AUTOSAR ADAPTIVE ON ZERO-COPY STEROIDS

Authors: Michael Pöhnl, Simon Hoinkis, Lyle Johnson

Apex.AI

Abstract — This white paper discusses how the inter-process communication of today's automotive ECUs can be realized with the open-source middleware Eclipse iceoryx™. It provides the functionality required for communication stacks dealing with SOME/IP or the AUTOSAR Adaptive ara::com API. Apex.AI is the most significant contributor to the Eclipse Foundation project, which is used as the base for the Apex.Middleware product. We describe the extensions provided with Apex.Middleware that enable the usage of iceoryx in safety-critical systems.

→ 1

INTRODUCTION

In the past, automotive software was written to run on Electronic Control Units (ECU) that would provide one dedicated function. For example, one ECU for the engine control, one for the airbag, one for the parking assistance, etc.
The sensors needed for these dedicated functionalities were typically connected directly to the ECUs to guarantee real-time information processing.

The software on such an ECU consists of tens or even hundreds of components that perform sub-tasks and run at different frequencies. This, together with the aforementioned hardware topology, led to the fact that most of the data communication takes place between software components on the ECU, and low bandwidth buses like LIN or CAN were sufficient to exchange the aggregated information between ECUs. The large amount of ECU internal messaging required a dedicated middleware that handles this type of communication on the embedded targets as efficiently as possible.

Since the entire ECU with hardware and software was developed by a Tier 1 supplier, the middleware for the internal data exchange was typically a proprietary piece of software and tightly coupled to an in-house development environment. This changed in the late 2000s when the first vehicles with the AUTOSAR Classic standard [1] hit the road. This standard includes the so-called Runtime Environment (RTE) that defines the APIs and semantics for the communication interfaces of software components.

With the recent emergence of powerful domain controllers and vehicle computers, more and more functionalities are being integrated on multi-core CPUs. This is accompanied by an increased decoupling of software and hardware, the use of POSIX operating systems, and Ethernet as the networking technology. This trend was addressed by the AUTOSAR consortium with the specification of the SOME/IP protocol [2] and AUTOSAR Adaptive [3] as a new platform for POSIX-based operating systems. The consequences for the middleware on modern vehicle computers are that it must be able to deal with GBytes/sec internal transmission rates, inter-process-communication (IPC), and applications with mixed safety-criticality (e.g., a non-safety-critical process communicating with a safety-critical process).

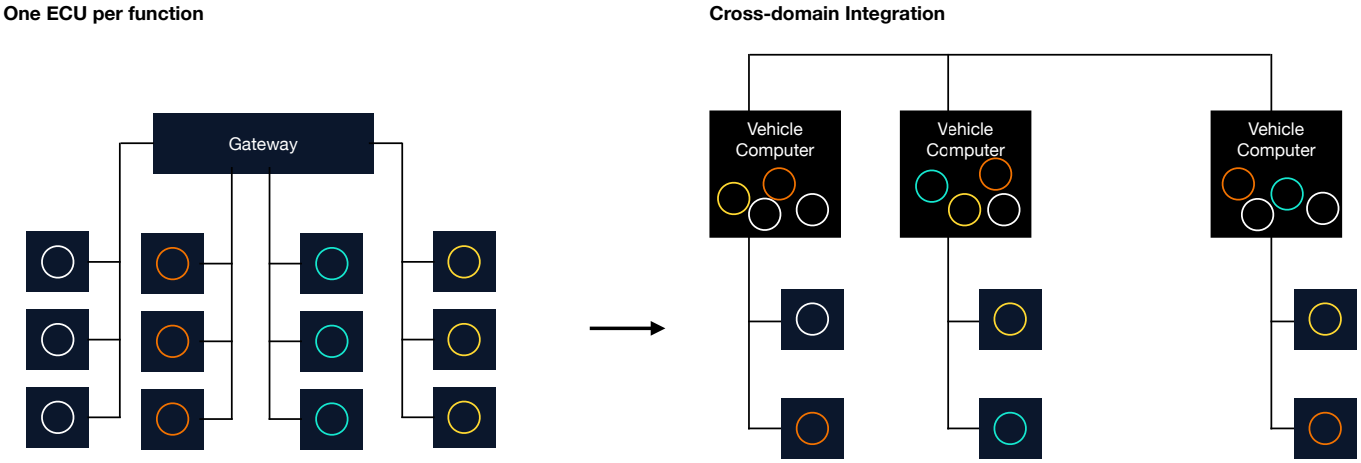

Figure 1:
Integration of many functionalities on vehicle computers

The remainder of this paper is structured as follows: we first provide a brief overview of how iceoryx has evolved from a proprietary alternative to the AUTOSAR Classic RTE to an open-source middleware used by multiple AUTOSAR Adaptive vendors. We then describe how iceoryx can be integrated in automotive communication stacks that are based on SOME/IP and the AUTOSAR Adaptive ara::com API. We introduce an open-source example that shows how a service-oriented communication API can be implemented on top of iceoryx, and provide measurements that compare this setup to one that is based on UNIX Domain Sockets [4]. Finally, we introduce the extensions that Apex.Middleware will provide for iceoryx to allow mixed-criticality systems and enable identity and access management enforcement.

