RCA
Reference CCS Architecture

OCORA

Generic Safe Computing Platform

Specification of the PI API between Application and Platform

Developed in collaboration among DB Netz AG, duagon AG, Nederlandse Spoorwegen, Real-Time Innovations (RTI), SBB, Siemens Mobility GmbH, SNCF Voyageurs, SNCF Réseau, SYSGO GmbH, Thales and Wind River

Version 2.0, July 2022

List of Contributors

Table 1. List of contributors to this document (in alphabetical order).

Name	Company
Zeeshan Ansar	SYSGO GmbH
Holger Blasum	SYSGO GmbH
Mario Brotz	SYSGO GmbH
Mark Carrier	Real-Time Innovations (RTI)
Christian Daniel	SNCF
Stefan Eberli	duagon AG
Heiko Erben	SBB AG
Markus Fuchs	Wind River
Reinhard Hametner	Thales
Mark Hary	Real-Time Innovations (RTI)
Michael Henze	duagon AG
Nikolaus König	Thales
Patrick Marsch	DB Netz AG
Thomas Martin	SBB AG
Angel Martinez Bernal	Real-Time Innovations (RTI)
Oliver Mayer-Buschmann	DB Netz AG
Prashant Pathak	DB Netz AG
Stefan Resch	Thales
Harald Roelle	Siemens Mobility GmbH
Remco Schellekens	Nederlandse Spoorwegen
Kai Schories	DB Netz AG
Christian Schuster	duagon AG
Kai Schwarzkopf	Siemens Mobility GmbH
Betül Sögütlü	DB Netz AG
Axel Träger	Siemens Mobility GmbH
Nicolas Van Spaandonck	Wind River
Julian Wissmann	DB Netz AG

Version History

Table 2. Version history of this document.

Date	Version	Description
Dec 7th, 2021	1.0	Draft initial considerations on the PI API between applications and platform, as developed among RCA / OCORA members.
July 1st, 2022	2.0	Strongly evolved considerations on the API among railway applications and computing platform, as developed jointly among DB Netz AG, duagon AG, Nederlandse Spoorwegen, Real-Time Innovations (RTI), SBB, Siemens Mobility GmbH, SNCF Voyageurs, SNCF Réseau, SYSGO GmbH, Thales and Wind River.

1 Motivation

The railway sector is currently undergoing the largest technology leap in its history, with many railways in Europe and across the globe aiming to introduce large degrees of automation in rail operation. Beyond the rollout of the European Train Control System (ETCS), most railways are for instance aiming at introducing Automated Train Operation (ATO), in some cases up to fully driverless train operation (Grade of Automation 4, GoA4), and an automated dispatching of rail operation, typically referred to as a Traffic Management System (TMS).

In this context, various novel technologies are introduced into the rail sector, such as artificial intelligence (AI), advanced perception, or high-precision train localization, and many new functions and railway applications are introduced on both train and infrastructure side. Further, the railway initiatives Reference Control Command and Signalling Architecture (RCA) [1] and Open Control Command and Signalling Onboard Reference Architecture (OCORA) [2] are driving a functional architecture for the trackside and onboard functions for future rail operation, which is expected to be taken further in the new Europe's Rail programme [3].

It is obvious that this massive transition in the rail sector has to be accompanied by the development of appropriate and future-proof connectivity and IT platforms. As a key paradigm change in this respect, the railways in RCA and OCORA find it important to introduce a standardized separation of (safety-relevant and non-safety-relevant) railway applications and the underlying IT platforms, in order to be able to decouple the very distinct life cycles of the domains, maximally leverage latest advances in the IT sector for the rail sector, and to be able to aggregate multiple railway applications on common IT platforms.

In pursuit of this paradigm change, RCA and OCORA started working on the concept of a Safe Computing Platform (SCP) in 2020 [4] that foresees the notion of a so-called Platform Independent Application Programming Interface (PI API) between safety-relevant railway applications and IT platform and hence supports portability of railway applications among IT platform realisations from different vendors.

2 Aim and Scope of this Document

This document aims to provide a first version of a (partial) specification of the aforementioned PI API between (safety-relevant and non-safety-relevant) railway applications and underlying IT platforms, following the motivation and objectives stated in [4].

The specification was jointly developed among DB Netz AG, duagon AG, Nederlandse Spoorwegen, Real-Time Innovations (RTI), SBB, Siemens Mobility GmbH, SNCF Voyageurs, SNCF Réseau, SYSGO GmbH, Thales und Wind River in the time frame between December 2021 and June 2022. This first version of a specification is envisioned to serve as a basis for subsequent prototyping on a modular separation of (safety-relevant and non-safety-relevant) railway applications and IT platforms, for instance in the context of the Europe's Rail Innovation Pillar [5].

It is important to note that due to the short time frame in which this specification was developed, and various open questions, this can only be seen as an initial hypothesis of a possible PI API. It is expected that this will be further refined, also based on findings from early prototyping, for instance in the context of the Europe's Rail System and Innovation Pillars. It is also expected that it will be complemented with further specifications, for instance related to standardized logging, diagnostics, life-cycle management, etc., and by the development of a modular certification approach.

Finally, it is important to note that the PI API as specified in this document refers to both, trackside data centre and onboard deployments, as it is assumed that key principles regarding the interaction of applications and computing platforms should be the same for trackside and onboard.

The further document is structured as follows:

- In Section 3, abbreviations and the notation used in this specification are introduced;
- In Section 4, key entities referred to in the specification are defined, complemented by specific deployment examples of these for illustration purposes;
- In Section 5, guiding principles regarding the PI API design are listed;
- In Section 6, key messaging principles are introduced;
- In Section 7, basic principles related to execution model and timing behaviour are introduced;
- In Section 8, the notion of gateways (for communication of applications with platform-external entities) is introduced;
- Section 9 deals with how the platform should handle faults, errors and failures;
- Section 10 then contains detailed considerations on the PI API between applications and IT platform, based on all the considerations from the previous sections;
- Section 11 provides a concrete application example for illustration purposes;
- Section 12 finally provides an outlook on the further specification and prototyping likely required in the future.

3 Abbreviations and Notation

3.1 Abbreviations

Table 3 lists the abbreviations that are used throughout this document.

Table 3. Abbreviations.

Abbreviation / Term	Description
ARINC	Aeronautical Radio Inc.
ATO	Automatic Train Operation
CCS	Control Command and Signaling
CPU	Central Processing Unit
DDS	Data Distribution Service
DMIPS	Dhrystone Millions of Instructions per Second
ETCS	European Train Control System
Ffs	for future study
FRMCS	Future Railway Mobile Communication System
GoA4	Grade of Automation 4
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronic Engineers
IPC	Inter-Process Communication
IT	Information Technology
JSON	JavaScript Object Notation
MooN	M-out-of-N configuration in the context of composite fail safety (acc. to EN 50129)
OCORA	Open CCS Onboard Reference Architecture
PI API	Platform-independent Application Programming Interface
POSIX	Portable Operating System Interface
RaSTA	Rail Safe Transport Application
RCA	Reference CCS Architecture
RTE	Runtime environment (see definition in [1])
SCP	Safe Computing Platform
SIL	Safety Integrity Level
SRAC	Safety-related application condition
Tbd	to be discussed
TCP/IP	Transmission Control Protocol / Internet Protocol
TMS	Traffic Management System
UDP/IP	User Datagram Protocol / Internet Protocol
UML	Unified Modelling Language

3.2 Notation

In the remainder of this document, the following notation is used:

Notes in italics and in grey are not parts of the specification as such, but additional explanatory comments (for instance explaining the background of certain decisions taken in the context of the API specification).

Open points are indicated in italics and in red. These are points that could not be fully concluded and which are for future study. In later versions of this specification, these notes are expected to be resolved and removed.

4 Definitions

4.1 Definition of Entities

In the remainder of this specification document, the following definition of entities is used, as described in Table 4.

Table 4. Definition of entities.

Functional Actor	A fully deterministic functional module that provides a specific application functionality. All functionality within a Functional Actor has a common functional safety requirement. A Functional Actor is the entity to which the Platform applies composite fail safety to meet the functional safety requirement. It is assumed that the split of a Functional Application into multiple Functional Actors is left to the discretion of the application vendor.
Functional Actor Replica	An instance of a Functional Actor that is run on a single Computing Element, possibly jointly with Replicas of other or the same Functional Actors, if mechanisms are provided by the Platform that ensure sufficient independence (e.g., in CPU and memory usage) between Functional Actor Replicas to fulfill the CENELEC norms EN 5012x. It can be restored to a specific state to be in sync again with other Replicas after an error. The Platform applies voting to the outputs of all Replicas of a single Functional Actor. Toward a single Replica, it is not visible that other Replicas of the same Functional Actor are running.
Functional Application	A comprehensive set of application functionality, assumed to be provided as one product by a single vendor. A Functional Application could for instance correspond to a subsystem as defined in the RCA architecture. Application-level communication among Functional Applications is expected to follow standardized interfaces, for instance defined by RCA or OCORA. Functions within one Functional Application may have different functional safety requirements.
Platform	Self-contained deployment of a Safe Computing Platform as defined in the white paper [4], comprising multiple Computing Elements and RTE Instances running on these. Note that a Platform may be geographically distributed (though the distribution is transparent to application and hence not relevant to this specification).
Computing Element	A single compute element, which may comprise multiple cores, used by a Platform. It is assumed that different Functional Actors (and different Functional Actor Replica of the same or different Functional Actors) may concurrently run on the same Computing Element if mechanisms are provided that ensure sufficient independence (e.g., in CPU and memory usage) between the Functional Actors and Functional Actor Replica to fulfill the CENELEC norms EN 5012x.

Runtime Environment (RTE)	An (instance of a) runtime environment, which comprises Safety Services and System Services as defined in [4], the communication stack for information exchange between Functional Actors on the same platform and with external entities and possibly (depending on the actual platform implementation) an operating system. It is expected that an RTE instance runs on a single Computing Element, though a single Computing Element may host multiple RTE instances. Regarding vendor multiplicity, the following constraints are assumed: • Functional Actor Replicas belonging to the same Functional Actor are expected to run on RTE Instances from the same vendor (or on RTEs which are otherwise interoperable), to avoid that inter-RTE interfaces have to be defined; • Functional Actors that communicate which each other (using Flows, as introduced in Section 6.2) are expected to run on RTE Instances from the same vendor (or on RTEs which are otherwise interoperable), unless Gateway functions are involved (see Section 8), again to avoid that related inter-RTE interfaces have to be defined.
Thread	A single independent sequence of execution running on a Computing Element.

It should be noted that for most previously defined entities one could differentiate between a *type* and a specific *instance of a type*. As an example, there could be a Functional Application type called "Interlocking" for which one instance serves geographical area A, and another instance serves geographical area B. Both instances would (in terms of code) be identical and would only differ in terms of their configuration. Similarly, a Functional Application could comprise a Functional Actor type "Field object interface" for which a single instance is created for each field object (e.g., each railway switch). The following entity relationship diagram refers to *instances* of Functional Applications, Functional Actors, etc. The relationship of the introduced entities is also illustrated in Figure 1.

Note: During the discussion on the defined entities and the entity relationship diagram in Figure 1, some shortcomings in this terminology were identified. It may hence be that a future version of this specification provides a refinement of terminology.

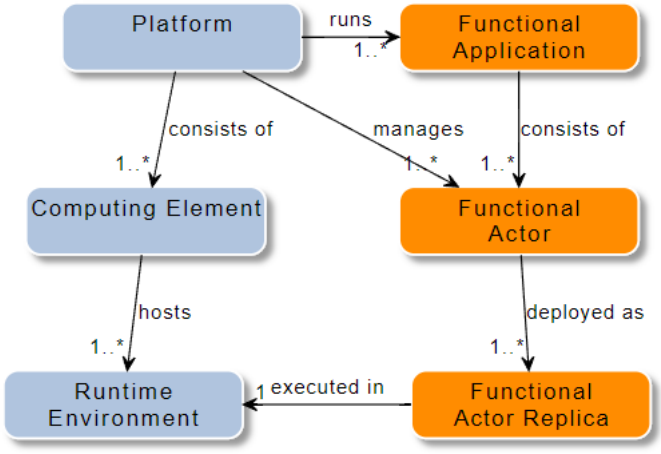


Figure 1. Relationship of defined entities (referring to *instances*, as commented in the text).

The UML code for the entity relationship diagram in Figure 1 can also be found under: LINK.

4.2 Deployment Options (for Illustration)

In this section, multiple exemplary deployment options are depicted to illustrate how the entities introduced before may relate in a practical deployment. It should be stressed that this document only aims to specify the API between Functional Applications and Platform, while the details of the Platform are left to the discretion of the vendor. Hence, all following examples are only for illustration purposes and not to be seen as part of this specification.

It is expected that the Safe Computing Platform concept and the specified PI API enable all shown deployments. However, this does not mean that each Platform realization from a specific vendor necessarily supports all deployment options (e.g., some vendors may support a geographical distribution of Functional Actor Replicas, while others don't, or there may be Platform products tailored to explicit onboard or trackside deployments).

In Figure 2, for instance, a Computing Platform deployment is shown where a Functional Application A is deployed in the form of 3 Functional Actors A1-A3. The first two of these require a 2oo3 redundancy constellation, while A3 only requires "basic integrity" and can hence run in a single replica (but it is assumed that A3 is so closely related to the other Functional Actors that it is nevertheless beneficial to run this Functional Actor on a Safe Computing Platform). In this example, all 7 required Functional Actor Replicas are assumed to run on dedicated RTE Instances.

As shown in the figure, there is a differentiation between a Platform-independent API (PI API) and a so-called "extended PI API", both of which will be detailed in Sections 8 and 10. The PI API is expected to be used by safe, replicated Functional Actors and is consequently strongly constrained. The extended PI API may only be used by non-safe, non-replicated Functional Actors and provides, e.g., access to the TCP/IP or UDP/IP stack of the Platform, as for instance needed by Gateway functions, as detailed in Section 8.

In the figure, the numbering of the RTE instances as 1-1, 1-2, etc., is used to indicate that RTE instances with the same prefix x- are in a close relation, as they for instance have to jointly handle the voting for the hosted Functional Actor Replicas.

