor
     * implementation, that the functionality of StopOfferService()
     * is internally triggered in case this service instance has
     * been offered before. This is a convenient cleanup functionality.
   **/
  ~RadarServiceSkeleton();

  /**
   * Offer the service instance.
   * method is idempotent - could be called repeatedly.
   **/
  void OfferService();

  /**
   * Stop Offering the service instance.
   * method is idempotent - could be called repeatedly.
   *
   * if service instance gets destroyed - it is expected that the
   Communication Management implementation
   * calls StopOfferService() internally.
```

```
30     **/
31   void StopOfferService();
32
33   /**
34    * For all output and non-void return parameters
35    * an enclosing struct is generated, which contains
36    * non-void return value and/or out parameters.
37    */
38   struct CalibrateOutput {
39     bool result;
40   };
41   /**
42    * For all output and non-void return parameters
43    * an enclosing struct is generated, which contains
44    * non-void return value and/or out parameters.
45    */
46   struct AdjustOutput {
47     bool success;
48     Position effective_position;
49   };
50
51   /**
52    * This fetches the next call from the Communication Management and
      executes it.
53    * Only available in polling mode. In event mode it shall throw an
      exception.
54    */
55   ara::com::Future<bool> ProcessNextMethodCall();
56   /**
57    * \brief Public member for the BrakeEvent
58    */
59   events::BrakeEvent BrakeEvent;
60
61   /**
62    * \brief Public member for the UpdateRate
63    */
64   fields::UpdateRate UpdateRate;
65
66   // All methods are pure virtual and have to be implemented
67   virtual ara::com::Future<CalibrateOutput> Calibrate(
68       std::string configuration) = 0;
69   // All methods are pure virtual and have to be implemented
70   virtual ara::com::Future<AdjustOutput> Adjust(
71       const Position& position) = 0;
72 };
```

Figure 6.13: RadarService Skeleton

6.2.1 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 a `ctor` with a parameter of type `ara::com::InstanceIdentifier`. Since you could deploy many different instances of the same type (and therefore same skeleton class) this is straightforward, that you have to give an instance identifier upon creation. This identifier has to be unique — creating two instances in a way that they would exist at the same time with the same instance identifier will raise an exception. If a new instance shall be created with the same identifier, the existing instance needs to be destroyed before.

The second parameter of the `ctor` of type `ara::com::MethodCallProcessingMode` has a default value and is explained in detail in subsection 6.2.3.

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).

6.2.2 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 RadarService - subclass of RadarServiceSkeleton
4 class RadarServiceImpl;
5
6 int main(int argc, char** argv) {
7   // read instanceId from commandline
```

```
8   std::string instanceIdStr(argv[1]);
9   RadarServiceImpl myRadarService(InstanceIdentifier(instanceIdStr));
10
11  // do some service specific initialization here ....
12  myRadarService.init();
13
14  // now service instance is ready -> make it visible/available
15  myRadarService.OfferService();
16
17  // go into some wait state in main thread - waiting for AppExecMgr
18  // signals or the like ....
19
20  return 0;
21 }
```

Figure 6.14: RadarService Init and Offer sample

6.2.3 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 a 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.

6.2.3.1 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.
   * Only available in polling mode. In event mode it shall throw an
   exception.
   */
ara::com::Future<bool> ProcessNextMethodCall();
```

We are using the mechanism of `ara::com::Future` again to return a result, which will be fulfilled in the future. What purpose does this returned `ara::com::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::com::Future::then()`
- the callback is invoked after the service method called by the midleware 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::com::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::com::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.

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.

6.2.3.2 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 implemen-

tations 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).

6.2.4 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::com::Future<AdjustOutput> Adjust(
      const Position& position) = 0;
```

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::com::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::com::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 body, so that an `ara::com::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::com::Future` may not have a set result at return.

```
1  using namespace ara::com;
2
3  // our implementation of RadarService
4  class RadarServiceImpl : public RadarServiceSkeleton {
5
6  public:
7    Future<AdjustOutput> Adjust(const Position& position)
8    {
9      Promise<AdjustOutput> promise;
10      // calling synchronous internal adjust function, which delivers
       results
11     struct AdjustOutput out = doAdjustInternal(position, &out.
       effective_position);
12     promise.set_value(out);
13     //we return a future from an already set promise...
14     return promise.get_future();
15   }
16
17 private:
18   AdjustOutput doAdjustInternal(const Position& position) {
19     // ... implementation
20   }
21
```

[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.

```
}
```

Figure 6.15: 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 `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 RadarService
class RadarServiceImpl : public RadarServiceSkeleton {

public:
  Future<AdjustOutput> Adjust(const Position& position)
  {
    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, Promise prom) { prom.set_value(
   doAdjustInternal(pos));},
      std::cref(position), std::move(promise));

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

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

Figure 6.16: 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`.