→ 2

THE HISTORY OF ECLIPSE ICEORYX™

The origins of iceoryx were in the development of video-based driver assistance systems at Robert Bosch GmbH, and go back to the year 2009. Compared to other ECUs, like engine control systems, the messages to exchange between the software components were no longer signals like temperatures or pressures, but rather complex data structures like object lists or even images. Hence, the message sizes grew by orders of magnitudes from bytes to kilobytes and megabytes. Back then, it was recognized that approximately 10% of the available runtime was spent with copying of messages in the middleware. The RTE of the emerging AUTOSAR Classic stack was not a practical option, as it also made several copies when transferring messages between local senders and receivers.

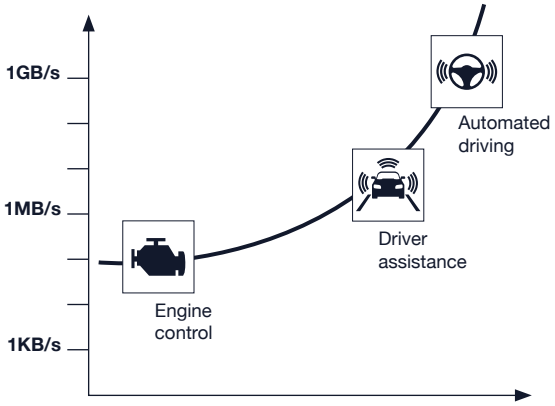

Figure 2:
Evolution of ECU internal data rate

As a solution, the DADDY (Driver Assistance Data DeliverY) middleware was developed. Compared to AUTOSAR Classic, it was already written in C++, and able to transfer a message without a single copy. This so-called zero-copy communication was enabled by smart API design and the management of memory that is shared by the software components. In the decade from the first start of series production in 2012 to 2022, DADDY was deployed on millions of advanced driver assistance system (ADAS) ECUs from Bosch.

When work on AUTOSAR Adaptive began around 2015, it was clear that the performance of the middleware would be crucial for the target systems. Therefore, Bosch decided to contribute the specification of the DADDY API when designing the ara::com communication API with BMW and other AUTOSAR partners [5]. This laid the foundation for zero-copy communication in future AUTOSAR Adaptive implementations.

During the standardization in the AUTOSAR Adaptive working group, DADDY was ported to POSIX operating systems at Bosch corporate research. One of the main changes was the introduction of shared memory across POSIX processes that have their own virtual address spaces. This eventually launched the development of what is today known as Eclipse iceoryx.

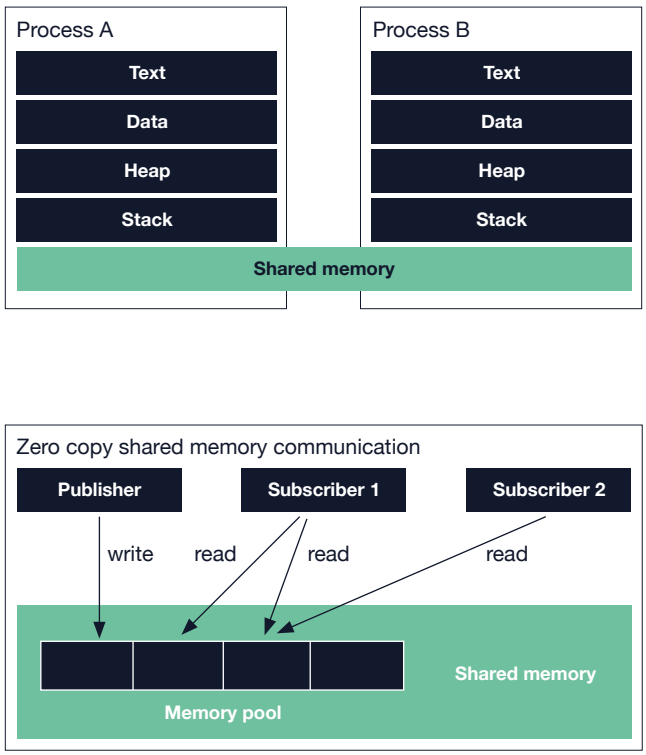

Figure 3:
Zero-copy communication with iceoryx

Since its initial release in 2016, iceoryx has been adopted in various ADAS, autonomous driving and robotic projects. In 2017 the joint venture of Robert Bosch GmbH and Daimler AG (now Mercedes-Benz Group) successfully started using iceoryx on the vehicle computers of their robotaxis [6]. A year later, ETAS joined as a user of iceoryx which led to speeding up their RTA-VRTE AUTOSAR framework [7].

In 2019, another milestone in the history of iceoryx was achieved: Bosch open-sourced its code under the Apache 2.0 license. The project was started under the umbrella of the Eclipse foundation [8] and quickly drew interest from various groups. In parallel, iceoryx gained ground in the field of robotics. Similar to the AUTOSAR community, the Robotic Operating System (ROS) community realized that zero-copy communication was to become an essential feature for modern embedded frameworks [9]. Starting from the ROS 2 release Foxy Fitzroy, the iceoryx zero-copy concept was introduced as a user API under the name LoanedMessage. At ROSCon 2019 the first zero-copy communication for ROS 2 was presented [10] — it utilized the rmw_iceoryx binding [11].