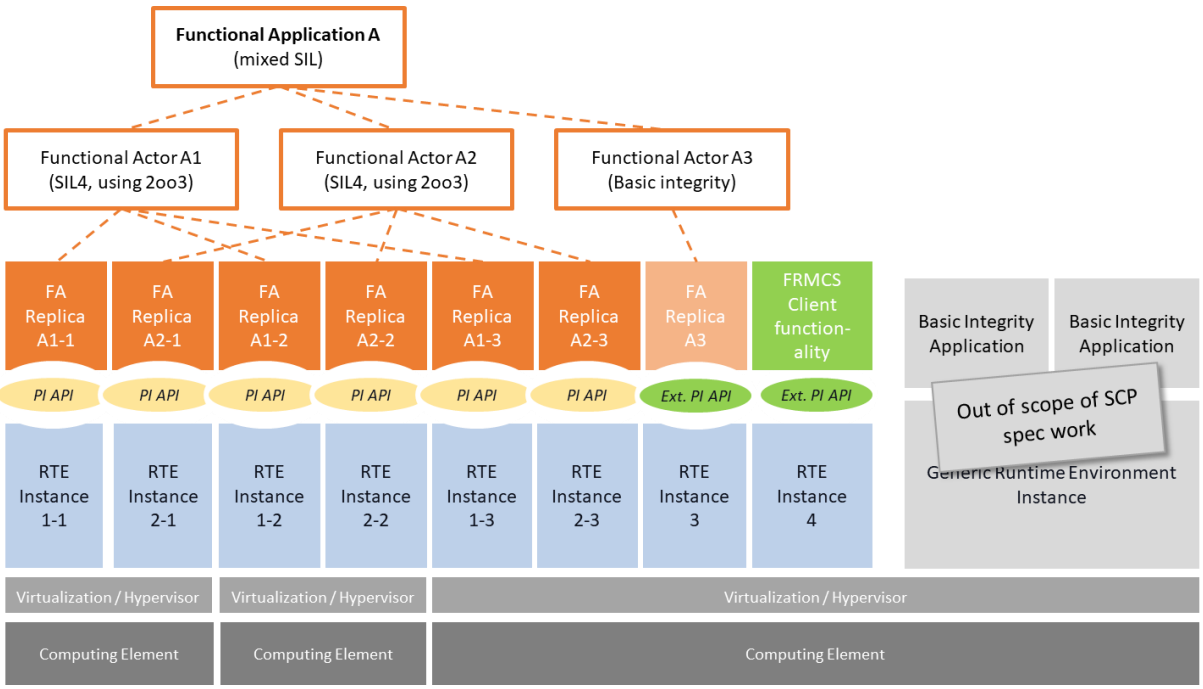


Figure 2. Computing Platform deployment where each Functional Actor Replica runs on a dedicated RTE instance.

Regarding vendor multiplicity, all different layers in Figure 2 (e.g., the Functional Actors, the RTEs, the virtualization / hypervisor / HW layer, and Gateway functions shown in green and detailed in Section 8) could be provided by different vendors (assuming that the RTEs of different suppliers would support the same virtualization environment, which is beyond the scope of this specification). Also, the two Functional Applications could of course be provided by different vendors. Different RTE instances could in principle also be provided by different vendors, as long as the two conditions listed in Section 4.1 are met, i.e.

- Functional Actor Replicas belonging to the same Functional Actor are assumed to run on RTEs provided by the same vendor (or on RTEs that are otherwise interoperable), to avoid the standardization of related inter-RTE interfaces;
- Functional Actors that communicate with each other (using Flows, as introduced in Section 6.2) are assumed to run on RTEs provided by the same vendor (or on RTEs that are otherwise interoperable), unless Gateway functions are involved (see Section 8).

Figure 3 shows the same deployment, but with the difference that now multiple Functional Actor Replicas are assumed to share a common RTE Instance. As, according to Section 4.1, one RTE is hosted on a single Computing Element, different Functional Actor Replicas may share a common RTE instance if mechanisms are provided on the underlying Computing Element that ensure sufficient independence (e.g., in CPU and memory usage) between the Functional Actors and Functional Actor Replica to fulfil the CENELEC norms EN 5012x (and if RTE is able to provide computing guarantees to all Functional Actor Replicas hosted).

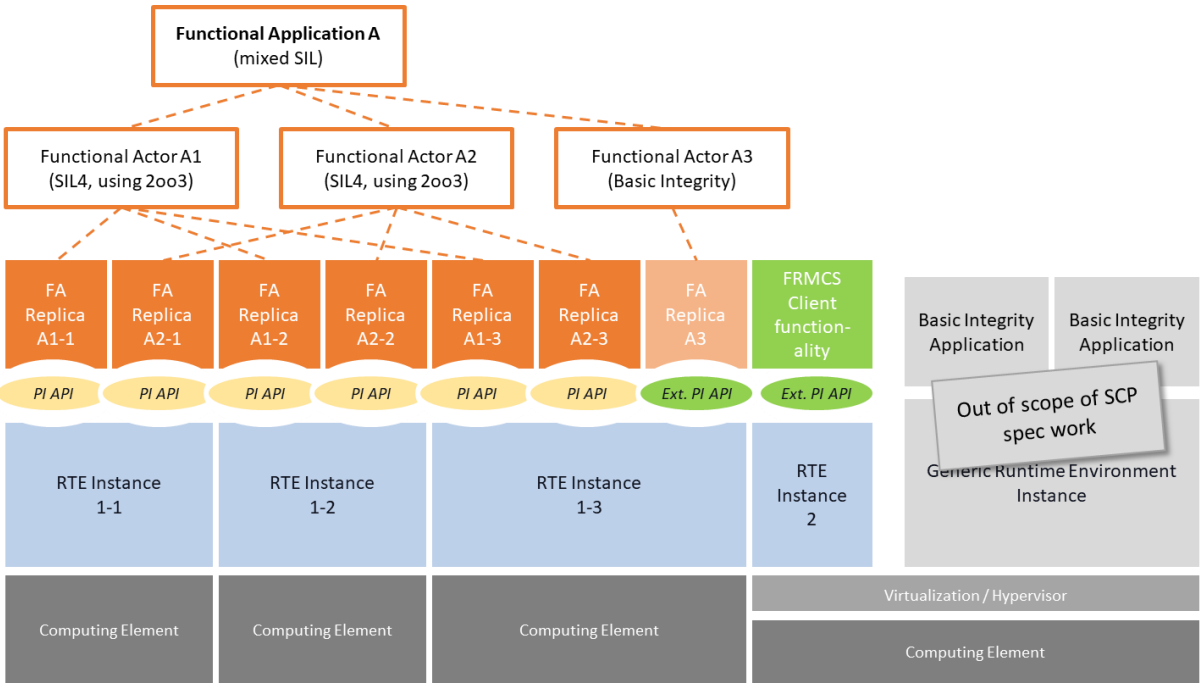


Figure 3. Computing Platform deployment where Replicas of different Functional Actors share the same RTE Instance.

Figure 4 shows yet another deployment variant where the Computing Entities are geographically distributed. In this particular example, Functional Actors A1 and A2 are now run in 4 Replicas each, of which 2 are placed in one data center, and 2 in another, in a 2x2oo2 configuration. This way, even if one data center completely fails, the two remaining Replicas in the other data center are sufficient (at least temporarily) to meet the safety requirements for these Functional Actors. This setup hence helps to increase availability. For the deployment examples shown in Figure 3 and Figure 4, the considerations and constraints regarding vendor multiplicity are the same as explained for Figure 2.

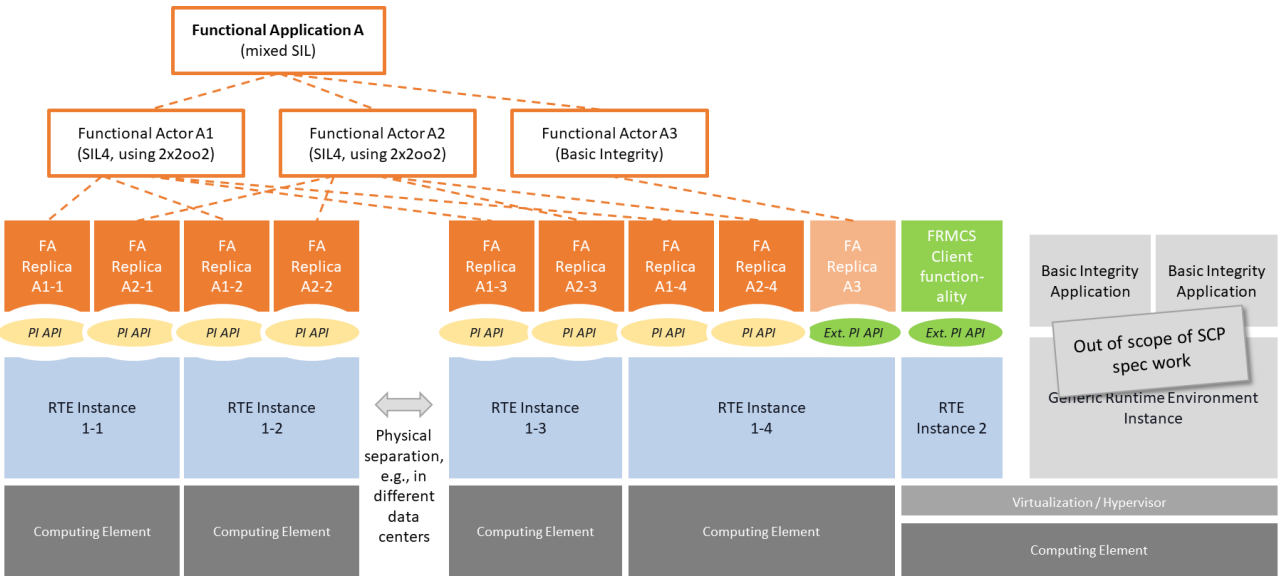


Figure 4. Physically distributed Computing Platform.

5 Key Paradigms and Guiding Principles for the PI API Design

The following **key paradigms** and **guiding principles** shall be followed for the design of the PI API:

- A key paradigm for the PI API design is that Functional Applications and Functional Actor (Replica) shall not be aware about safety and fault tolerance mechanisms such as composite fail safety provided by the Platform. More precisely, **it shall not be visible to a Functional Actor that itself (and also other Functional Actors) are being run by the Platform in Replica**, as all mechanisms related to voting, etc., are transparently performed by the Platform. This way, Application developers can focus on implementing the application logic - in consideration of possible safety related application conditions (SRACs) -, without needing to be concerned about fault tolerance, composite fail safety, etc.;
- The PI API design shall **maximally leverage API specifications** that are already available (to the extent that this is possible) in order to facilitate portability of existing Applications to the new API, and to ensure that platform realizations can maximally leverage existing implementations (open source etc.);
- **Restricted number of API functions:** For the ease of portability of applications among platform realizations, and for the ease of certification and acceptance, the number of API functions should be as few as possible;
- **Maximize common functions for onboard and trackside:** The API specification describes the superset of functions for onboard and trackside. It is clear, that there are functional differences in onboard and trackside railway subsystems, with corresponding explicit requirements on computing platforms, which may lead to easier implementation of separate functions for onboard and trackside. However, the PI API is intended to contain the same superset of functions, so the application developer can select functions based on the application demands, as shown in Figure 5.

  Note: In the work that has led to this first version of the PI API specification, no capabilities or functions of the API have been identified that would be needed only for onboard or only for trackside deployments

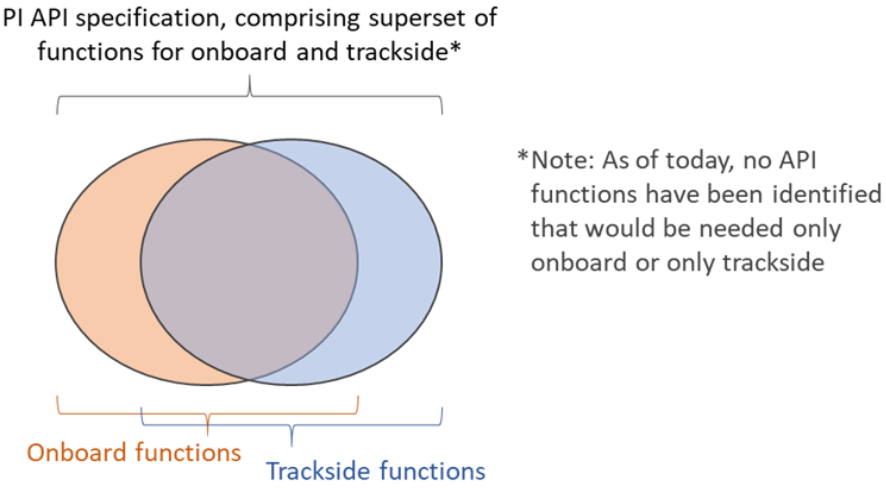


Figure 5. Scope of PI API specification regarding onboard and trackside.

- **Evolvability of PI API specs**: It is expected that the PI API can evolve over time (even if it is expected that already the first version deployed in real railway operation should be usable for 10-20 years). Future evolved versions are expected to be decently backward-compatible as shown in Figure 6. In this example, Functional Application version 1 is able to run on an RTE implementing PI API version 1 or version 2. Functional Application version 2, however, cannot run on an RTE implementing PI API version 1, as it requires functions that the PI API version 1 does not provide. Possibly, semantic versioning could be an option.

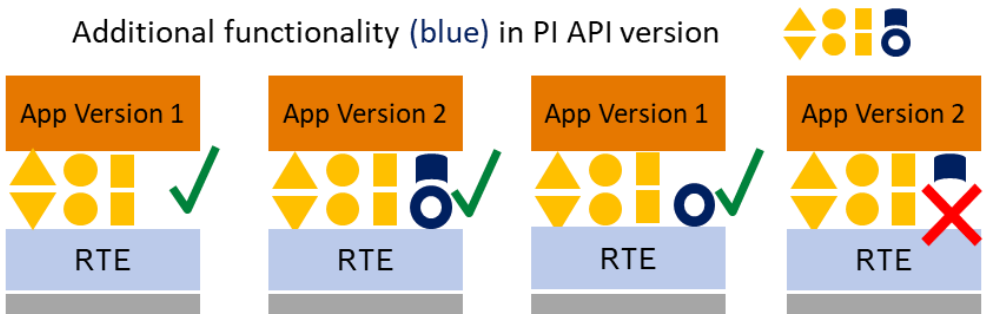


Figure 6. Illustration of the notion of backward compatibility.

