RCA
Reference CCS Architecture

OCORA

Generic Safe Computing Platform

Draft Initial Specification of the PI API between Application and Platform

A joint outcome from the Initiatives RCA and OCORA

Version 1.0, December 2021

Table of Contents

1. Aim and Scope

This document aims to provide a very first draft of the specification of the envisioned Platform Independent Application Programming Interface (PI API) between applications and underlying platform in the Safe Computing Platform (SCP) concept described in [1]. This draft specification shall serve as an input to the envisioned joint specification work among the railways in RCA/OCORA and selected industry partners, which is then expected to lead to a version 1.0 of the PI API specification around Spring 2022.

It is important to note that:

- The PI API as specified in this document refers to both trackside datacenter 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;
- All aspects in this very first specification draft are of course subject to discussion and review with industry partners.

Clearly, the specification of the PI API is only one of the specifications needed to realize the vision of the Safe Computing Platform as stated in [1] – it is expected that this will later be complemented with other specifications, for instance related to a standardized logging, diagnostics, life-cycle management, safety-related application constraints (SRACs), etc., for different platform realizations.

This document is structured as follows:

- In Section 2, the PI API and its motivation are shortly recapped from the White Paper;
- In Section 3, key terms used in this specification are introduced;
- In Section 4, key entities referred to in the specification are defined;
- In Section 5, guiding principles and key premises regarding the PI API design are listed;
- In Section 6, the state machines of the defined entities are introduced;
- In Section 7, certain data types are defined;
- In Section 8, procedures are defined;
- In Section 9, API exchanges are introduced;
- In Section 10, additional platform capabilities are described which are expected to be inherently enabled through the previously defined API exchanges (and hence do not need explicit standardization), and finally,
- In Section 11, open points for further discussion are identified.

2. PI API – Motivation and Introduction

PI API refers to the general abstracted interface that allows functional applications to run unchanged on different computing platform implementations, as detailed in [1]. PI API is one of the key platform capabilities that help a buyer to achieve many high-level objectives such as ‘Respect diverse lifecycles of business logic and runtime environment (RTE)’, ‘Open market to new players’, ‘Migratable and portable business logic’, ‘System evolvability’ and ‘Vendor independence’, as also detailed in [1]. It should be stressed that it is distinguished between [1]

- PI API (used by all applications running on the platform);
- Extended PI API (used only by gateway functions running on the platform, including access to lower layer communication protocol stacks such as TCP/IP or UDP/IP).

In this initial draft of the specification, the focus is only on the PI API (not the extended PI API).

3. Abbreviations

Table 1. Abbreviations.

Abbreviation / Term	Description
CCS	Control Command and Signaling
FFS	For future study
IPC	Inter-Process Communication
MooN	M out of N system concept
OCORA	Open CCS Onboard Reference Architecture
PI API	Platform-independent Application Programming Interfaces
RCA	Reference CCS Architecture
RTE	Runtime environment (see definition in [1])
SCP	Safe Computing Platform
SIL	Safety Integrity Level
SRACs	Safety-related application constraints
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP/IP	User Datagram Protocol / Internet Protocol
UML	Unified Modelling Language

4. Definition of Entities

In the remainder of this specification document, the following definition of entities is used:

- **Platform**: An entire (physical) instance of a Safe Computing Platform as defined in [1], comprising multiple Compute Nodes and RTE Instances running on these. For the purpose of specifying the PI API, it is for now assumed that all RTE Instances are provided by the same vendor. Note that a Platform may be geographically distributed (though the distribution is transparent to application and hence not relevant to this specification);

- **Compute Node**: A single (possibly multi-core) compute element used by a Platform. It is assumed that different Functional Actors may concurrently run on the same Compute Node in separate Partitions. However, it is also assumed that different Functional Actor Replica of the same Functional Actor need to run on different Compute Nodes;

- **Partition:** A dedicated execution environment with an isolated memory address space and limited execution time. It is expected that multiple Partitions can be created on a single Compute Node, allowing that different Functional Actor Replicas related to different Functional Actors can concurrently run on the same Compute Node (but not multiple Functional Actor Replicas of the same Functional Actor);