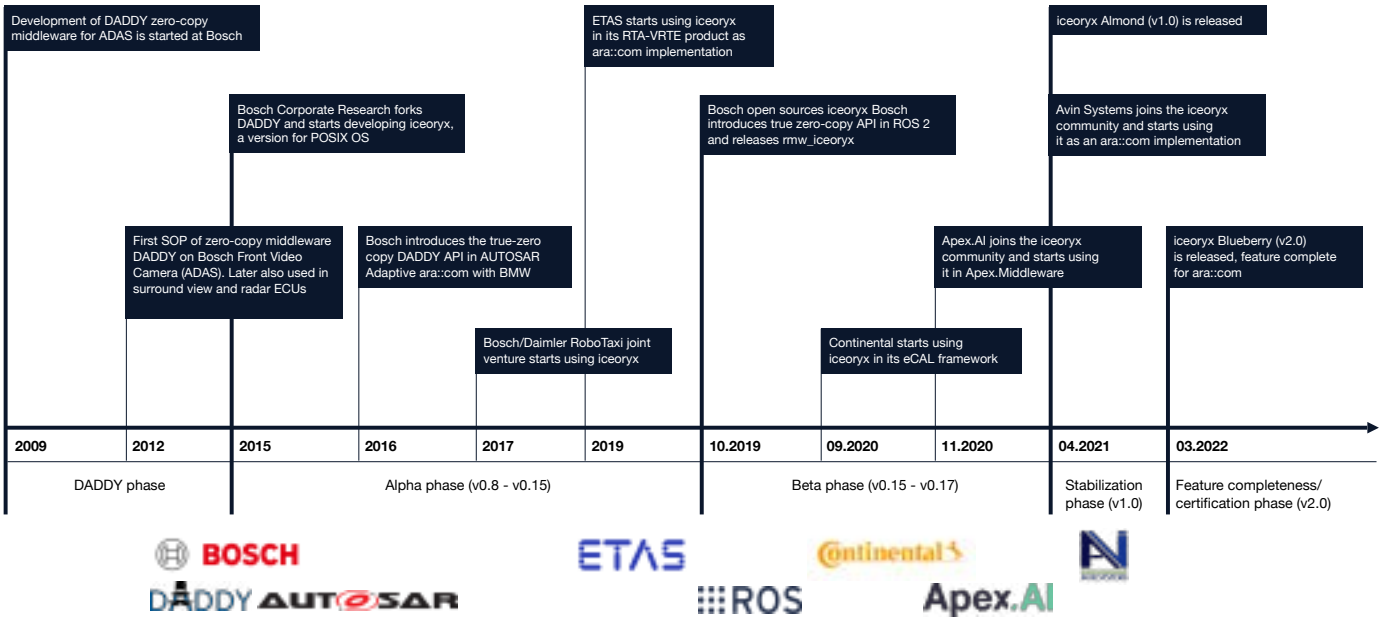

Figure 4:

The history of Eclipse iceoryx

In addition to ROS 2, iceoryx has been integrated into various proprietary and open-source frameworks in recent years. Amongst others:

- ➔ Continental's open-source eCAL framework used in ADAS development [12]
- ➔ Eclipse Cyclone DDS, one of the leading open-source implementations of the DDS middleware standard [13]
- ➔ AUTOSAR Adaptive Platform implementation from Avin Systems [14]]
- ➔ Apex.Middleware, the automotive grade middleware from Apex.AI [15]

More details on how zero-copy communication works with iceoryx can be found in an Eclipse article [16] and on the iceoryx website [17].

→ 3

ECLIPSE ICEORYX™ COMPLEMENTS SOME/IP AND ARA::COM

iceoryx provides all that is needed for a high-performance, inter-process-communication in stacks that are based on SOME/IP or that otherwise come with an ara::com API. In the sections that follow, we introduce the AUTOSAR Adaptive service model and describe how the key requirements for these stacks were considered when designing iceoryx.

The AUTOSAR Adaptive service model

Unlike most other middleware technologies, SOME/IP and ara::com allow the grouping of many communication entities into services [18]. A service can have zero or multiple of the following elements:

- → **Event:** push communication from a sender to many receivers. They can be sent cyclically or on change
- → **Method:** remote procedure calls that can be sent from many clients to a server
- → **Field:** always holds the latest value and enables the following
  - Provide the value on subscription
  - Get updates on change of the value after the subscription, like an event
  - Have a *getter* to query the latest value with a remote procedure call
  - Have a *setter* to change the value with a remote procedure call

A service can be identified with a unique service ID. Concrete instances of a service can be identified with an instance ID. As events, methods, and fields within a service are also identified with an ID, the combination of all these IDs allows for uniquely identifying a provided communication entity in the system. On the are::com API, a service provider is implemented by a skeleton class and a service consumer by a proxy class, whereas an instance of a proxy class is always bound to a specific service instance [5].

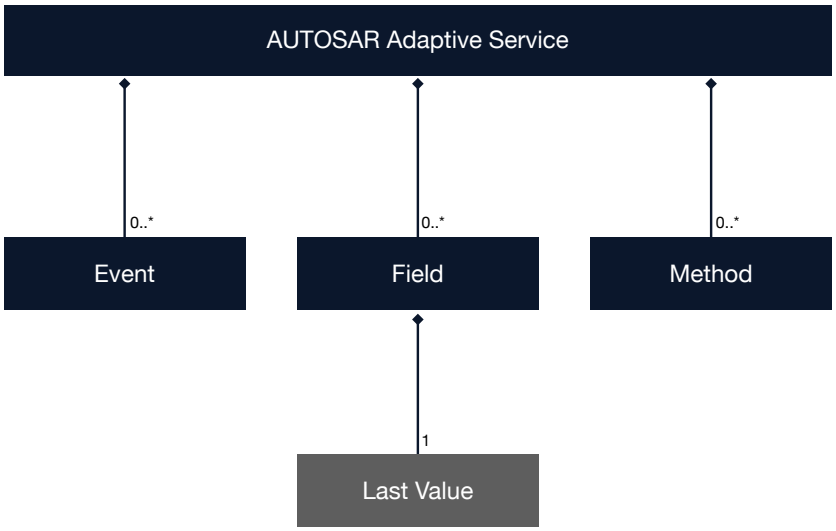

Figure 5:
The AUTOSAR Adaptive service model

Publish / subscribe communication

Publish / subscribe is a messaging pattern that allows to decouple the senders of messages (publishers) from the receivers of messages (subscribers). The shared knowledge is the so-called topic which is the communication channel to which publishers can send messages and subscribers can subscribe. This architecture also allows for a more flexible topology with different lifetimes of publishers and subscribers.