- **PI API definition is programming language agnostic**: the PI API may be implemented in any programming language supporting application development up to Safety Integrity Level (SIL) 4. It is to be noted that the selected language must support the Control Command and Signaling (CCS) application requirements (e.g., real-time demands).

6 Messaging

6.1 Introduction and Key Paradigms

The exchange of information among Functional Actors is one of the key services that a Safe Computing Platform realization has to offer via the PI API to Functional Actors. Key messaging aspects are hence introduced in this section, with Section 6.2 introducing the general notion of Flows, Section 6.3 covering addressing, Section 6.4 covering properties of messages as such, and Section 6.5 referring to considerations on using the Data Distribution Service (DDS) for implementing messaging related aspects of the Safe Computing Platform.

Key paradigms behind the presented messaging concepts are:

- It should be transparent to a Functional Actor whether it is communicating to a local entity (i.e., residing on the same local Platform instance) or a remote entity (i.e., residing on a remote Platform instance, and possibly involving Safe Communication Protocols and/or gateway functions in between);
- The RTE takes care of the authentication and authorization of Functional Actors, so that Functional Actors can trust that the entities they are receiving messages from or transmitting messages to are the entities they claim to be;
  Note: It is assumed that specific RTE implementations will in this respect refer to a specific security standard (e.g., IEC 62443) and security level
- It should generally be transparent to Functional Actors whether they themselves, and the Functional Actors they are exchanging messages with, are replicated or not (Note: An exception to this are specific forms of messaging exchanges that will be explained later);
- A message is created by only one Functional Actor;

6.2 Flows

6.2.1 Introduction

Messages are exchanged between Functional Actors using Flows, which are defined as:

> ***Definition***: Flow: A Flow is a messaging relation between Functional Actors.

Flows may be joined or disjoined, registered or subscribed to by Functional Actors (possibly within constraints defined via configuration, as detailed later).

Once a Flow is established, Functional Actors may use it to exchange messages.

As detailed in the subsequent sections, a Flow may have various properties relating to the usage of voting, the usage of specific Safe Communication Protocols, quality of service, etc.

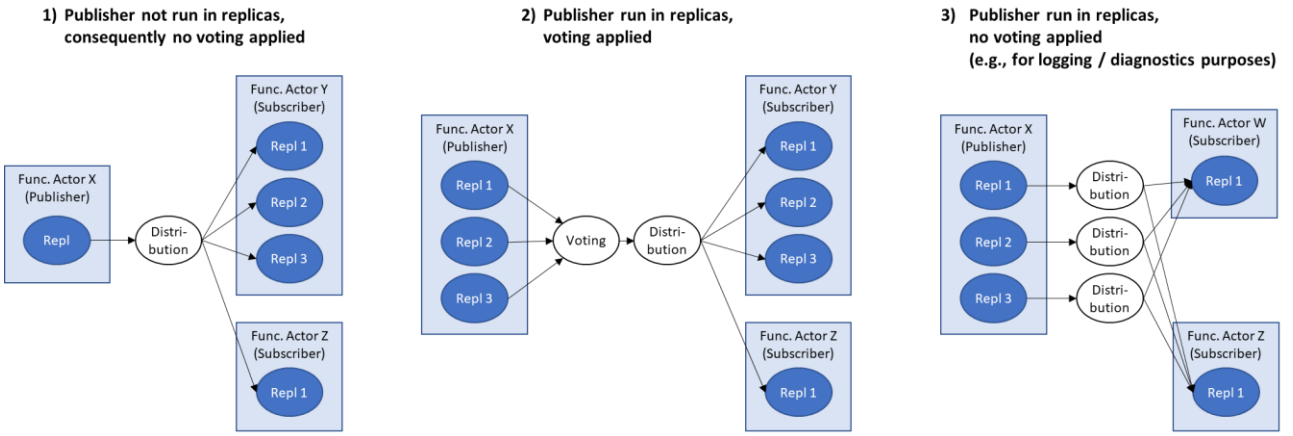
6.2.2 Uni-directional Flows

6.2.2.1 Introduction and Key Characteristics

Uni-directional Flows enable the transmission of messages from one or multiple publishing Functional Actors to one or multiple subscribing Functional Actors without implicit message acknowledgement from the receiving side.

A key static property of a Uni-directional Flow is the number of publishing Functional Actors, where we differentiate the cases:

- Exactly one publisher;
- Multiple publishers, possibly confined to a set of Functional Actor (types).



Note: While omitted from the figure for brevity, the displayed options would also apply in the case of multiple publishers and any constellation of subscribers

Figure 7. Options of applying voting and distribution in the context of Uni-directional Flows.

Uni-directional Flows may also differ by whether voting and distribution are applied to exchanged messages. In this respect, the following schemes are foreseen, as also shown in Figure 7:

1) A publisher is not run in Replicas, and its published messages are consequently not voted on but just distributed to all (Replicas of all) subscribers (in the example in Figure 7 these are Functional Actors Y and Z);

2) A publisher is run in Replicas, and its published messages are voted on and then distributed to all (Replicas of all) subscribers (in Figure 7 these are again Functional Actors Y and Z);

3) A publisher is run in Replicas, but its published messages are purposely not voted on, but directly distributed to all (Replicas of all) subscribers (in the example now non-replicated Functional Actors W and Z). In this case, assumed to be relevant only, e.g., for logging or diagnostics, each receiving Replica would hence redundantly receive the output of each publishing Replica (and would hence be aware that the publisher is running in Replicas).

For each publisher, the applied scheme depends on whether the publisher is replicated, and whether for the overall Flow voting is set to be applied or not.

If for the same Flow there are multiple publishers of which some are run in Replicas and others not, there may be a mix of the schemes shown in Figure 7.

For each publisher, the applied scheme depends on whether the publisher is replicated, and whether for the overall Flow voting is set to be applied or not, as detailed in Table 5.

Table 5. Application of voting in Uni-Directional Flows.

	Publisher not replicated	**Publisher replicated**
Voting set to be applied for this Flow	Scheme 1 from Figure 7 applied	Scheme 2 from Figure 7 applied
Voting set not to be applied for this Flow	Scheme 1 from Figure 7 applied	Scheme 3 from Figure 7 applied

Key characteristics of Uni-directional Flows are:

- Posted messages (on the same Flow and by the same publisher) are delivered to all subscribers in the exact same order as they have been published;

- Missing messages are identified by the platform (e.g., through the usage of message sequence numbers or some other platform-specific mechanism). The subscribed FAs are notified by the Platform whenever there are missing messages;
  Open Point: When and how exactly Functional Actors are informed about missing messages by the Platform is ffs
- Messages are time-stamped by the Platform, so that subscribers are able to determine how old messages are, and whether they should still be processed or discarded, etc. (see also Section 7.2);
  Open Point: It is tbd whether absolute or relative timestamps are to be used, whether there is a globally synchronized clock, etc.
- The RTE provides identification and authentication, so that from each message, subscriber(s) can determine and trust the identity of the publisher;
- If desired by the subscriber(s) (e.g., via configuration or upon subscription to a Flow), subscriber(s) are notified when a publisher "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond any more, or that it has not responded to past messages since a configured period of time).
  Note: Other forms of notifications (e.g., when new publishers appear, or when subscribers die) have been discussed, but then dropped, as there seems to be no strong need for these

6.2.2.2 Static properties of Uni-directional Flows

Static properties of Uni-directional Flows (which are defined in the configuration of a Flow), to the extent that these are exposed to the PI API or needed in configuration files and hence required to be specified, are:

- Name of the Flow (see Section 6.3);
- Constraints on the number of publishers (exactly one publisher or multiple publishers), as listed in Section 6.2.2.1;
- (Optionally) constraints on the Functional Actor(s) or Functional Actor Type(s) that are allowed to publish on the Flow, or that are allowed to subscribe to the Flow;
- Whether voting is suppressed even if publishers are running in Replicas (see Section 6.2.2.1), for instance for logging or diagnostics purpose;
- Message delivery options: Whether posted messages are received at most once or at least once;
- Desired maximum message delivery time (between publishing of a message by a Functional Actor to the Platform and the provision of the message to a receiving Functional Actor by the Platform). It should be noted that this is not to be understood as a guarantee - subscribers should be able to use / compare time stamps to see if the maximum delivery time was fulfilled, and to decide whether messages should still be processed or discarded, etc.

6.2.2.3 Dynamic properties of Uni-directional Flows

Dynamic properties of Uni-directional Flows (which can be dynamically changed throughout the lifetime of a Flow), to the extent that these are exposed to the PI API and hence required to be specified, are:

- Set of currently registered publishers and subscribers;

Open Point: It is to be clarified which information about publishers and subscribers of a Uni-directional Flow is made available to whom.

6.2.2.4 Configuration of Uni-directional Flows

Uni-directional Flows are defined in configuration files available to the Platform. It is left to the Platform implementation whether Uni-directional Flows are initialized (in the sense that CPU, memory or connectivity resources are reserved in any form) upon the initialization of Functional Actors that may use these Flows, or only when Functional Actors actually register to or subscribe to a Uni-directional Flow.

Open point: Whether it should also be possible for Functional Actors to dynamically create additional Uni-directional Flows is for future study

6.2.2.5 Register (as a publisher) to and unregister from a Uni-directional Flow

Functional Actors can dynamically register as a publisher to a Uni-directional Flow, as long as this registration fulfills all constraints in the number and type of publishers defined for this Flow (see Section 6.2.2.2).

If the Functional Actor fulfils all constraints w.r.t. the number and type of publishers defined for this Flow, the Platform accepts the registration of this Functional Actor to the Flow.

If the Functional Actor does not fulfil all constraints w.r.t. the number and type of publishers defined for this Flow, the Platform rejects the registration of this Functional Actor to the Flow.

Functional Actors can dynamically unregister from being a publisher to a Flow. The Platform may not reject this un-registration.

6.2.2.6 Subscribe to and unsubscribe from a Uni-directional Flow

Functional Actors can dynamically subscribe to a Uni-directional Flow, as long as this subscription fulfils any constraints that are possibly set on the Function Actor Type(s) allowed to subscribe to this Flow (see Section 6.2.2.2).

If the Functional Actor fulfils all constraints w.r.t. the number and type of subscribers defined for this Flow, the Platform accepts the subscription of this Functional Actor to the Flow.

If the Functional Actor does not fulfil all constraints w.r.t. the number and type of subscribers defined for this Flow, the Platform rejects the subscription of this Functional Actor to the Flow.

Functional Actors can dynamically unsubscribe from a Flow. The Platform may not reject this un-subscription.

6.2.3 Bi-directional Flows

6.2.3.1 Introduction and key characteristics

Bi-directional Flows enable the transmission of messages from exactly one requesting Functional Actor to exactly one responding Functional Actor, with an explicit response message to each request message.

Same as for Uni-directional Flows, Bi-directional Flows may involve voting or not. More specifically, the following schemes are supported:

1) Neither requestor nor responder replicated, no voting applied

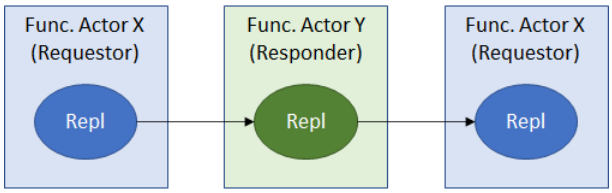


Figure 8. Possible scheme for Bi-directional Flow: Neither requester nor responder replicated, no voting applied.

2) Only requestor replicated, voting applied on request path only

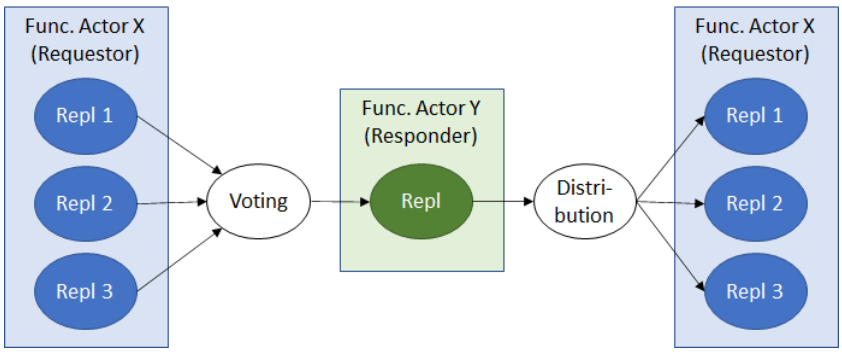


Figure 9. Possible scheme for Bi-directional Flow: Only requestor replicated, voting applied on request path only.

3) Only responder replicated, voting applied on response path only

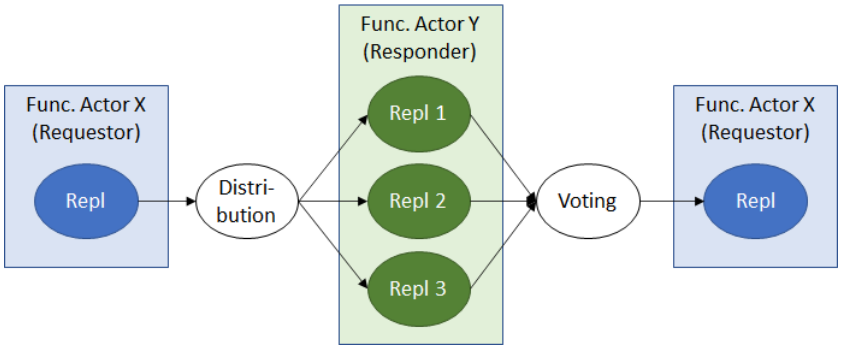


Figure 10. Possible scheme for Bi-directional Flow: Only responder replicated, voting applied on response path only.