- **RTE Instance**: An instance of a runtime environment as defined in [1], which runs on a single Compute Node;

- **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 [3]. Application-level communication among Functional Applications is expected to follow standardized interfaces, for instance defined by RCA [4] or OCORA [5]. Functions within one Functional Application may have different functional safety requirements;

- **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 in order to guarantee that the output of the Functional Actor complies with the MooN configuration required to meet the functional safety requirement. As such, a Functional Actor can be viewed as an artifact consisting of a piece of application logic and that application logic's configuration, as well as a set of replication rules the Platform must apply to it and its voter in order to comply with its MooN configuration. 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** (for brevity often only referred to as "Replica"): An instance of a Functional Actor that is run on a single Compute Node, in parallel to other Replicas of the same Functional Actor running on physically separated Compute Nodes. It can be restored to a specific state between the computation of two incoming messages. The Platform applies voting to the outputs of all Replicas of a single Functional Actor to ensure that the output of the Functional Actor complies with the required MooN configuration. Toward a single Replica, it is not visible that other Replicas of the same Functional Actor are running.

The relationship of the introduced entities is also illustrated in Figure 1.

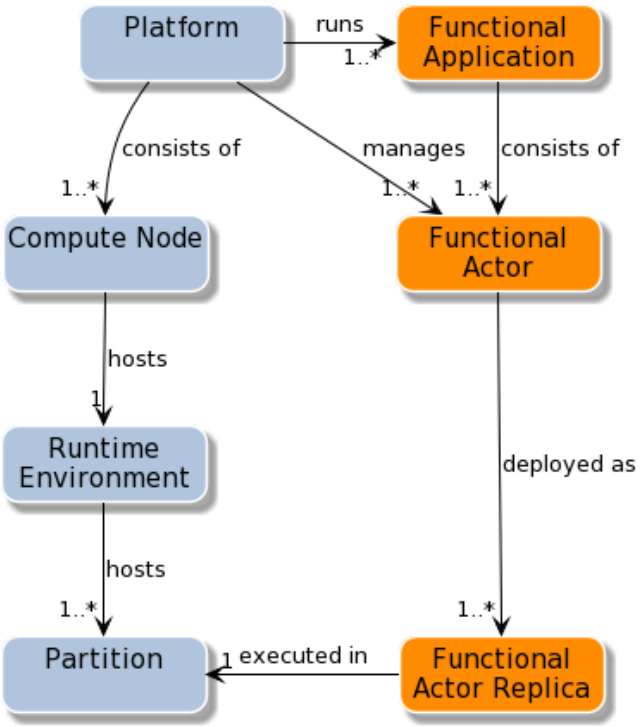

Figure 1. Relationship of defined entities.

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

5. Guiding Principles and Key Premises for the PI API Design

In addition to the requirements captured in [2], the following **guiding principles** shall be followed for the design of the PI API:

- **Restricted number of functions** As the application contains only the business logic, and needed safety and fault-tolerance mechanisms are expected to be encapsulated transparently in the Platform, the number of functions should be as few as possible, e.g., the exchanges for safety should be masked within the platform;

- **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 different 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 2. Note: In the work that has led to this first draft 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;