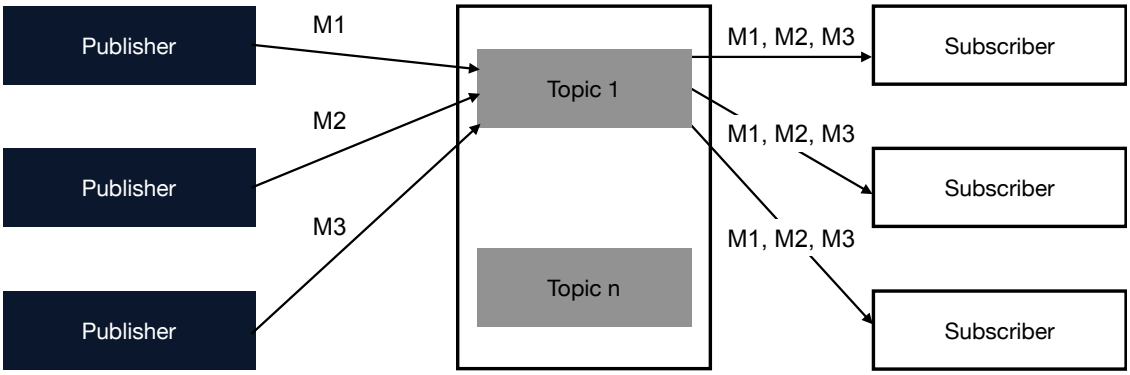

Figure 6:
The publish/subscribe communication pattern

Publish / subscribe communication with dynamic connections has been supported by iceoryx from the start, and it is well suited to implement the event communication needed by SOME/IP and ara::com. The delivery of updates of the field values can also be realized with this mechanism. The combination of service, instance, and event ID then corresponds to a topic ID. iceoryx allows for dynamically unsubscribing or resubscribing to an existing subscriber object, as well as for requesting the current subscription state. This fine grained control is needed when implementing the ara::com event API.

iceoryx supports n:m publish / subscribe communication, which means that n publishers can send messages to a topic to which m subscribers are subscribed. This flexibility is needed when using iceoryx in a DDS-based system like ROS 2. SOME/IP and ara::com only need a 1:m subset as proxies are created for a specific service instance. Therefore, an event receiver in a proxy object will only subscribe to a single event sender. This special kind of publish / subscribe communication can be taken into account in iceoryx with a configuration that treats more than one publisher on a topic as an error.

Having introduced the history of iceoryx in the last chapter, it is obvious that iceoryx is a perfect match for the zero-copy API in ara::com. With this API on top of iceoryx, proper zero-copy communication is possible without a single copy when transferring messages between event senders and receivers on a single host.

Request / response communication

Request / response is a communication pattern that is based on clients and servers. Clients can send requests to servers which respond after having processed a request. This pattern is needed to implement the method communication as well as the *setters* and *getters* of fields.

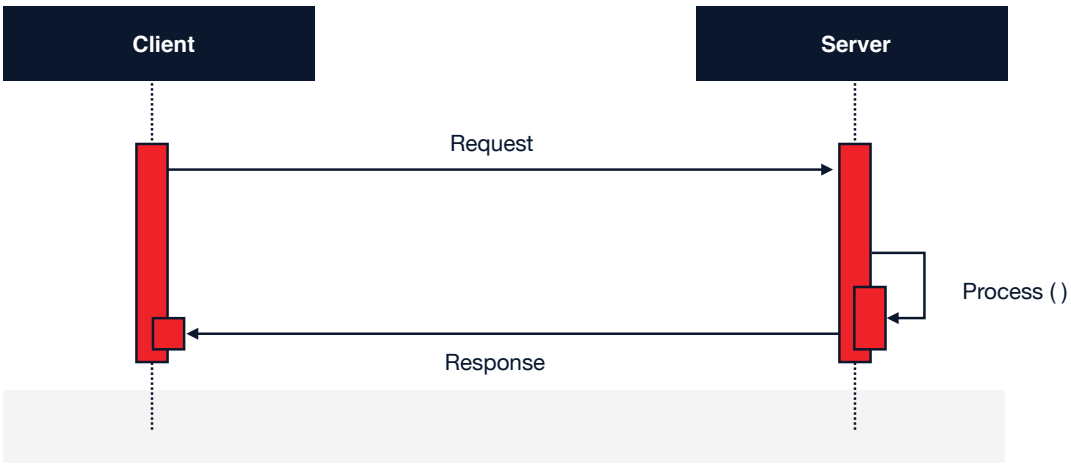

Figure 7:
The request/response communication pattern

Support for request / response communication was introduced as a first-class citizen in iceoryx Blueberry (v2.0.0). It uses the same efficient shared memory mechanisms behind the scenes, and provides client and server classes on the user API that enable things like mapping of responses to requests. iceoryx detects if a client wants to send a request to a non-existing server, which can be used for the necessary error handling on the ara::com layer.