4) Only responder replicated, no voting applied (note: mainly for debugging / logging purposes)

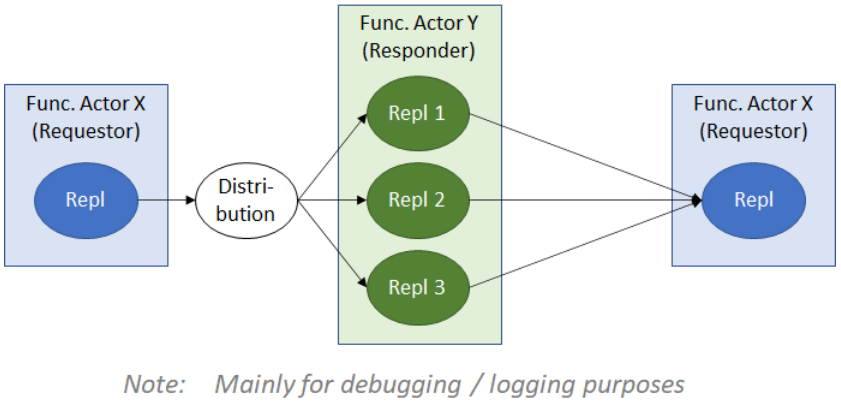


Figure 11. Possible scheme for Bi-directional Flow: Only responder replicated, no voting applied.

5) Both requestor and responder replicated, no voting applied ("bundling" scenario)

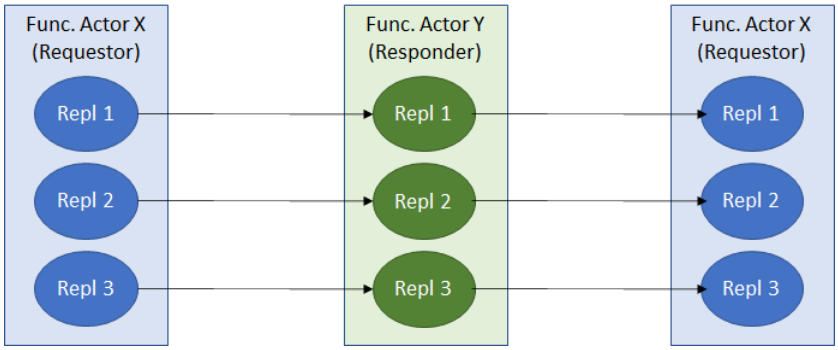


Figure 12. Possible scheme for Bi-directional Flow: Both requestor and responder replicated, no voting applied ("bundling" scenario).

Note: This scenario relates to the "bundling" of Functional Actors X and Y in this case. "Bundling" means that the voting on information flows among two or more Functional Actors is purposely skipped, for instance to improve latency, and voting is only applied (if applicable) on Flows between the bundled Functional Actors and Functional Actors outside the bundle. See Section 11.4 for a possible deployment including bundling. It is assumed that the depicted example scenario is only possible if Replica 1 of Functional Actor X and Replica 1 of Functional Actor Y run on the same Computing Element. Further, it is assumed that in this case if Functional Actor X or Functional Actor Y provide inconsistent output (as determined via voting) to another Functional Actor, both the related Replicas of Functional Actor X and Functional Actor Y have to be restarted, as the platform cannot identify whether an error has occurred in Functional Actor X or Functional Actor Y.

6) Both requestor and responder replicated, voting applied

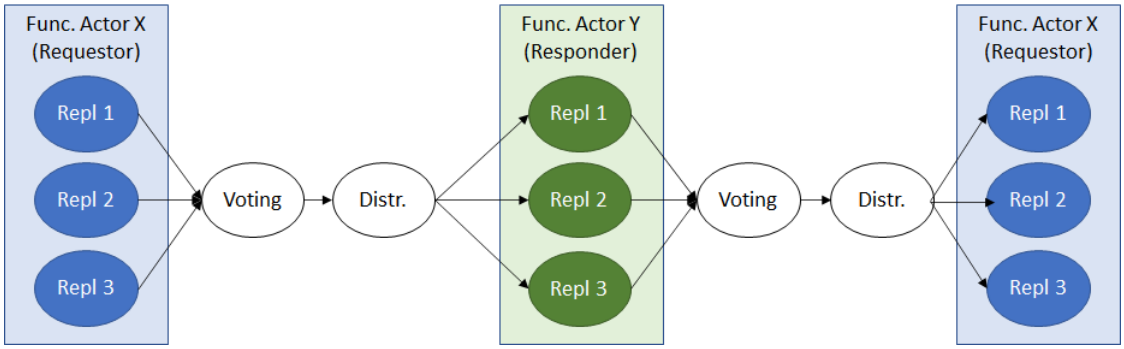


Figure 13. Possible scheme for Bi-directional Flow: Both requestor and responder replicated, voting applied.

Key characteristics of Bi-directional Flows are:

- Posted messages are received by the receiver in the exact same order as they have been sent. This applies to both messages sent by the requester, and the response messages sent by the responder;
- The Platform delivers messages (both requests and responses) exactly once (subject to a possible timeout); *Note: The Platform may use mechanisms to replicate messages to increase reliability, but the receiving Functional Actor(s) ultimately only obtain each message exactly once (unless an error condition occurs);*

- The Platform, if so configured for the Flow, notifies the involved Functional Actors if the desired maximum message delivery time is exceeded. Also, this applies to both request and response messages;
- Messages are time-stamped by the Platform, so that the involved Functional Actors are able to determine how old messages are, and whether they should still process or discard them, etc., as also covered in Section 7.2.2;
  Open Point: It is tbd whether absolute or relative timestamps are to be used, whether there is a globally synchronized clock, etc.
- The Platform informs the requesting Functional Actor when the responding Functional Actor has joined the Flow (for the first time, or, e.g., after a crash)
- The Platform informs the requesting Functional Actor when the responding Functional Actor "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond anymore, or that it has not responded to past messages since a configured period of time);
- The Platform informs the responding Functional Actor when the requesting Functional Actor "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond any more, or that it has not responded to past messages since a configured period of time);
- The Functional Actors involved in a Bi-directional Flow can trust the identity of the requesting / responding side.

6.2.3.2 Static properties of Bi-directional Flows

Static properties of Bi-directional Flows (which are defined in the configuration of a Flow) are:

- Name of the Flow (see Section 6.3);
- The identities of the requesting and receiving Functional Actors;
- Whether or not voting is applied;
- Desired maximum message delivery time (between sending and receiving). It should be noted that this is not to be understood as a guarantee - receivers should be able to use / compare time stamps to see if the maximum delivery time was fulfilled, and to decide whether messages should still be processed or discarded, etc.
- Whether or not requesting and receiving Functional Actors are informed about exceeded desired maximum message delivery times.

6.2.3.3 Dynamic properties of Bi-directional Flows

None

6.2.3.4 Configuration of Bi-directional Flows

Bi-directional Flows are defined in configuration files available to the Platform. It is left to the Platform implementation whether Bi-directional Flows are initialized (in the sense that CPU, memory or connectivity resources are reserved in any form) upon the initialization of Functional Actors that may use these Flows, or only when Functional Actors join these.

6.2.3.5 Joining / disjoining Bi-directional Flows

Functional Actors can dynamically join or disjoin Bi-directional Flows for which they are configured to be either requestor or respondent. The Platform may not reject such join or disjoin request as long as the Functional Actor is listed as either requestor or respondent of the Flow.

A Bi-directional Flow can be used for requests from the requesting Functional Actor once both sides have joined the Flow.

6.3 Addressing

Addressing is built around

- **Functional Actor Names** (known to the Platform and Functional Actors - in the latter case for instance such that Functional Actors can identify from which FA a message is);
- **Replica Identifier**, from which a Functional Actor receiving a message can tell from which Replica of the publishing Functional Actor the message is (in case of the debugging / logging related messaging schemes introduced in Sections 6.2.2.1 and 6.2.3.1). A single Functional Actor Replica can hence be uniquely identified by its **Functional Actor Name** and its **Replica Identifier**;
- **Functional Actor Types**. As introduced in Section 4.1, there may be multiple instantiations of Functional Actors of the same type, for instance multiple instances of a Functional Actor type "fixed object controller" – all would be based on the same software but have different configurations. For a specific Uni-directional Flow it may be defined that only Functional Actors of specific type(s) are allowed to register to or subscribe to a Flow (see Section 6.2.2.1);
- **Flow Names** (known to the Functional Actors and to the Platform).

It is in general assumed that Functional Actor Names and Flow Names must at least be unique on the same local Platform deployment.

When Functional Actors (un)register or (un)subscribe from Uni-Directional Flows or request to join/disjoin Bi-directional Flows, they use (unique) Flow Names.

6.4 Messages

When Functional Actors receive messages, they obtain the following information from the Platform (either directly through the message, or through some other context information):

- Flow Name that the message belongs to
  Note: This is inherently known to a receiving Functional Actor due to the usage of a handle related to the Flow;
- Functional Actor Name and Replica Identifier (if applicable, in the logging/debugging related cases defined in Sections 6.2.2.1 and 6.2.3.1) of the Functional Actor from whom the message (in the case of a Bi-directional Flow a request or response) originates;
- Time stamp when the message was originally submitted by the Functional Actor to the Platform;
- Actual message payload (only readable by the receiving Functional Actors and transparent to the Platform).

6.5 Possible usage of DDS as Basis for SCP Messaging

In the course of the work that has led to this specification, there has also been elaboration on the potential usage of the Data Distribution Service (DDS) as an existing protocol in the context of the SCP. As a result, it is described in a separate document [6] how the aforementioned notion of Flows in the SCP could be implemented via DDS. One additional aim is here to show the practicability of the SCP messaging specification, though other implementations that are not based on the DDS protocol would be equally possible.

7 Execution Model and Timing Behavior

7.1 Execution Model

A Functional Actor Replica obtains execution time based on the following kinds of triggers (or combinations of these):

- timer-based, i.e., in configured regular intervals, or in the form of one-shot timers;
- event-based, i.e., upon receipt of (certain types of messages);
- timer- and event-based, i.e., the Functional Actor obtains execution time in regular intervals, or in the form of one-shot timers, only if (certain types of) messages have (or have not) been received.

7.2 Timing

7.2.1 Introduction

Key design principles of the Safe Computing Platform API related to timing are:

- Specific Platform implementations shall have maximum degree of freedom in scheduling Functional Actor Replicas, as long as the following very confined list set of requirements is fulfilled;
- It is ultimately in the responsibility of the Functional Actors to determine whether messages have been received in time, whether enough processing time has been provided, etc., and to react if this is not the case.

7.2.2 Messaging-related Timing Requirements

The Platform shall supervise that messages are not delayed beyond the maximum message delivery times defined for the related Flows, as defined in the context of configuring Flows (see Section 6.2).

The Platform shall inform receiving Functional Actors (if so configured) when maximum message delivery times are exceeded.

In addition, the Platform shall complement messages with timestamps, so that receiving Functional Actors can check whether and how strongly received messages are outdated and possibly take appropriate action (i.e., either discard such messages or take other action).

Open Point: It is tbd when exactly a timestamp is inserted to a message, and whether these are relative or absolute timestamps.

This requires the notion of synchronized platform clocks (also among distributed platforms) at least to the extent/granularity (e.g., on the order of tens of ms) that is required to detect outdated messages. *Note: This is for instance already achieved by RaSTA.*

7.2.3 Scheduling-related Timing Requirements

When a Functional Actor Replica is invoked to process messages (either timer-based or message-triggered, as described in Section 7.1) it has to get enough processing budget to conclude the message processing before the next invocation. An option could be that in the configuration of a Functional Actor one would list in which time intervals the function needs how much processing budget (e.g., in DMIPS) - if the Platform is not able to provide this extent of processing budget, it shall reject the initialization of a Functional Actor.

Note: This approach is seen as suboptimal, but no better approach has been proposed so far.

The Platform shall monitor whether Functional Actor Replicas are able to conclude processing within a defined time period. If these are not able to conclude, the Platform shall inform the Functional Actor about this and (if configured) shut down the Functional Actor.

For development purposes, or for non-safety relevant Functional Actors, Replicas may themselves measure if they obtain sufficient processing budget (for instance through observing the number of pending incoming messages) and potentially take appropriate action (e.g., reduce the number of subscriptions, shift load to other Functional Actors, or enter a degraded mode).

In general, when the Platform invokes the multiple Replicas of the same Functional Actor, it shall ensure that individual Replicas process the same messages as their counterparts, as otherwise it could not be guaranteed that the Replicas yield the exact same output. It is left to the Platform implementation how to achieve this.

7.2.4 Time Stamps

The Platform shall upon request provide two kinds of time stamps to Functional Actors:

- One time stamp (“unsynchronized time”) that corresponds to the time at the point when a Functional Actor Replica requests this (and for which different Replicas of the same Functional Actor may obtain a different result);
- One time stamp (“synchronized time”) that is exactly the same for all Replicas of the same Functional Actor requesting this (even if there is a time lag in when this is requested). This is important if the time stamp has an impact on any (voted) output of the Functional Actor.

Functional Actors shall ensure that they only use the "unsynchronized time" in cases where this does not have impact on any (voted) output of the Functional Actor (i.e., only for logging or diagnostics purposes).

Open Point: It is tbd whether there is a globally synchronized clock, how distributed clocks are synchronized, etc.

8 Gateway Concept

8.1 Introduction and Basic Design Paradigms

A Gateway concept is required to enable that Functional Actors can communicate with entities outside a local Safe Computing Platform deployment.

Basic design paradigms are here:

- it should not be (explicitly) visible to a Functional Actor whether it is communicating to another entity on the same physical Safe Computing Platform realization or a remote entity (obviously, there would be a difference in latency, etc.);

- it should in particular be possible to port Functional Actors in between different physical Safe Computing Platform realizations without having to change the Functional Actor implementation w.r.t. connectivity;
- communication protocols (both safe communication protocols like RaSTA and non-safe communication protocols like, e.g., FRMCS-related protocols) should be separated from the Application, so that they can evolve independently and can be provided by different vendors.