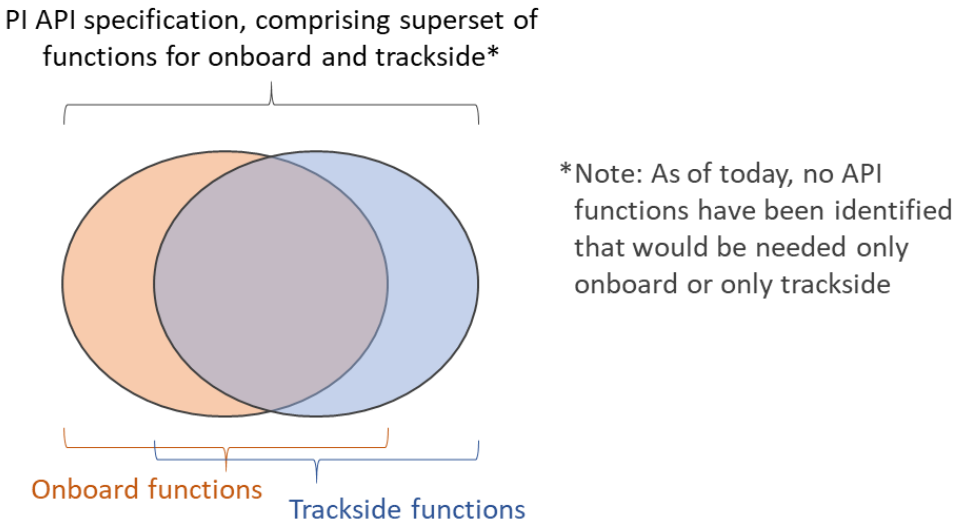

Figure 2. Scope of PI API specification regarding onboard and trackside.

- **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 should be usable for 10-20 years), but future evolved versions are expected to be backward-compatible to at least one or two versions, as shown in Figure 3. Possibly, semantic versioning could be an option;

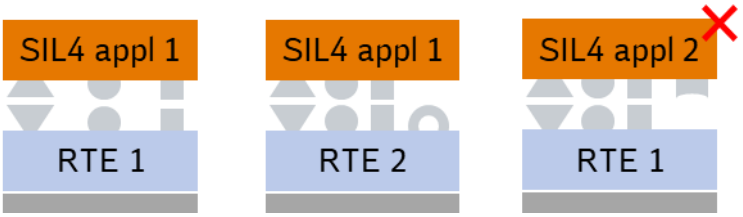

Figure 3. Illustration of notion of backward compatibility.

- **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).

Further, the PI API design captured in this document is based on the following **key premises**:

- **Each Functional Actor Replica runs in a dedicated execution environment (Partition)** with an isolated memory address space and limited execution time. It is assumed that one Partition can host only one Replica related to one Functional Actor since safety-critical applications need computing guarantees;

- **Functional Actor Replicas of the same Functional Actor run in Partitions hosted on different Compute Nodes;**

- **State-machine replication**: The SCP uses state-machine replication for implementing fault tolerance;

- **Publish/subscribe mechanism**: Every topic has a single publisher and may have 0-n subscribers. Having only one publisher guarantees the deterministic sequence of messages within a topic;

- **Voting**: Messages of different Functional Actor Replicas are voted on by the Platform. It is assumed that this voting takes place before messages are distributed via the publish/subscribe mechanism (one benefit of this being that discrepancies between the voted Functional Actor outputs are identified at an early stage);

- **Data message queue**: All data messages of all subscribed topics will be available to the Functional Actor Replica in one single queue. This guarantees the deterministic sequence of messages for all Functional Actor Replicas;

- **Control message queue**: Control messages from the RTE are available to the application in a dedicated queue. All messages in the control queue have to be handled before handling data messages. Having a dedicated control queue allows to easily prioritize those messages;

- **Guaranteed message integrity:** Integrity of messages exchanged between the Platform and Replicas (and the order of messages) is guaranteed by the Platform, for instance through shared memory or inter-process communication (IPC) message passing.

It is to be noted that the above assumptions could be different (including additional assumptions) for specific technical realizations of Safe Computing Platforms and should be considered non-blocking as far as the platform independence objective is achieved.

6. State Definitions

In this section, the state machines of the defined entities are defined, including the transitions from one state to another.

6.1. Functional Actor Replica States

It is assumed that Functional Actor Replica (from the perspective of the Platform) are at any time in one of the following states, as also illustrated in Figure 4:

- **INITIALIZING:** The Replica is in the process of being setup, for instance obtaining its configuration settings and values and subscribing to relevant topics. It does not yet receive or send any control messages or messages from/to other Functional Actors;

- **IN SERVICE:** The Replica is fully functional and correctly executing its business logic. It is able to receive and send control messages as well as messages from/to other Functional Actors;

- **UNRECOVERABLE:** The Replica is either not running at all (e.g., has crashed), or is producing output that is not consistent with that of the other Replica of the same Functional Actor. In consequence, it needs to be restarted. Any possible output of the Replica in this state is not processed any more by the Platform.