SOME/IP and ara::com also support *fire & forget* communication, which is a special variant of the method communication and is technically a request to which the server sends no response. Since this actually corresponds to a n:1 publish / subscribe communication, it can be realized easily with iceoryx.

Providing historical data

Historical data in the context of a middleware is about the ability to provide previously sent messages to late joining subscribers. In middleware technologies like DDS, a subscriber can request a number of last sent messages which will be delivered on subscription by the matching publishers. Certainly, it depends on the number of already produced messages whether this request can be fulfilled or not.

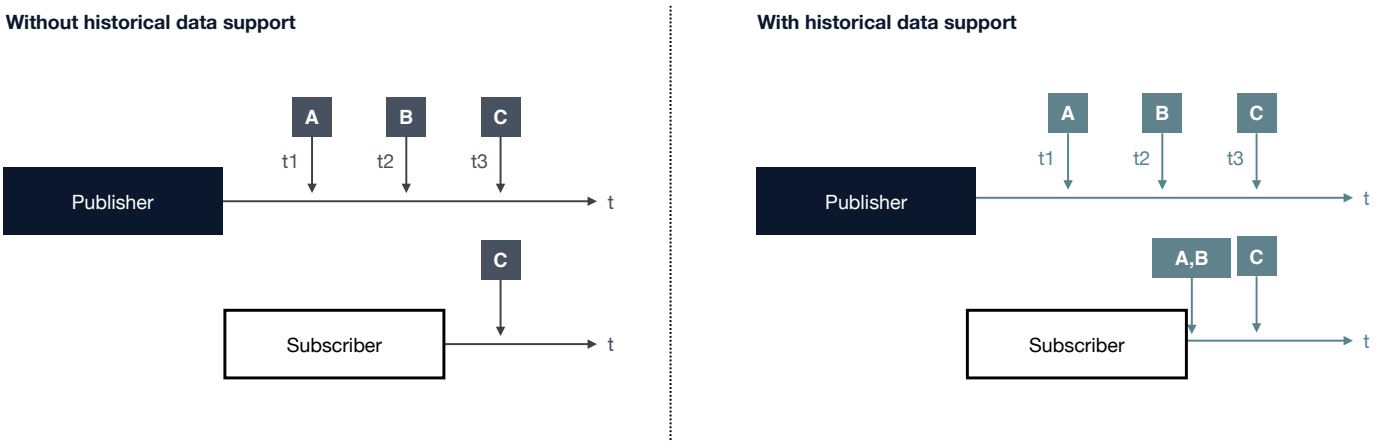

Figure 8:
Support for historical data

iceoryx allows configuring the history capacity on the publisher side, which will end up with a ring-buffer containing references to the last sent messages. The subscriber can be configured to request some number of last messages. This functionality is used to implement the field behavior needed for SOME/IP and ara::com. A field always has to hold the last value to be able to provide it on subscription or when the *getter* method is called. If both, the publisher history capacity and the subscriber history request, are set to 1, iceoryx will provide the last sent value on subscription, even before the subscription state is changed to “SUBSCRIBED”.

The ara::com specification prescribes that a field value must be set before the field is offered to others. This is needed to fulfill the contract that a subscriber or a *getter* call will always get a value. This contract can be enforced with the help of iceoryx, as it allows to configure that a publisher is not offered on creation and enables the ara::com layer to first perform the necessary checks before explicitly making the publisher visible to subscribers.

Polling and event-driven interaction

The ara::com specification describes the two most common ways for consuming messages received by the middleware. Polling means that the middleware is explicitly queried for new data, this could be done in a cyclically executed application, for example. Event-driven means that the middleware notifies the application when new data arrives or if a state change happens. This could be done with callbacks, which are heavily used by the ara::com API.

Unlike many other middleware APIs, iceoryx supports both polling and event-driven interaction. Subscribers, clients, and servers all have configurable queues to store incoming messages, requests, and responses. These queues can be polled for data whenever the consuming side wants to do so. Additionally, iceoryx provides WaitSet and Listener classes which are comparable to the corresponding elements in the DDS specification. A WaitSet allows you to wait in a user thread for an event, like a new message. This means the thread of execution is in a non-busy wait until an event or an optional timeout happens. The Listener comes with its own thread of execution that will asynchronously execute a provided callback whenever an event occurs.