6.2.5 Events

On the skeleton side the service implementation is in charge of notifying about occurrence of an event. As shown in Figure 6.13 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:

  /**
   * \brief 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);

};
```

Figure 6.17: 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 explic-

itly 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:

```
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!

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!

}
```

Figure 6.18: Event Allocate/Send sample

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 7.1), which can ruin any 'zero-copy' attempts, which we did also hint at explicitly in subsection 7.1.1. 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.

6.2.6 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 Figure 6.13 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 ob-

viously 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`:

```
class UpdateRate {
  public:

  using FieldType = uint32_t;

  /**
   * Update equals the send method of the event. This triggers the transmission of the notify (if configured) to the
   * subscribed clients.
   *
   * In case of a configured Getter, this has to be called at least once to set the initial value.
   */
  void Update(const FieldType& data);
  /**
   * Registering a GetHandler is optional. If registered the function is called whenever a get request is received.
   *
   * If no Getter is registered ara::com is responsible for responding to the request using the last value set by update.
   * This implicitly requires at least one call to update after initialization of the Service, before the service is offered.
   * This is up to the implementer of the service.
   *
   * The get handler shall return a future.
   */
  void RegisterGetHandler(std::function<ara::com::Future<FieldType>()> getHandler);
  /**
   * Registering a SetHandler is mandatory, if the field supports it.
   * The handler gets the data the sender requested to be set. It has to validate the settings and perform