While in [4] different options for handling communication protocols were listed, the following approach seems to best satisfy the points above and fit to the prior agreements on Messaging:

- Safe communication protocols (e.g., RaSTA) which are specific to the application domain (e.g., defined for CCS-related applications) and applied end-to-end among Functional Actors hosted on different physical Safe Computing Platform realizations (and not only applied locally within one physical Platform realization) are implemented as dedicated Functional Actors, also being replicated like the (safe) Functional Actors using these;
  Note: The specification of safe communication protocols is beyond the scope of this specification.
- Non-safe communication protocols (e.g., FRMCS related protocols) are also implemented as Functional Actors, but these are not safety-relevant, not replicated, and have access to a so-called “extended PI API” (i.e., with direct access to the TCP/IP or UDP/IP protocol stack of the Platform), which is further detailed in Section 10.1. These are in the following referred to as “Gateway functions”;
  Note: The term “non-safe” here refers to the fact that these protocols do not need safety-certification, as there are other mechanisms (for instance provided by the Safe Communication Protocol Support) that ensure safety.

8.2 Gateway Example for Illustration

The aforementioned gateway approach is now illustrated through an example depicted in Figure 14. It is here assumed that a Functional Actor FA-X (orange block in the figure) needs to enter a request/response relationship to an external entity FA-Y (grey block in the figure).

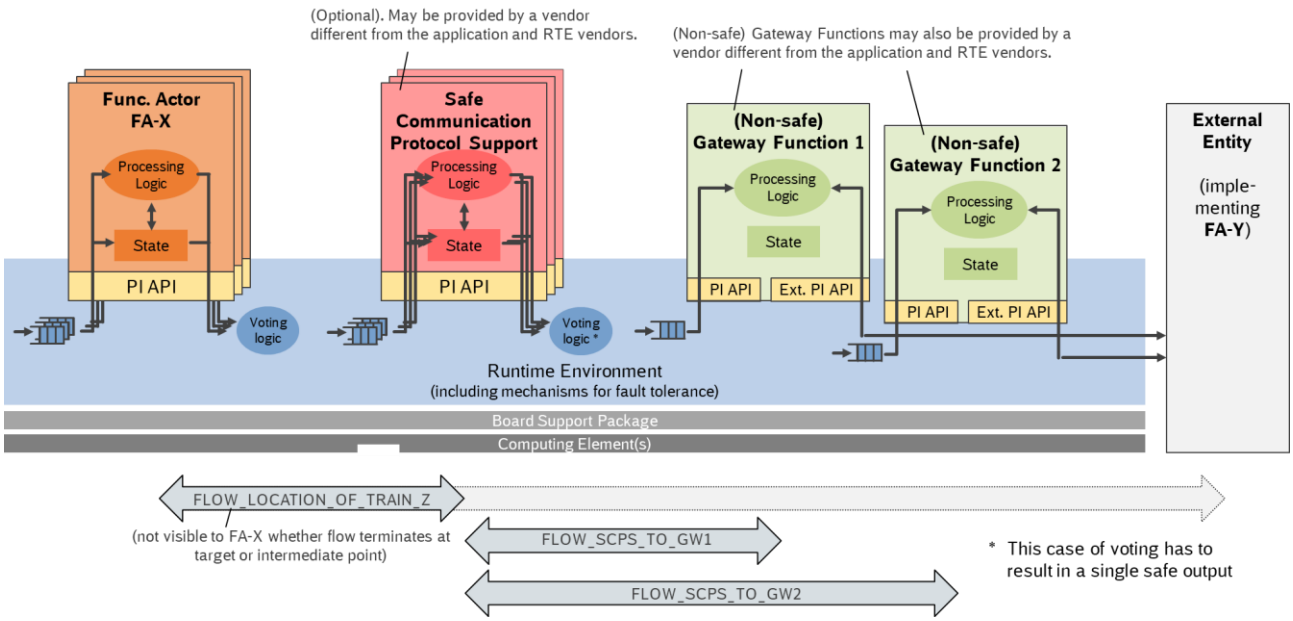


Figure 14. Example for usage of a Gateway approach.

In terms of Flow setup, the following would happen:

- The Functional Actor FA-X would join a Bi-directional Flow with name, say, "FLOW_LOCATION_OF_TRAIN_Z" (the Flow name should be defined such as to be agnostic of whether the flow terminates at a co-located or remote entity);
- The physical Platform realization that Functional Actor FA-X is running on would determine (through the prior configuration of this Flow) that the communication counterpart should be the "Safe Communication Protocol Support" (red box in Figure 14). The “Safe Communication Protocol Support”, possibly provided by a vendor different from the Platform vendor, would be a safe function that is replicated and to which voting is applied, same as for FA-X itself;
- The " Safe Communication Protocol Support" itself would join Flows to one or multiple (in this example two) Gateway functions. In the example, we call the two related flows "FLOW_SCPS_TO_GW1" and "FLOW_SCPS_TO_GW2";
- The two Gateway functions are not replicated and have access to the Extended PI API, as detailed in Section 10.1. Both Gateway functions represent independent grey/black communication channels to the external entity. It should be noted that the two Gateway functions may follow the same implementation (and just differ in configuration), or may also be completely different (e.g., one may establish a wireline connection to the external entity, while the other establishes a wireless link). Naturally, the involved Gateway functions should run on different Computing Elements for availability reasons.

For each request sent from FA-X to FA-Y, the following happens:

- FA-X submits a request message on Flow "FLOW_LOCATION_OF_TRAIN_Z", which is voted on and distributed to all Replicas of the Functional Actor "Safe Communication Protocol support";
- Functional Actor "Safe Communication Protocol support" implements the actual Safe Communication Protocol (e.g., RaSTA), the output being messages submitted on the Flows "Flow_SCPS_TO_GW1" and "Flow_SCPS_TO_GW2" to the Gateway functions;
- The output messages of "Safe Communication Protocol support" are voted on and sent to the (single) instances of the Gateway functions. It is in this context important that this voting steps yields a single safe output. Together with the respective communication links to the external entity, the Gateway functions constitute grey/black channels to the external entity;
- The two Gateway functions implement any non-safe protocols and may directly access the TCP/IP or UDP/IP stack of the Platform.

Figure 15 depicts, for the previous example, how the different entities contribute to the overall communications protocol stack toward the external entity. Note that in this figure only one of the two non-safe gateway functions is depicted for brevity. The second one would obviously establish a similar communications protocol stack toward the external entity.

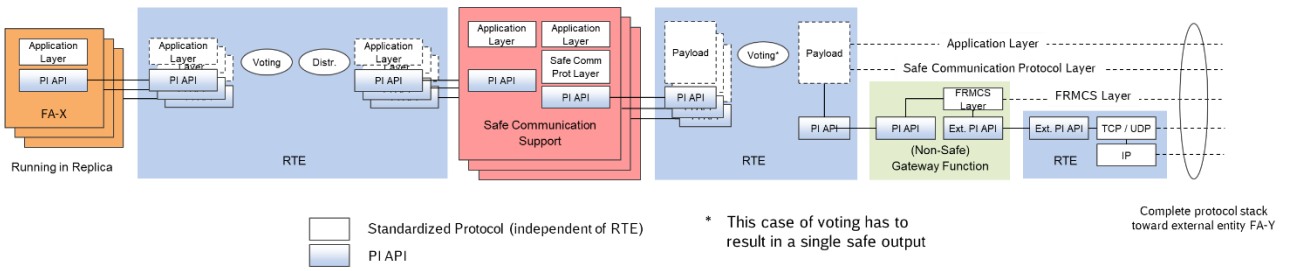


Figure 15. Contribution of the involved entities in the protocol stack used toward the external entity.

9 Fault, Error and Failure Handling and Recovery

This section describes how the Platform should handle faults, errors and failures, and which implications this has on the API among Applications and Platform.

9.1 Used Terminology

As shown in Table , the subsequent sections follow the terminology used in EN 50129:2018, but apply this to the specific context of the Safe Computing Platform.

Table 6. Terminology related to faults, errors and failures.

Term	Definition in EN 50129:2018	Usage of term in context of Safe Computing Platform	Specified Platform behavior
Fault	Abnormal condition that could lead to an error in a system	abnormal condition that could lead to an error in a system	See Section 9.2
Error	Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition	A number of Functional Actor Replicas and/or Computing Elements is impacted, but to an extent that can still be mitigated by restarting Replicas or moving them to other Computing Elements, so that Functional Actors are NOT impacted. *Example: A Functional Actor Replica provides different output than its counterpart Replicas (or no output at all).*	See Section 9.3
Failure	Loss of ability to perform as required	Functional Actor(s) are impacted (in the way that these cannot perform anymore and/or an application message is lost)	See Section 9.4

9.2 Fault Detection and Response

It is left to the specific Platform implementation to which extent it detects and handles faults, as long as it complies to EN 50129:2018. It is also left to the specific Platform implementation to decide when a fault (according to EN 50129:2018) is to be flagged as an error.

In any case, the Platform shall ensure fault containment between Functional Actor Replicas by sufficient independence according to EN 50129:2018, as one of the prerequisites to fulfil the safety case.

9.3 Error Detection and Response

The Platform shall detect and handle errors according to EN 50129:2018 (with both the definition of “error” and the handling of these according to EN 50129:2018).

Beyond what is covered in the stated norm, the Platform shall explicitly take the recovery and informational actions listed in Table 7. In addition, the Platform may also take custom actions (e.g., trigger certain maintenance actions or, eventually, shutdown), but these are beyond the scope of this specification.

Table 7. Recovery and informational actions to be taken by the Platform in the case of errors.

Entity affected by error according to EN 50129:2018	Actions to be taken by Platform
Functional Actor Replica	• Restart the Functional Actor Replica and recover its state; • Inform Functional Actors that have requested diagnostics information about the affected Functional Actor Replica via a "State Information Interface". *Open Point: The details of the State Information Interface are for future study.*
Computing Element	• Restart the Computing Element and recover or restart all affected RTE instances and Functional Actor Replicas; • Inform Functional Actors that have requested diagnostics information about the affected Functional Actor Replica(s) via a "State Information Interface". *Open Point: The details of the State Information Interface are for future study.*

If the recovery actions defined in Table 7 are not successful (e.g., due to repeated failure of a Replica or a Computing Element), or because more Replicas of the same Functional Actor are affected than the MooN configuration allows for, this constitutes a failure that is treated as defined in the following section.

9.4 Failure Response

As defined in Table 6, a failure implies that one or multiple Functional Actor(s) are not able to perform as required. In this case, the Platform shall take the following action:

- It informs other Functional Actor(s) which are in a Flow relation to the affected Functional Actor(s) that their communication counterpart is affected (see Sections 6.2.2.1 and 6.2.3.1);
- It informs Functional Actors that have requested diagnostics information about the affected Functional Actor Replica(s) via a "State Information Interface";
  Open Point: The details of the State Information Interface are for future study.
- It informs involved Functional Actor(s) in a Flow that the Flow is impaired (not because the communication counterpart is affected, but because the communications stack itself is affected).
  Open Point: It is to be further discussed whether and in which detail this information is needed.

10 API Considerations

10.1 Introduction and Basic Considerations

As already mentioned in Section 8.1, this specification differentiates between two different variants of API between Functional Applications and Platform:

- **PI API**: A constrained API for safe, replicated Functional Actors, as detailed in Section 10.2;
- **Extended PI API**: An extended API for non-replicated Functional Actors (e.g., with TCP/IP, UDP/IP protocol stack access, as for instance needed by Gateway functions described in Section 8), as detailed in Section 10.3 .

Both variants of API need to facilitate the porting of applications onto the runtime environment as well as portability between different RTE implementations. Furthermore, both variants of the API are expected to meet the following requirements: They have to be

- suitable for real-time applications;
- language agnostic (as stated in Section 5);
- established in the field;
- defined such that the error states are well understood and stable.

10.2 PI API, as used by Safe, Replicated Functional Actors

On top of the requirements listed in the previous section, the PI API must obviously be suitable for safe applications (i.e., fulfil EN 5012x up to SIL4). As such it is our goal, to limit the scope of this API to the functionalities really needed and then to extend it with functionality that eases development of safety critical applications. This shall be achieved by providing APIs that enable the platform to provide common network security, redundancy and replication services, so that applications do not need to implement such services by themselves.

Based on the considerations in the previous section, it becomes clear that one should aim at choosing subsets of existing, well-established APIs, with minor modifications where needed, for the PI API. Possible choices in this respect are POSIX (in particular IEEE POSIX 1003.1 and IEEE POSIX 1003.13) and ARINC 653 APIs, as these are already well established, and subsets of these are already used in certified and approved systems up to SIL4.

Careful evaluation of these APIs has led to the conclusion that the POSIX API is a good overall fit. It provides the freedom to allow for the usage of the full API for non-safety relevant Functional Actors and a reduced subset for safety-relevant Functional Actors. However, it was concluded that the POSIX real-time profiles are not a perfect fit, which is why, in the following, subset definitions for safety-relevant Functional Actors are defined.

Note that all functions described in Sections 10.2.1 and 10.2.2 are invoked from the side of the Functional Actor (Replicas).

10.2.1 Subselection of POSIX Functions for the PI API

In Table 8, the POSIX functions (grouped according to the header files as defined in [7]) are preliminarily commented w.r.t. their possible inclusion in the PI API for safety-relevant (and consequentially replicated) Functional Actors.

It shall be noted that Table 8 only gives a rough indication of which POSIX functions could be included in the PI API, at the granularity of POSIX headers. The common understanding is that an in-

depth analysis has to be done for each function in POSIX whether this is suitable for the PI API, especially considering certifiability for SIL4 applications, which will then be captured in an evolved version of this specification. In this respect, user needs obviously have to be taken into account, and hence any further input in this direction is explicitly appreciated.

Table 8. Proposal for inclusion of POSIX functions in PI API for safety-relevant Functional Actors.