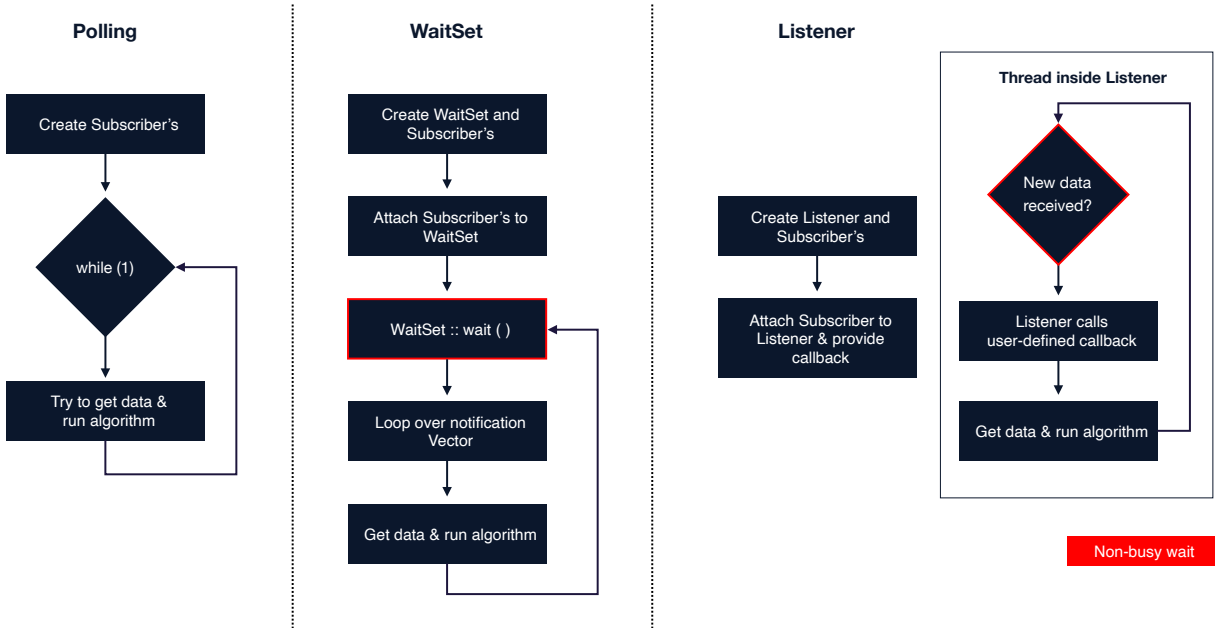

Figure 9:

The different ways for consuming messages with iceoryx

In iceoryx, subscribers, clients, and servers can be attached to and detached from the WaitSet and Listener during runtime. This allows an ara::com implementation to handle all the setting and unsetting of callbacks. For example, the SetReceiveHandler() method to register a callback to signal the arrival of new data can attach the corresponding iceoryx subscriber to an iceoryx Listener. The Listener will then take care of executing the callback provided by the user.

Service discovery

A middleware that supports service discovery is able to discover available communication entities and can provide this information to the developer upon request. Many of the middleware technologies that support service discovery automatically connect matching entities, such as a publisher and a subscriber, to the same topic. SOME/IP and ara::com have service discovery, but a subscription is not done automatically — it has to be initiated by the user.

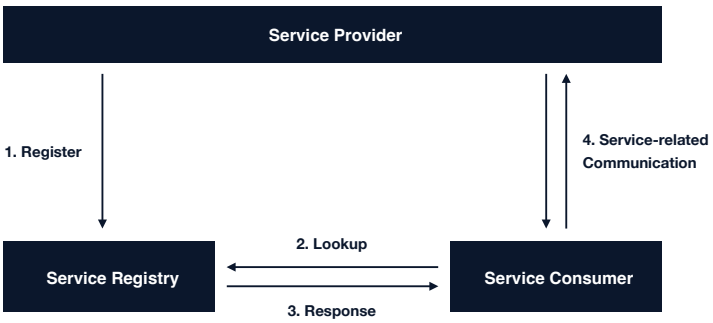

Figure 10:

Service discovery

iceoryx supports these different philosophies which enables its usage in many different communication stacks. For example, a subscriber can be configured whether it shall try to subscribe on creation or not. Once created, the developer can request a subscription or an unsubscription at any time.

Additionally, iceoryx allows querying of the available publishers and servers. The service description used to identify communication entities contains three different IDs; these can be used to hold the service ID, instance ID, and event / method ID needed for ara::com. A discovery query can then be done for one specific combination of IDs or with wildcards, for example to search for all entities belonging to instances of a specific service.

The iceoryx discovery component is an API element that can also be attached to WaitSet's and Listener's. This enables developers to be triggered whenever there is a change in availability of publishers and servers, and can be used to implement the discovery callbacks for ara::com.

→ 4

AN AUTOMOTIVE SOA EXAMPLE BASED ON ECLIPSE ICEORYX™

The following picture depicts the content of the automotive SOA (service-oriented architecture) example [19] discussed in this chapter.