   * an update of its internal data. The new value of the field should than be set in the future.
   *
   *
   * The returned value is sent to the requester and is sent via notification to all subscribed entities.
   */
  void RegisterSetHandler(std::function<ara::com::Future<FieldType>(const FieldType& data)> setHandler);
};
```

Figure 6.19: 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.

6.2.6.1 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 an 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 "optimzized away" if the developer does not register a "GetHandler" and leaves the handling of `Get()` calls entirely to the Communication Management implementation.

6.2.6.2 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::com::Future<FieldType>(const FieldType& data)>` revelas, 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 superflous as the Communication Management would update the field value with the returned value of the "SetHandler" anyways.

6.2.6.3 Ensuring existence of "SetHandler"

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

6.2.6.4 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 throw an unchecked excepetion indicating this programming error.

6.3 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 Appendix

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.

7.1 Serialization

`Serialization` (see [6]) 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 com-

piler settings for different `ara::com` applications were different regarding alignment decisions is consequently high. Therefore an AP product (more concrete its IPC binding 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::vector` data-types, which are generally NOT contiguous in-memory (depending on the allocation strategy). So even if the data contained in the vector is compatible with the receiver layout wise, a deep copy (meaning collecting contained elements and their references from various memory regions — see [7]) 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.

7.1.1 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 Figure 6.18, 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.

7.2 Service Discovery Implementation Strategies

Binding implementer hint

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

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 6.3) 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 [8]. 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.

7.2.1 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 demon 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.

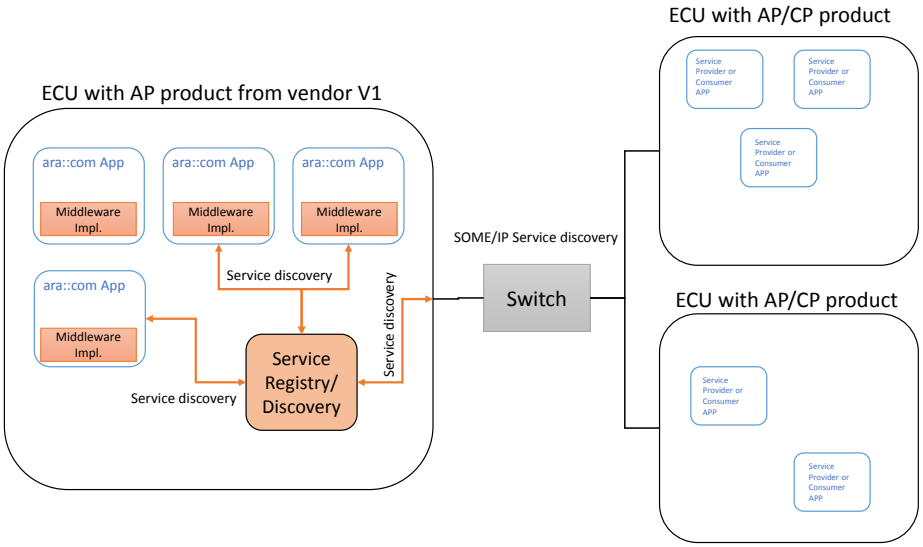

Figure 7.1: Centralized discovery approach

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.

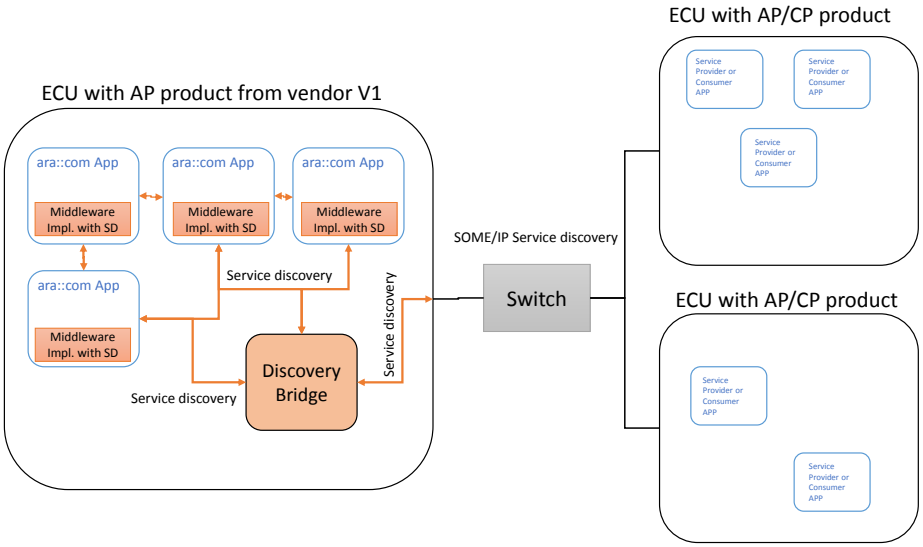

Figure 7.2: Distributed discovery approach

7.3 Multi-Binding implications

As shortly discussed in subsection 6.1.1 `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.

7.3.1 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).

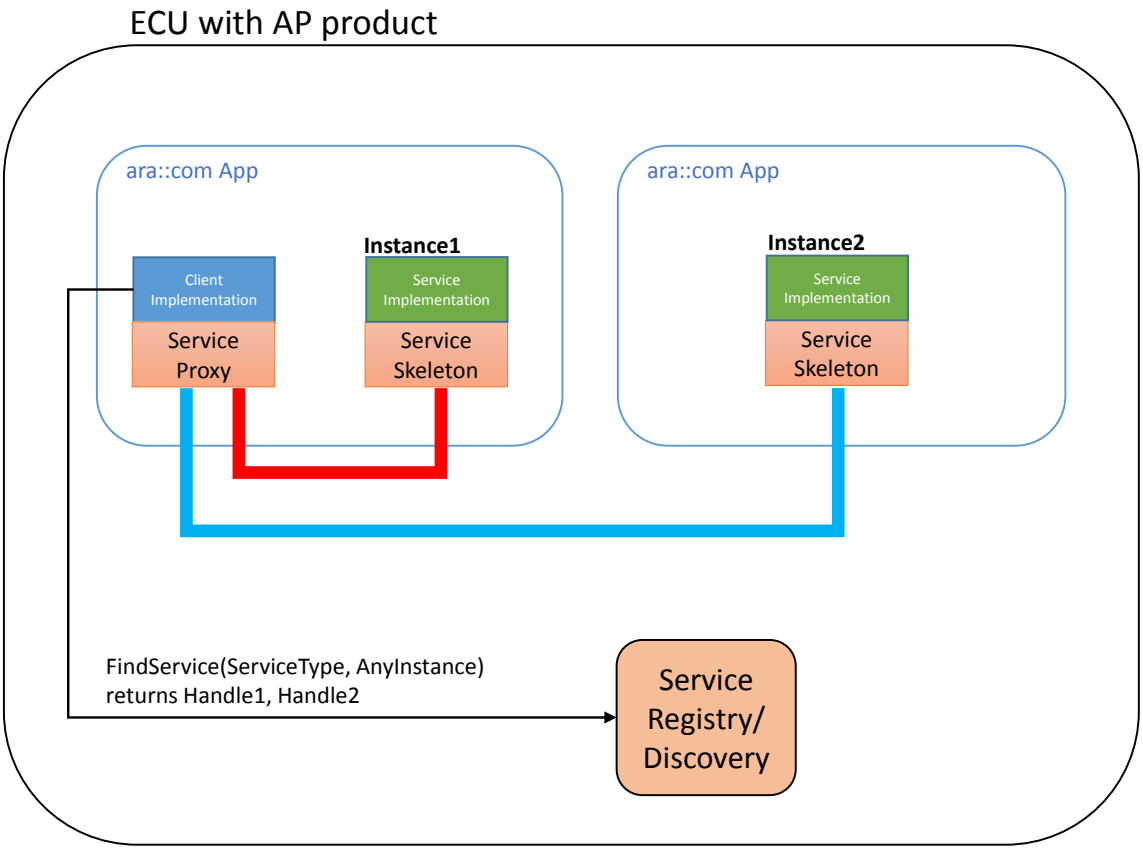

Figure 7.3: Simple Multi-Binding intra AP example

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 contact 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 can not 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!

7.3.2 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 cries out, 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 7.3.3*), but for now suppose, that this is a realistic scenario. (For other network protocols it might indeed be realistic)*

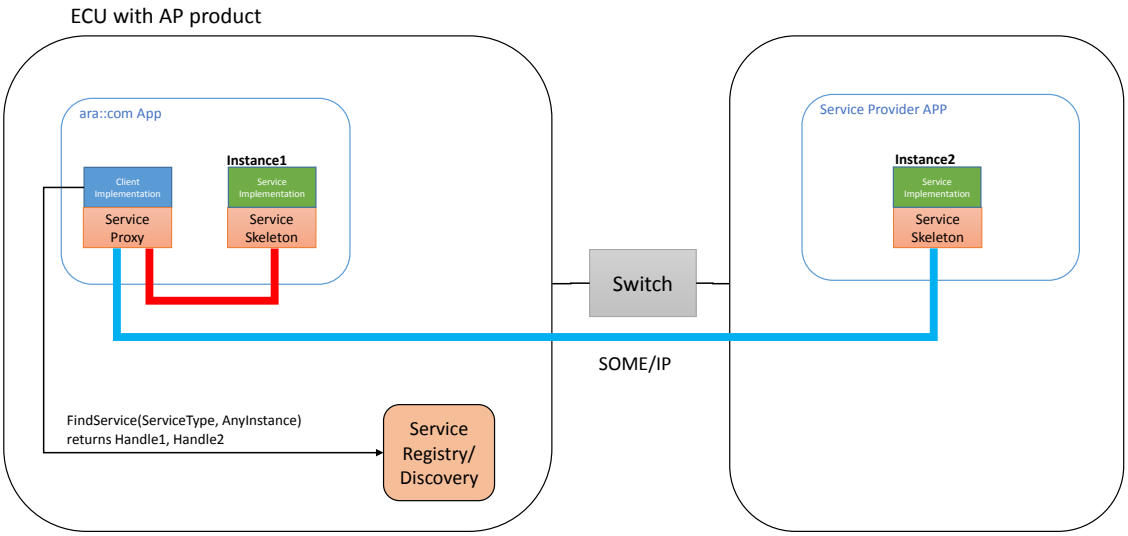

Figure 7.4: Multi-Binding local and network example

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 7.3.1) 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.

7.3.3 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.

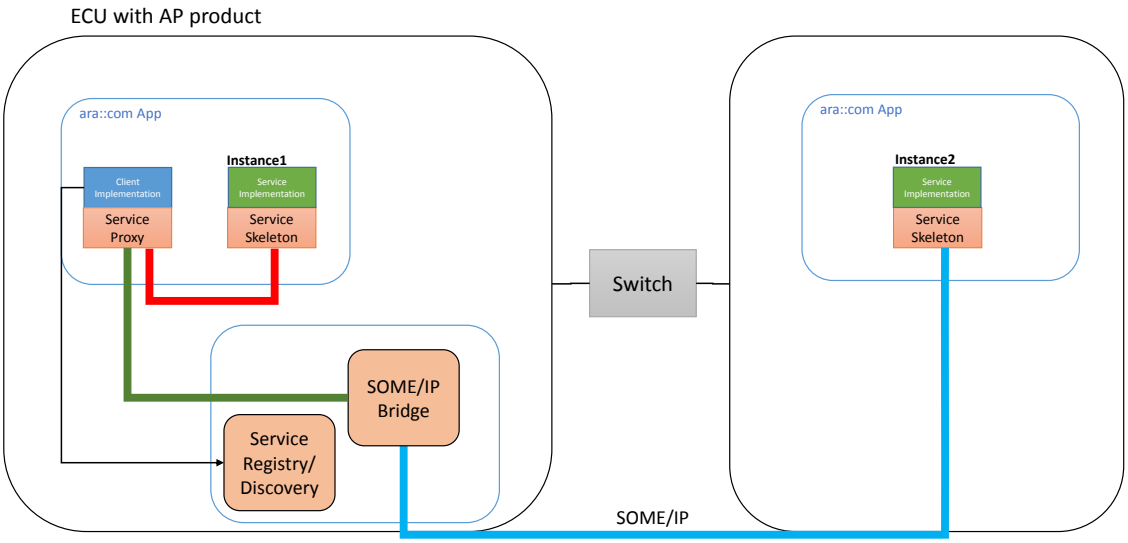

Figure 7.5: SOME/IP Bridge

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 7.1) 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 external 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.