Header	Description	Proposal for inclusion in PI API for safe (and replicated) Functional Actors	
aio.h	asynchronous input and output	×	Not included
arpa/inet.h	definitions for internet operations	×	Not included
assert.h	verify program assertion	✓	Included
complex.h	complex arithmetic		Open Point: To be concluded later
cpio.h	cpio archive values		Open Point: To be concluded later
ctype.h	character types	(✓)	Included excluding functions that take a locale as argument.
dirent.h	format of directory entries	✓	Included
dlfcn.h	dynamic Linking		Open Point: To be concluded later
errno.h	system error numbers	✓	Included; Open Point: An extension to account for the distributed nature of flows needs to be defined and added
fcntl.h	file control options	✓	Included
fenv.h	floating-point environment	✓	Included
float.h	floating types	✓	Included
fmtmsg.h	message display structures		Open Point: To be concluded later
fnmatch.h	filename-matching types		Open Point: To be concluded later
ftw.h	file tree traversal		Open Point: To be concluded later
glob.h	pathname pattern-matching types		Open Point: To be concluded later
grp.h	group structure		Open Point: To be concluded later
iconv.h	codeset conversion facility		Open Point: To be concluded later
inttypes.h	fixed size integer types		Open Point: To be concluded later
iso646.h	alternative spellings	×	Not included
langinfo.h	language information constants		Open Point: To be concluded later
libgen.h	definitions for pattern matching functions		Open Point: To be concluded later
limits.h	implementation-defined constants	✓	Included
locale.h	category macros		Open Point: To be concluded later
math.h	mathematical declarations		Open Point: To be concluded later
monetary.h	monetary types		Open Point: To be concluded later
mqueue.h	message queues (REALTIME)	×	Not included, as not required and superseded by *flows.h*
ndbm.h	definitions for ndbm database operations	×	Not included
net/if.h	sockets local interfaces	×	Not included
netdb.h	definitions for network database operations	×	Not included
netinet/in.h	Internet address family	×	Not included

netinet/tcp.h	definitions for the Internet Transmission Protocol (TCP)	×	Not included
nl_types.h	data types		Open Point: To be concluded later
poll.h	definitions for the poll() function	×	Not included
pthread.h	Threads	(✓)	A subset is required to safely handle threads. This subset shall support the following functions: • pthread_attr_init • pthread_attr_setdetachedtstate, • pthread_attr_setschedparam • pthread_create, • pthread_mutex_init • pthread_mutex_lock • phtread_mutex_unlock • pthread_mutexattr__init • pthread_self
pwd.h	password structure		Open Point: To be concluded later
regex.h	regular expression matching types		Open Point: To be concluded later
sched.h	execution scheduling	(✓)	A subset is included for thread scheduling during the initialization phase of an actor. Process scheduling is not supported.
search.h	search tables		Open Point: To be concluded later
semaphore.h	Semaphores	×	Not included. PThread Mutex is provided for locking.
setjmp.h	stack environment declarations		Open Point: To be concluded later
signal.h	Signals	(✓)	Subset included Open Point: Exact subset to be concluded later
spawn.h	spawn (ADVANCED REALTIME)		Open Point: To be concluded later
stdarg.h	handle variable argument list	✓	Included
stdbool.h	boolean type and values	✓	Included
stddef.h	standard type definitions	✓	Included
stdint.h	integer types	✓	Included
stdio.h	standard buffered input/output	(✓)	A subset is included and can be used during the initialization phase of a functional actor. Open Point: This subset is to be defined in the future.
stdlib.h	Standard library definitions	(✓)	Included, whereby a subset is only allowed during the initialization phase of a functional actor. Open Point: This subset is to be defined further in the future.
string.h	String operations	(✓)	Included, subset POSIX_C_LANG_SUPPORT
strings.h	String operations		Open Point: To be concluded later
stropts.h	STREAMS interface (STREAMS)	×	Not included

sys/ipc.h	XSI interprocess communication access structure	×	Not included
sys/mman.h	memory management declarations	(✓)	A subset is Included. Open Point: Exact subset to be defined later
sys/msg.h	XSI message queue structure	×	Not included
sys/resource.h	definitions for XSI resource operations	×	Not included
sys/select.h	Select types	✓	Included. Open Point: Tbd whether in addition epoll should be used
sys/sem.h	XSI semaphore facility	×	Not included
sys/shm.h	XSI shared memory facility	×	Not included
sys/socket.h	main sockets header	×	Not included
sys/stat.h	data returned by the stat() function	✓	Included
sys/statvfs.h	VFS File System Information structure	×	Not included
sys/time.h	time types	✓	Included; Additionally, replicated timers are defined in *sync_time.h*, see 10.2.2.2 for details.
sys/times.h	file access and modification times structure	✓	Included
sys/types.h	data types	✓	Included
sys/uio.h	definitions for I/O operations		Open Point: To be concluded later
sys/un.h	definitions for UNIX domain sockets	×	Not included
sys/utsname.h	system name structure		Open Point: To be concluded later
sys/wait.h	declarations for waiting	×	Not included
syslog.h	definitions for system error logging	✓	Included
tar.h	extended tar definitions		Open Point: To be concluded later
termios.h	define values for termios		Open Point: To be concluded later
tgmath.h	type-generic macros		Open Point: To be concluded later
time.h	time types	✓	Subset POSIX_TIMER included. To support the SCP timing needs, additional clocks need to be defined, as detailed in Section 10.2.2.2.
trace.h	Tracing	✓	Included
ulimit.h	ulimit commands	×	Not included
unistd.h	standard symbolic constants and types		Open Point: To be concluded later
utime.h	access and modification times structure	✓	Included
utmpx.h	user accounting database definitions	×	Not included
wchar.h	wide-character handling		Open Point: To be concluded later
wctype.h	wide-character classification and mapping utilities		Open Point: To be concluded later
wordexp.h	Word-expansion types		Open Point: To be concluded later

10.2.2 API extensions

In the following sections, more details will be provided on extensions to the POSIX API.

10.2.2.1 Flows

To support the notion of Flows introduced in Section 6.2, it is expected that the functions listed in Table 9 are required.

Table 9. Additional required functions related to Flows.

Function	Description
fl_open	Used to open a Uni-directional or Bi-directional Flow.
fl_close	Used to close a Uni-directional or Bi-directional Flow.
fl_send	Used to send a message. *Open Point: It is tbd whether addition functions for data serialization are required*
fl_receive	Used to receive messages. *Open Point: It is tbd whether addition functions for data serialization are required*
fl_getattr	Used to obtain attributes of a Flow.
fl_setattr	Used to set attributes of a Flow.

An example for how these functions could possibly be defined in detail, including a possible usage of parameters, is given in Annex A.1.

10.2.2.2 Timers

As mentioned in Section 7.2.4, it is required that the Platform can provide two forms of time stamps to Functional Actors, namely unsynchronized and synchronized time stamps. It appears that the functions related to timers provided in POSIX are sufficient also for the SCP, but additional synchronized timers should be introduced. Overall, the usage of timers for the SCP would then be as follows:

- **Unsynchronized** (i.e., where different Replica of a Functional Actor may get different responses): For this, timers existent in POSIX could be reused;
- **Synchronized** (i.e., where different Replica of a Function Actor obtain the exactly same result when requesting a time stamp): For this, new timers should be introduced:
    - SYNCHRONIZED_REALTIME: Potentially also synchronized to external clock sources
    - SYNCHRONIZED_MONOTONIC: Increasing time

How exactly these additional timers could be defined is listed in Annex A.2.

10.2.2.3 Configuration Management

To support configuration management, it is expected that the functions listed in Table 10 are required.

Table 10. Additional required functions related to configuration management.

Function	Description
get_configuration_labels	Used to obtain configuration labels.

get_default_configuration	Used to obtain the default configuration.
get_configuration_by_label	Used to obtain configuration information by label.

How exactly these additional functions could be defined is listed in Annex A.3.

10.2.2.4 Checksum Functions

To support checksum related functions, it is expected that the functions listed in Table 11 are required.

Table 11. Additional required functions related to checksums.

Function	Description
messagedigest_create	Used to create a message digest.
digest	Used to perform a final update on a digest and then completes the digest computation.
digest_update	Used to update a digest using the specified characters.
digest_reset	Used to reset a digest.
digest_delete	Used to delete a digest.

How exactly these additional functions could be defined is listed in Annex A.4.

10.2.2.5 State handling

In general, it is assumed that both Functional Actors and Functional Actor Replicas could be in one of the following states:

- INITIALIZING
- OPERATION
- FAILURE / RECOVERY

Open Point: It is assumed that the API should be extended by functions for Functional Actor (Replicas) to report state changes to the Platform (e.g., a Functional Actor may have finished initialization and now be ready for operation), and for Function Actor (Replicas) to obtain state information about other Functional Actor (Replicas). The details of this are ffs.

10.2.2.6 Recovery

Open Point: Additional functions related to recovery are ffs.

10.3 Extended PI API, as used by non-replicated Functional Actors

For non-replicated Functional Actors, it currently appears that larger parts of the POSIX API (in version POSIX 1003.1 and the evolution thereof) are in scope, as are the additional APIs defined in Section 10.2.2.

Open Point: The exact scope of the Extended PI API is ffs.

It is here important to stress that when Functional Actors utilizing the Extended PI API for instance make use of direct access to the TCP/IP or UDP/IP protocol stacks of the Platform, they in consequence have to take care of security, redundancy, etc., themselves.

11 Application Example

11.1 Introduction

In this section, an abstract application example is provided to illustrate how Functional Actors use Flows for messaging among each other, how related configuration files could look like, and how the API functions described in Section 10 could be utilized. Finally, two deployment options get introduced to show how Functional Actors could be deployed on Computing Elements.

It should be stressed that the provided application example is for illustration purposes only; in particular the considerations on possible configuration files and formats are only to be seen as an option and require further specification work.

11.2 Chosen Abstract Application Setup

An exemplary abstract application setup is shown in Figure 16. Here, the main Functional Actor of interest is depicted in the centre of the figure and named FA-EX. This Functional Actor could, for instance, represent an onboard vehicle locator function that takes input from safe and non-safe sources, and provides output (i.e., location information) to both safe and non-safe recipients. In more detail,

- The Functional Actor FA-EX receives data from a uni-directional Flow_1 (coming from three publishing Functional Actors FA-PUB_A/B/C), as well as request messages from Functional Actor FA-IN via a bi-directional Flow_3;
- The Functional Actor FA-EX uses received data for its internal business logic;
- If Functional Actor FA-EX doesn't receive a request message from FA-IN within 50 ms via the bi-directional (Request/Response) Flow_3, it outputs an error;
- Every 50 ms the Functional Actor FA-EX sends output to a uni-directional Flow_0 and to a dedicated Functional Actor FA-OUT using bi-directional Flow_2;
- The three Functional Actors FA-PUB_A, FA-PUB_B and FA-PUB_C periodically publish data to the uni-directional Flow_1;
- The two Functional Actors FA-SUB_A and FA-SUB_B consume the periodically published values of FA-EX obtained via the uni-directional Flow_0.

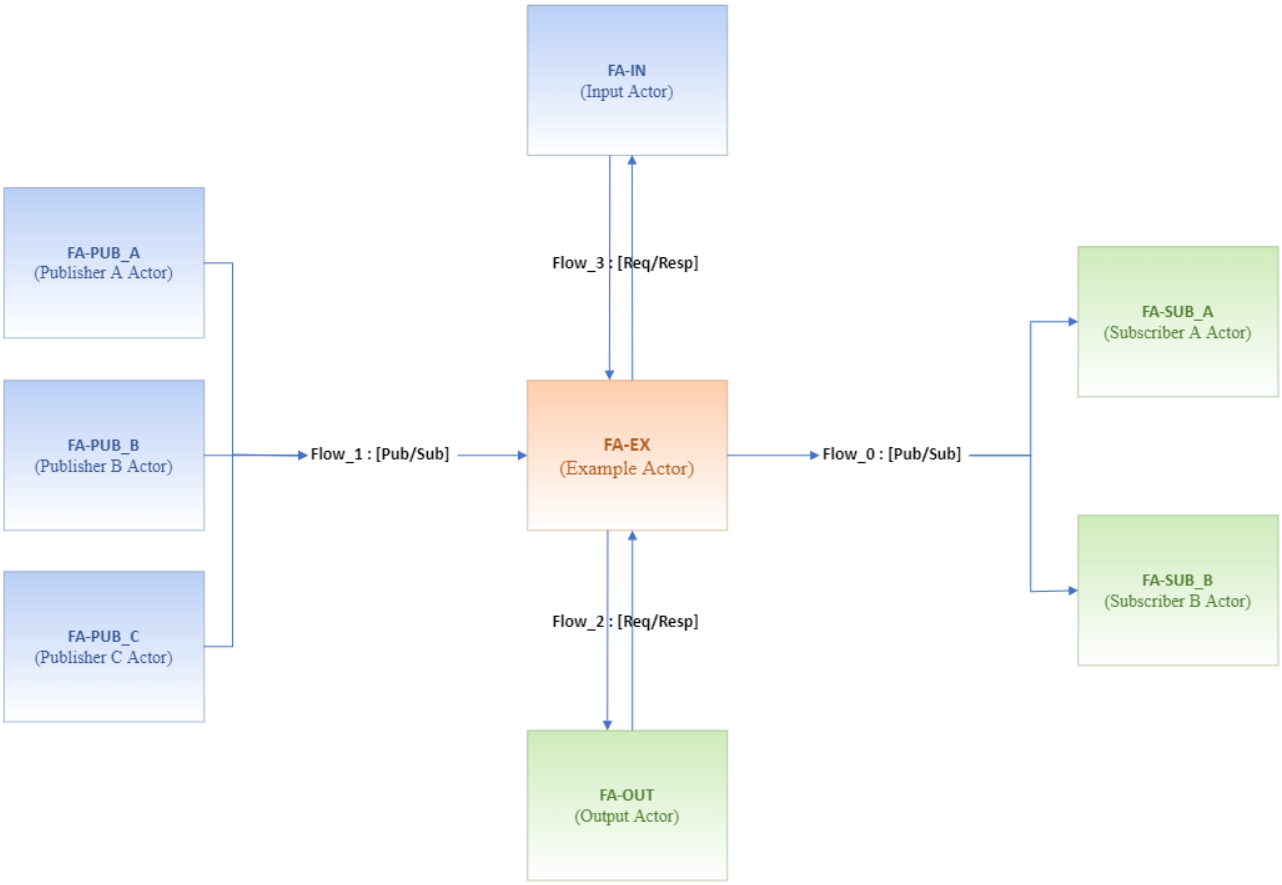


Figure 16. Abstract application example for illustration purposes.