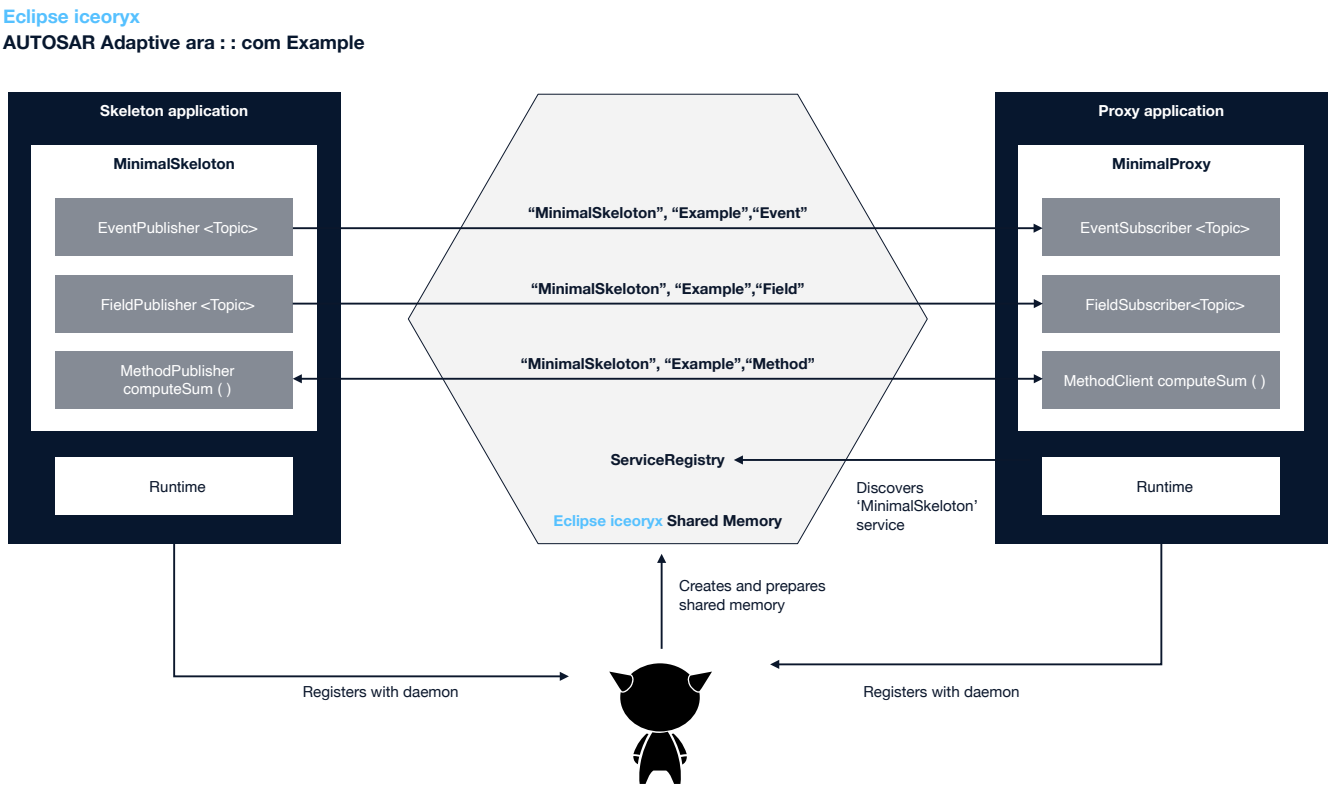

Figure 11:

The iceoryx ara::com example

The example consists of two applications and the iceoryx daemon. The daemon is responsible for setting up the shared memory and maintaining a service registry. The skeleton application is the one producing data, and it owns a MinimalSkeleton object. In AUTOSAR, each skeleton represents an abstract service which can be instantiated for other applications to subscribe to. The proxy application is consuming data and owns a MinimalProxy object. Typically, with AUTOSAR Adaptive skeletons and proxies are not written by the developer, but rather it is generated from an AUTOSAR meta-model. Besides the skeleton and the proxy, each application has a Runtime object which registers with the daemon on creation and maps the shared memory used for inter-process communication. In addition, the Runtime provides the interface to discover services by querying the iceoryx ServiceRegistry in shared memory.

Skeleton application and MinimalSkeleton

The MinimalSkeleton class contains all three communication patterns defined in SOME/IP:

- → **Events**
- → **Fields**
- → **Methods**

While the first two are members which can be accessed by the developer, the method is a functor which the developer can call as if it was a method of the MinimalProxy.

To realize this example, four different communication entities are needed in iceoryx. One publisher for the event, one server for the method, and a combination of publisher and server for the field.

The service MinimalSkeleton is considered as available as soon as all four members are available.

In the section Latency comparison, sending different structs via iceoryx's shared memory implementation is compared with a UNIX domain sockets one.

Proxy application and MinimalProxy

The MinimalProxy class contains the respective counterpart to be able to consume the MinimalSkeleton service.

When starting the proxy application, the discovery phase happens first. Initially, a synchronous find service call is performed. If the skeleton application was not started yet and the instance of MinimalSkeleton is not yet available, an asynchronous find service callback is set up. Once the MinimalSkeleton instance becomes available, a callback will be executed, which will create the MinimalProxy instance. The main() function is then able to run through its routine and receive the data from the field, and trigger the remote-procedure call by calling computeSum(). Receiving the data from the event is done in an asynchronous manner by setting up a receiver handler, which is called instantly once data has arrived.

Entity	Kind	Service Identifier	Instance Identifier	Event / Field / Method Identifier
m_event	Publish / Subscribe	"ExampleService"	"ExampleInstance"	"Event"
m_field	Publish / Subscribe	"ExampleService"	"ExampleInstance"	"Field"
m_field	Request / Response	"ExampleService"	"ExampleInstance"	"Field"
computeSum	Request / Response	"ExampleService"	"ExampleInstance"	"Method"

→ 5

LATENCY COMPARISON OF ICEORYX'S ZERO-COPY MECHANISM WITH UNIX DOMAIN SOCKET

As of today, many AUTOSAR Adaptive ara::com solutions use UNIX domain sockets for inter-process-communication. This chapter discusses the results of measuring the latency with both — the iceoryx implementation from the previously discussed example and an implementation based on UNIX domain sockets.

The measurements are based on the example presented previously. Different structs are sent via m_event which contain a counter, a timestamp, and an array of arbitrary bytes.

The measurements were performed with the following hardware and software:

Board: Renesas R-Car V3H
CPU architecture: aarch64le
Operating system: QNX QOS222 v0.5.1
Compiler: QCC 8.3.0

Other settings were:

Branch on GitHub: measurement-without-prints [20]
Samples per data point: 500

The following topics were used to measure the latency [21]:

- → TimestampTopic1Byte
- → TimestampTopic4Kb
- → TimestampTopic16Kb
- → TimestampTopic64Kb
- → TimestampTopic256Kb
- → TimestampTopic1Mb
- → TimestampTopic3Mb

Depicted in the logarithmic chart below are the mean values for different message sizes.