11.3 Possible Configuration Files

For the application example introduced in the previous section, configuration files could look like in the following example (here assuming that a JSON notation is used). Please note that configuration items listed are for illustration purposes only and certainly not conclusive. Properties for memory needs (RAM and non-volatile) and processing and performance quota might be added for instance.

```
{
     "functional actors" :
    [
        {
            "name" : "FA-EX" ,
            "type" : "TypeExample"
        },
        {
            "name" : "FA-IN" ,
            "type" : "TypeInput"
        },
        {
            "name" : "FA-OUT" ,
            "type" : "TypeOutput"
        },
        {
            "name" : "FA-PUB_A" ,
            "type" : "TypePublisherA"
        },
        {
            "name" : "FA-PUB_B" ,
```

```
            "type" : "TypePublisherB"
        },
        {
            "name" : "FA-PUB_C" ,
            "type" : "TypePublisherC"
        },
        {
            "name" : "FA-SUB_A" ,
            "type" : "TypeSubscriberA"
        },
        {
            "name" : "FA-SUB_B" ,
            "type" : "TypeSubscriberB"
        }
    ],
    "flows" :
    [
        {
            "name"  : "Flow_1" ,
            "publishers"  : [ "FA-PUB_A" , "FA-PUB_B" , "FA-PUB_C" ] ,
            "subscribers" : [ "FA-EX" ] ,
            "message_delivery" : "at most once" ,
            "voting" : true
        },
        {
            "name"  : "Flow_0" ,
            "publishers"  : [ "FA-EX" ] ,
            "subscribers" : [ "FA-SUB_A" , "FA-SUB_B" ] ,
            "message_delivery" : "at least once" ,
            "voting" : true
        },
        {
            "name"      : "Flow_2" ,
            "requester" : "FA-EX" ,
            "responder" : "FA-OUT" ,
            "maximum_message_delivery_time_ms” : 50 ,
            "inform_requestor_about_exceeded_delivery_time” : true ,
            "voting" : true
        },
        {
            "name"      : "Flow_3" ,
            "requester" : "FA-EX" ,
            "responder" : "FA-IN" ,
            "inform_responder_about_exceeded_delivery_time” : true ,
            "voting" : true
        }
    ]

}
```

Possible High-level Sequence Diagram for FA-EX

Assuming that Functional Actor FA-EX is implemented with multi-threading support, a possible sequence diagram for its initialization and operation is depicted in Figure 17. Please note that this sequence diagram is for illustration purpose only and does not claim for completeness. For the sake of simplicity, no concrete POSIX function calls are depicted nor specific functional exchanges of the PI API as proposed in Section 10 are used.

During initialization of Functional Actor FA-EX three additional threads get created:

- Two threads for the handling of receiving massages from flows Flow_1 and Flow_3;

- One thread for sending request and receiving responses on the bi-directional Flow_2.

Messages on Uni-directional Flow_0 are supposed to be sent from the main thread context, since they are non-blocking. The above-mentioned timing requirements and error handling in case of a timeout get described on a very high level and needed synchronization primitives are assumed to be in place and out of scope.

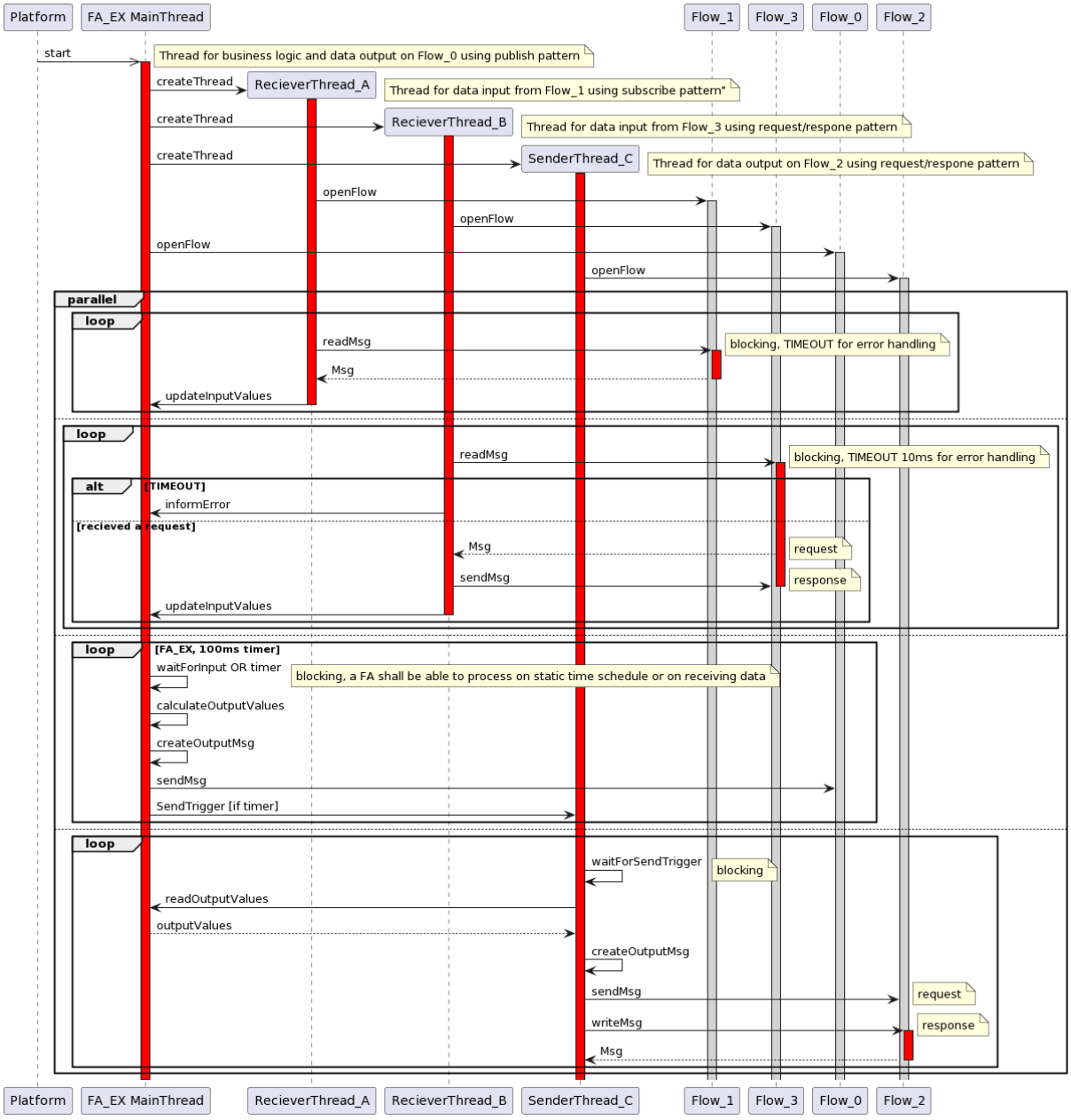


Figure 17. Possible high-level sequence diagram for Functional Actor FA-EX.

11.4 Possible Deployment Scenarios

Irrespective of the implementation of Functional Actor FA-EX as described in the previous section, the following two deployments of the exemplary Functional Actors would be thinkable (among other options).

In both deployment scenarios, the example Functional Actor is deployed on a Platform comprised of three Computing Elements. Furthermore, it is assumed that in case Functional Actors are being bundled, replication (if applicable) is applied jointly to the set of bundled Functional Actors.

For optimisation (e.g., to improve latency), voting on messages exchanged between bundled Functional Actors is purposely skipped. However, voting is applied to messages exchanged with Functional Actors residing outside the bundled Functional Actors (refer also to Section 6.2.3.1).

In the first exemplary deployment scenario depicted in Figure 18, the example Functional Application is deployed without the use of bundling. As a result, voting and distribution is applied on all messages exchanged via Flow_0, Flow_2 and Flow_3.

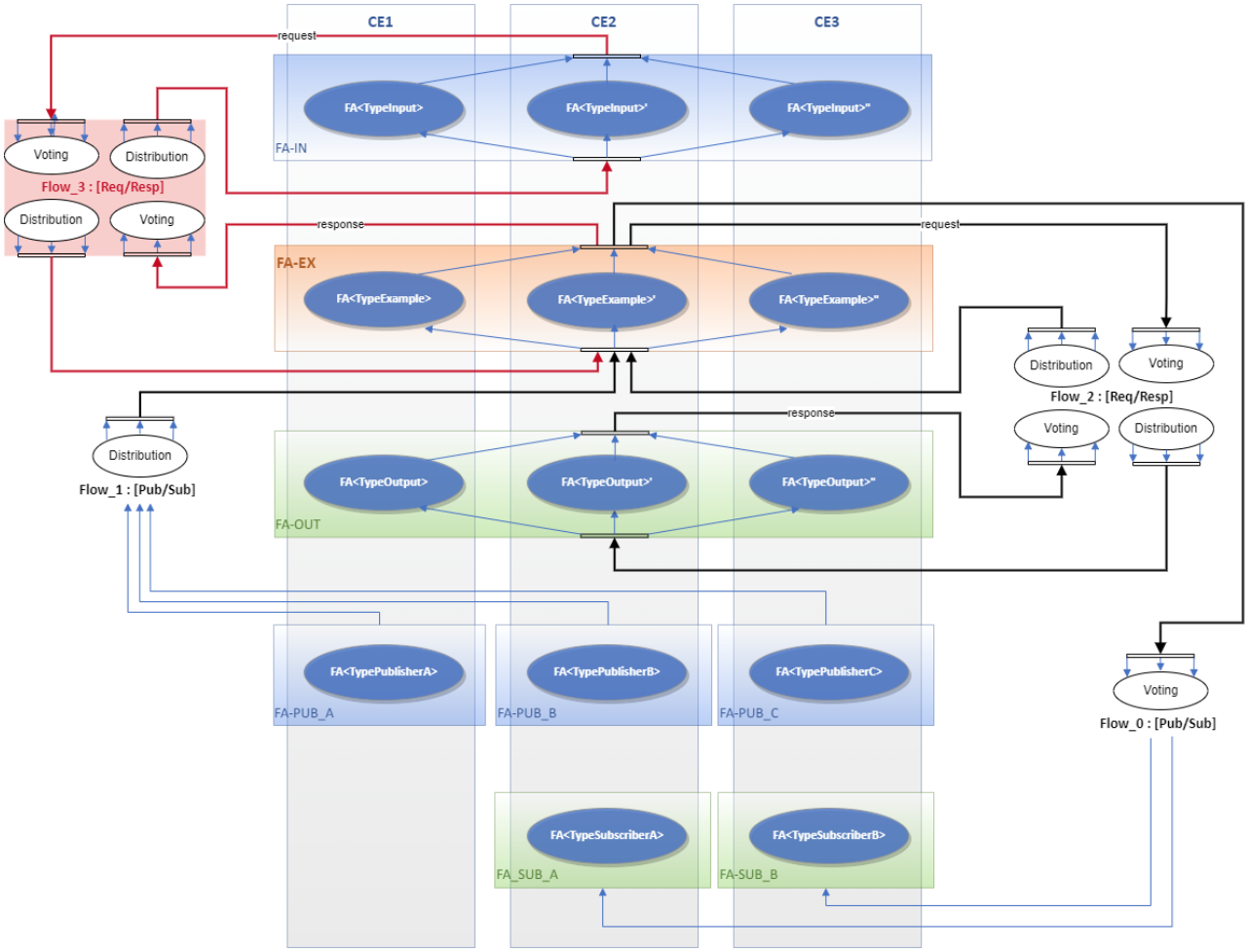


Figure 18. Possible deployment example where Functional Actors are not bundled.

In a second exemplary deployment scenario depicted in Figure 19, the Functional Actors FA-IN and FA-EX are bundled. As a result, the platform may optimise the communication between FA-IN and FA-EX and skip voting and distribution on all messages exchanged via Flow_3.