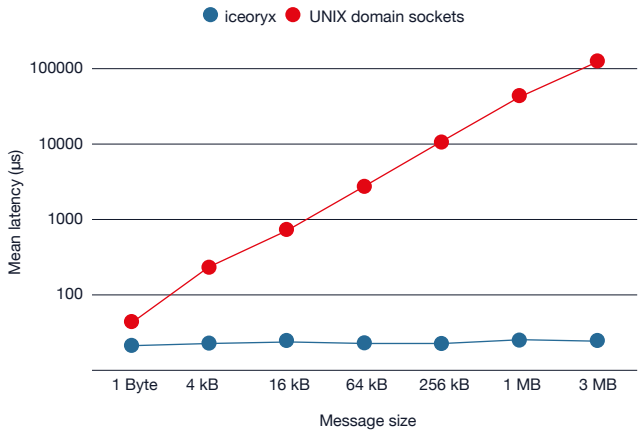

Figure 12:
Comparison of iceoryx and UNIX domain sockets

It is seen that the mean latency increases with the message size when communicating over UNIX domain sockets. Unlike UNIX domain sockets, iceoryx provides an approximately constant mean latency for the different message sizes. The message size of 1 Byte shows the minimal latency achievable by UNIX domain sockets and its abstraction class. Larger message sizes result in higher latency due to the fragmentation of the message into several UNIX domain messages.

In summary, one can tell that the zero-copy API of iceoryx enables a constant transmission time independent of the message size and is hence beneficial for systems which need to exchange large messages like for example ADAS.

→ 6

EXTENSIONS TO ECLIPSE ICEORYX™ IN APEX.MIDDLEWARE

The Apex.Middleware product is based on iceoryx and is currently being safety certified according to ISO 26262 ASIL D. Besides artifacts like MC/DC test coverage and a safety manual, additional safety and security features are crucial for the usage of iceoryx in automotive systems. Two of the most important features are identity and access management, and support for mixed safety criticality.

Identity and access management

As the name indicates, identity and access management consists of two parts. The first one is the authentication of the identity of applications in the system. The second one is controlling the access to communication resources for these applications. For the latter, authentication is a prerequisite.

The iceoryx communication infrastructure is based on shared memory segments. Applications register with a daemon which provides the information required for the segments, which are then mapped into the virtual address space of the application's process. Part of this registration is the authentication of the process identity. By that, an application gets only read or write access to selected segments defined by a system configuration.

After mapping the shared memory segments, an application can create communication resources, like publishers and subscribers. Which resources an application is allowed to create and access is again predefined and enforced by the middleware daemon.

Mixed criticality support

In a mixed criticality system, applications with different ASILs coexist. One crucial aspect is freedom-from-interference which means that an application with a higher ASIL cannot be influenced by an application with lower ASIL. Freedom-from-interference covers different aspects like memory or timing interference. Memory interference is given when a lower ASIL application corrupts memory that is accessed by a higher ASIL application. An example of timing interference is a higher ASIL application being delayed in execution because the CPU is blocked by a lower ASIL application.

In the scope of a shared memory middleware like iceoryx, freedom-from-interference is primarily about memory interference. With the previously mentioned shared memory segments and the identity and access management, spatial separation between unrelated communication entities can be guaranteed.

The remaining challenge is then the direct communication between applications with different ASIL. For this use case additional measures are needed to prevent or at least reliably detect things like corruption of the shared memory communication infrastructure or modification of already sent messages that are currently being processed by a higher ASIL application. Depending on whether a message is produced or just forwarded by a lower ASIL application, additional mechanisms like plausibility checks or end-to-end protection are possible.

→ 7

SUMMARY AND CONCLUSION

iceoryx is well suited as a shared memory backbone in modern automotive communication stacks. In its more than 12 year history, it has evolved from a proprietary zero-copy middleware to an open-source technology used by many automotive companies.

It was developed in parallel with AUTOSAR Adaptive's ara::com API and supports all the communication patterns that are needed for interfacing with the SOME/IP protocol. At Apex.AI, iceoryx is used as the core for the Apex.Middleware product and it is extended with the features that enable the usage in safety-critical automotive systems.

[1] https://www.autosar.org/standards/classic-platform/
[2] https://some-ip.com
[3] https://www.autosar.org/standards/adaptive-platform/
[4] https://en.wikipedia.org/wiki/Unix_domain_socket
[5] https://www.autosar.org/fileadmin/user_upload/standards/adaptive/21-11/AUTOSAR_EXP_ARAComAPI.pdf
[6] https://www.youtube.com/watch?v=nMyqQHqDmMo
[7] https://www.etas.com/en/products/rta-vrte.php
[8] https://projects.eclipse.org/projects/technology.iceoryx
[9] https://design.ros2.org/articles/zero_copy.html
[10] https://vimeo.com/379127778
[11] https://github.com/ros2/rmw_iceoryx/
[12] https://github.com/continental/ecal/
[13] https://cyclonedds.io/
[14] https://www.avinsystems.com/products/autosar_ap/
[15] https://www.apex.ai/apex-middleware
[16] https://www.eclipse.org/community/eclipse_newsletter/2019/december/4.php
[17] https://iceoryx.io/
[18] https://www.autosar.org/fileadmin/user_upload/standards/foundation/21-11/AUTOSAR_PRS_SOMEIPProtocol.pdf
[19] https://github.com/eclipse-iceoryx/iceoryx-automotive-soa
[20] https://github.com/mossmaurice/automotive_soa_unix_domain_socket_comparison/tree/measurement-without-prints
[21] https://github.com/mossmaurice/automotive_soa_unix_domain_socket_comparison/blob/measurement-without-prints/include/topic.hpp