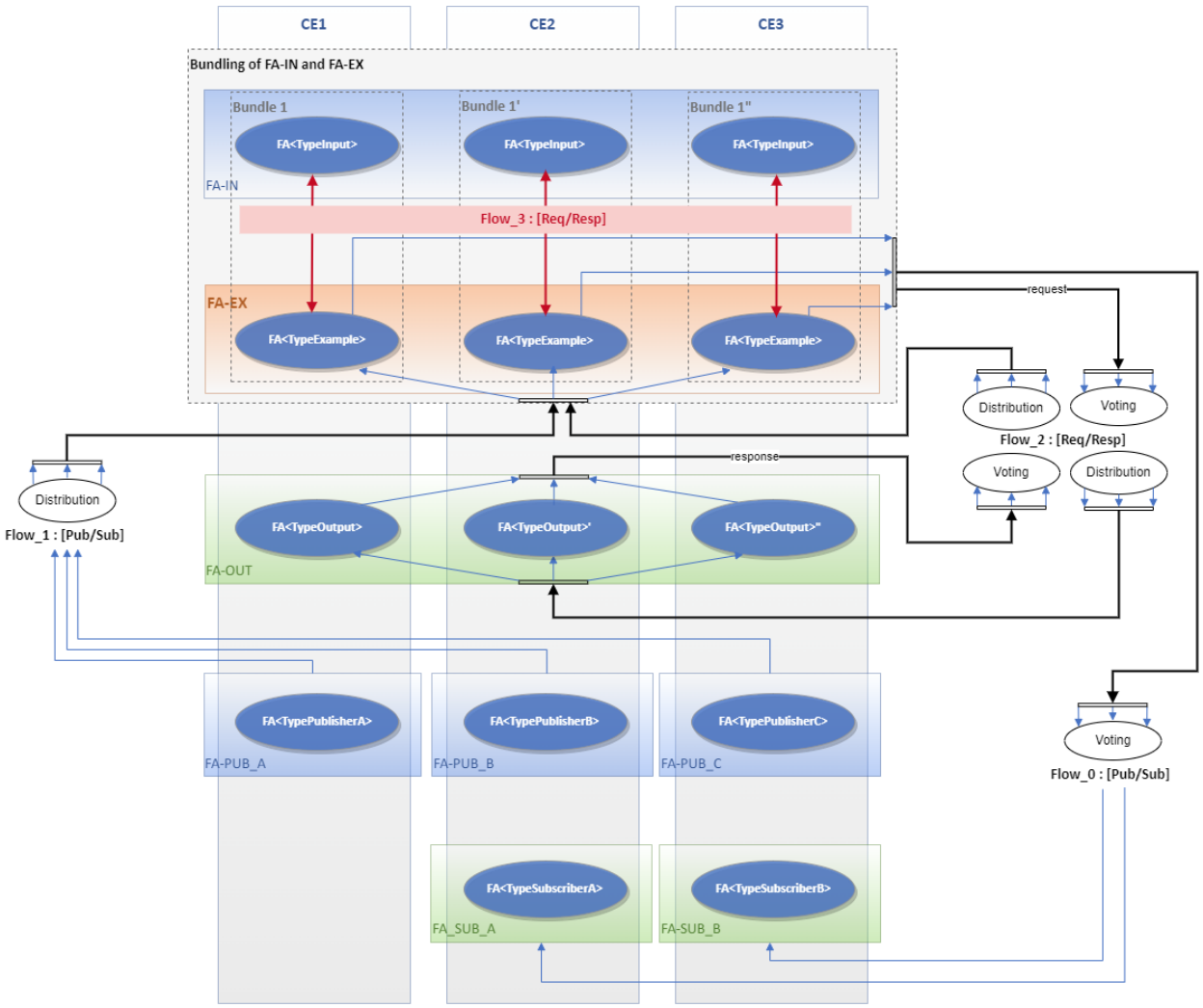


Figure 19. Possible deployment example where Functional Actors FA-IN and FA-EX are bundled.

12 Outlook on Future Work

While the joint specification work among railways and industry suppliers that has been captured in this document has obviously helped to move the vision of a standardized separation of (safety-related and non-safety-related) railway applications from the underlying IT platforms forward, substantial further specification work and of course prototyping is required.

In general, it is assumed that this document provides a good basis for the development of prototypes of trackside and onboard implementations of compute platforms following the notion of a PI API, though additional Platform requirements would need to be defined that are beyond the scope of this API specification. In such prototyping, one could then for instance experiment with using DDS to realize the messaging needs noted in Section 6. Based on the learnings of prototyping, the various open points listed in this document could then be addressed, and a detailed and binding PI API specification could be developed, basically further narrowing down and detailing the considerations from Section 10.

Beyond this, it is expected that also additional specifications should be developed, or further considerations are required, for instance related to

- logging and diagnostics;

- remote updates;
- orchestration;
- standardized tooling and testing;
- modular certification approaches;
- IT Security;
- persistence of application data;
- interfaces "below" to the virtualization (and HW layer);
- juridical recording;
- scalability.

Annex A: Possible Definitions for API Functions required beyond POSIX

This section shows a possible API definition that extends the POSIX functions defined in Section 10.2.1. To represent the API, the Interface Description Language (IDL) is used as a base with some modifications, such as the usage of POSIX data types. It is not intended that the following IDL definitions follow exactly the IDL syntax, but to show the different types and functions that have been identified in Section 10.2.2. Therefore, code generation may not be performed from the following IDL, but it allows manual implementation of the API types and functions.

A.1 Possible Definitions for Functions related to Flows

Flow-related functions could be defined in the Header file *flows.h*. The functionality provided by *flows.h* could be defined as follows:

```
interface Flow {

    // Flags to be used when opening a Flow
    enum e_fl_oflags
    {
        O_REQUESTER  = 1,
        O_RESPONDER  = 2,
        O_PUBLISHER  = 4,
        O_SUBSCRIBER = 8,
        O_NONBLOCK   = 16
    }

    // Flags to be used when sending/receiving messages - only necessary if we
decide that there is a need for a control channel
    enum e_fl_channels
    {
        C_USER = 1,  // user message channel
        C_CTRL = 2   // control message channel
    }

    // flow attributes (static and dynamic)
    struct fl_attr {
        // to be defined
    };

    // function used to join, register or subscribe to a Flow
    fld_t fl_open(in string name,
                  in int oflags);

    // function used to leave, unregister or unsubscribe from a Flow
    int fl_close(in fld_t fldes);

    // function used to send a user or a control message to a flow. Blocking or
non-blocking, depending on the oflags used when opening the Flow
    int fl_send(in fld_t fldes,
                in void * data,
```

```
                    in int msg_len,
                    in e_fl_channels fl_channel);

// forward declaration of the metadata struct that is defined by each platform
    struct platform_metadata_t;

    // function to receive a user or control message from a flow. Blocking or
non-blocking, depending on the oflags used when opening the Flow
    void * fl_receive(in fld_t fldes,
                      inout platform_metadata_t meta_data,
                      in int msg_len,
                      out e_fl_channels fl_channel);

    // function to get flow attributes (static & dynamic attributes)
    int fl_getattr(in fld_t fldes,
                      in fl_attr attr);

    // function to set flow attributes (restricted set of dynamic attributes)
    int fl_setattr(in fld_t fldes,
                      in fl_attr newattr,
                      out fl_attr oldattr);

};
```

A.2 Possible Definitions for additional Timers

As stated in Section 10.2.2.2, the functions related to timers in POSIX should also be sufficient for the SCP, but additional timers should be defined. This could be done in a Header file *sync_time.h* as stated in the following:

```
// Please note this definition is not strictly IDL as it tries to mimic the
// definition of timers in POSIX
interface Sync_Time {
   enum sync_clockid_t { SYNCHRONIZED_REALTIME, SYNCHRONIZED_MONOTONIC };
   int clock_getres(in sync_clockid_t clock_id);
   int clock_gettime(in sync_clockid_t clock_id);
   int clock_nanosleep(in sync_clockid_t clock_id);
   int clock_settime(in sync_clockid_t clock_id);
   int timer_create(in sync_clockid_t clock_id);
   int timer_delete(in timer_t timerid);/
   int timer_getoverrun(in timer_t timerid);
   int timer_gettime(in timer_t timerid);
   int timer_settime(in timer_t timerid);
};
```

A.3 Possible Definitions for Functions related to Configuration Management

Functionality to support passing configuration to Functional Actors could be supported through a Header file *configuration.h*. It could be defined as follows:

```
interface Configuration {
    // A location description that can be passed to the application.
    // Contains a path as well as permission information for paths
    // or files under platform supervision
    struct cfg_path_t {
        string<4096> path;
        boolean directory;
        octet mode;
    }
    // Retrieve all labels for configuration locations from the API
    void get_configuration_labels(out string<255>[] labels);
    // Retrieve default configuration
    cfg_path_t get_default_configuration();
    // Retrieve a labeled configuration path from the API
    cfg_path_t get_configuration_by_label(in string<255> label);
};
```

A.4 Possible Definitions for Functions related to Checksums

Functionality to support checksumming on the SCP could be defined in a Header file *checksum.h*. The functionality provided by *checksum.h* could be defined as follows:

```
interface Checksum {
    enum checksum_t { MD4, MD5, SHA1, SHA256, SHA512 };
    typedef md4_sum_t CharArray[32];
    typedef md5_sum_t CharArray[32];
    typedef sha1_sum_t CharArray[40];
    typedef sha256_sum_t CharArray[64];
    typedef sha512_sum_t CharArray[128];

    // Create a message digest
    boolean messagedigest_create(in checksum_t algorithm_id,
                                 out messagedigest_t digest_id );
    // Completes the operation
    string digest();
    // Performs a final update on the digest using the specified array of bytes,
then completes the digest computation.
    string digest(in messagedigest_t digest_id,
                  in char[] data);
    // Updates the digest using the specified characters.
    void digest_update(in messagedigest_t digest_id,
                       in char[] data);
    // Resets the digest for further use.
    void digest_reset(in messagedigest_t digest_id);
    // Deletes the digest.
    void digest_delete(in messagedigest_t digest_id);
};
```

Annex B: Kubernetes Style YAML Configuration for the SCP

B.1 Introduction

In the following, a possible configuration approach will be presented that orients itself on the Kubernetes API. In no way is this supposed to imply that a usage of Kubernetes is envisioned in the SCP context. It is, so far, not understood if Kubernetes is usable as an orchestrator in a safety critical system at all, however it is an industry standard for deployment and orchestration in other fields and as such the idea was born to see if the SCP concepts could be expressed in a similar style.

In this excerpt it is assumed that basic primitives provided by Kubernetes configuration would be available and can be used in the description of a deployable.

In the following YAML file, which is modelled after the example application depicted in Figure 16, a fictional deployment kind Functional Application is described. Note that in this example, bundles are also defined in the case where they contain only one Functional Actor:

```yaml
---
apiVersion: functionalApplication/v1
kind: FunctionalApplication
metadata:
  name: Example Application
  namespace: example1
  labels:
    app: example-app
spec:
  selector: # Tells the scheduler which labels to manage
    matchLabels:
      app: example-app
    template: # Annotate each bundle with label app: example1
      metadata:
        labels:
          app: example-app
      spec:
        bundles:
        - name: fa-ex
          actors:
          - name: FA-EX
            metadata:
              actorname: fa-ex
            # ... Further attributed need to be specified in order to e.g. define        resource constraints
          replicas: 1
        - name: fa-in
          actors:
          - name: FA-IN
          replicas: 1
        - name: fa-out
          actors:
          - name: FA-OUT
          replicas: 1
        - name: fa-pub_a
          actors:
          - name: FA-PUB_A
          replicas: 1
        - name: fa-pub_b
          actors:
```

```
      - name: FA-PUB_B
        replicas: 1
      - name: fa-pub_c
        actors:
        - name: FA-PUB_C
        replicas: 1
      - name: fa-sub_a
        actors:
        - name: FA-SUB_A
        replicas: 1
      - name: fa-sub_b
        actors:
        - name: FA-SUB_B
        replicas: 1
```

Likewise, the involved Flows could be described using the functional kind Flow:

```
---
apiVersion: functionalApplication/v1
kind: Flow
metadata:
  name: Example Application Flow 0
  labels:
    app: example-app
spec:
  type: unidirectional
  sourceSelector:
    app: example-app
    bundle: fa-ex
  targetSelector:
  - labelSelector:
    - key: app
      operator: In
      values:
      - example-app
    - key: bundle
      operator: In
      values:
      - fa-sub_a
      - fa-sub_b

---
apiVersion: functionalApplication/v1
kind: Flow
metadata:
  name: Example Application Flow 1
  labels:
    app: example-app
spec:
  type: unidirectional
  sourceSelector:
  - labelSelector:
```

```
    - key: app
      operator: In
      values:
      - example-app
    - key: bundle
      operator: In
      values:
      - fa-pub_a
      - fa-pub_b
      - fa-pub_c
  targetSelector:
    app: example-app
    bundle: fa-ex

---
apiVersion: functionalApplication/v1
kind: Flow
metadata:
  name: Example Application Flow 2
  labels:
    app: example-app
spec:
  type: bidirectional
  sourceSelector:
  - labelSelector:
    app: example-app
    bundle: fa-ex
  targetSelector:
    app: example-app
    bundle: fa-out

---
apiVersion: functionalApplication/v1
kind: Flow
metadata:
  name: Example Application Flow 3
  labels:
    app: example-app
spec:
  type: bidirectional
  sourceSelector:
  - labelSelector:
    app: example-app
    bundle: fa-ex
  targetSelector:
    app: example-app
    bundle: fa-in
```

For the sake of following the example and for readability, several details were omitted in this example that shall be explained further in the following sections.

B.2 Constraints regarding mapping of Replicas to Computing Elements

As Figure 18 and Figure 19 indicate, there is a need to express if different Functional Actor Replicas should or should not be deployed on the same Computing Element. This requirement can easily be fulfilled with existing Kubernetes primitives. One such way is using affinities [8], which can be expressed with either an affinity or an anti-affinity. See an example for the bundling from Figure 19:

```
- name: fa_sub-a
 metadata:
   bundle: fa_sub-b
 actors:
 - name: FA_SUB-A
 replicas: 1
 affinity:
   replicaAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - labelSelector:
       bundle: fa_sub-b
       topologyKey: platform/hostname
```

B.3 Referencing Individual Functional Actors in Flows

The Kubernetes API defines a rather smart filtering functionality to find other components [9]. In the flow examples seen, so far, this was used to reference the bundle a communication is to be established to. However, if the actors' metadata is extended with appropriate metadata, as is the actor in the bundle fa-ex in the above example, direct referencing of the actor becomes possible.

References

[1] RCA initiative, see https://www.eulynx.eu/index.php/news

[2] OCORA, see https://github.com/OCORA-Public/Publication

[3] Europe's Rail programm, see https://rail-research.europa.eu/about-europes-rail/

[4] RCA/OCORA, "An Approach for a Generic Safe Computing Platform for Railway Applications", White paper, OCORA-TWS03-010, Version 1.1, July 2021, see https://github.com/OCORA-Public/Publication/blob/master/04_OCORA%20Delta%20Release/OCORA-TWS03-010_Computing-Platform-Whitepaper.pdf

[5] Europe's Rail Multi-Annual Work Plan, see https://rail-research.europa.eu/wp-content/uploads/2022/03/EURAIL_MAWP_final.pdf

[6] Angel Martinez Bernal, Mark Carrier and Mark Hary, "OMG DDS Reference Implementation for Safe Computing Platform Messaging", Version 1.0, July 2022, see https://github.com/OCORA-Public/Publication/blob/master/91_SCP_OMG_DDS_Reference_Implementation/SCP_OMG_DDS_Reference_Implementation.pdf

[7] POSIX Headers, see https://pubs.opengroup.org/onlinepubs/9699919799/idx/head.html

[8] Affinities, see https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[9] Labels and Selectors, see https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/