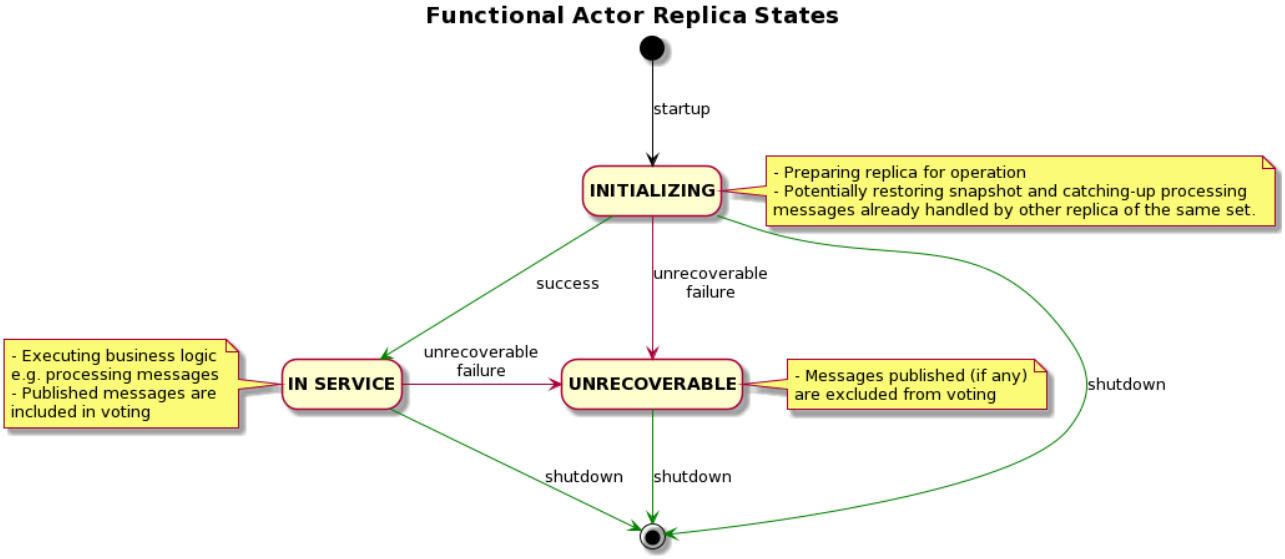

Figure 4. State diagram for Functional Actor Replicas.

The UML code for the Functional Actor Replica state diagram can also be found under: LINK.

6.2. Functional Actor States

It is assumed that Functional Actors as a whole (consisting of multiple Functional Actor Replica as described in Section 4) are at all times in one of the following states (see also Figure 5):

- **INITIALIZING:** The Functional Actor is in the process of being initialized (e.g., a sufficient number of Functional Actor Replica are currently being initiated). The Platform provides created Functional Actor Replica with incoming messages to move them to a predefined state, but does not process the outgoing messages of the Functional Actor Replicas;

- **IN SERVICE:** The Functional Actor is running, in the way that it processes incoming messages from other Functional Actors and generates output messages to other Functional Actors. We here further differentiate between:

    - **NORMAL OPERATION:** All N Functional Actor Replicas (in a MooN redundancy scheme) are in state IN SERVICE;

    - **RECOVERING:** Less than N, but more or equal to M Functional Actor Replicas (in a MooN redundancy scheme) are in state IN SERVICE. The remaining Functional Actor Replicas are in the process of being recovered.

- **UNRECOVERABLE:** The Functional Actor is subject to a major failure (e.g., too many Functional Actor Replicas have failed, and it has not been possible to recover those failed Replicas). In this state, the Functional Actor is not able to operate normally anymore.

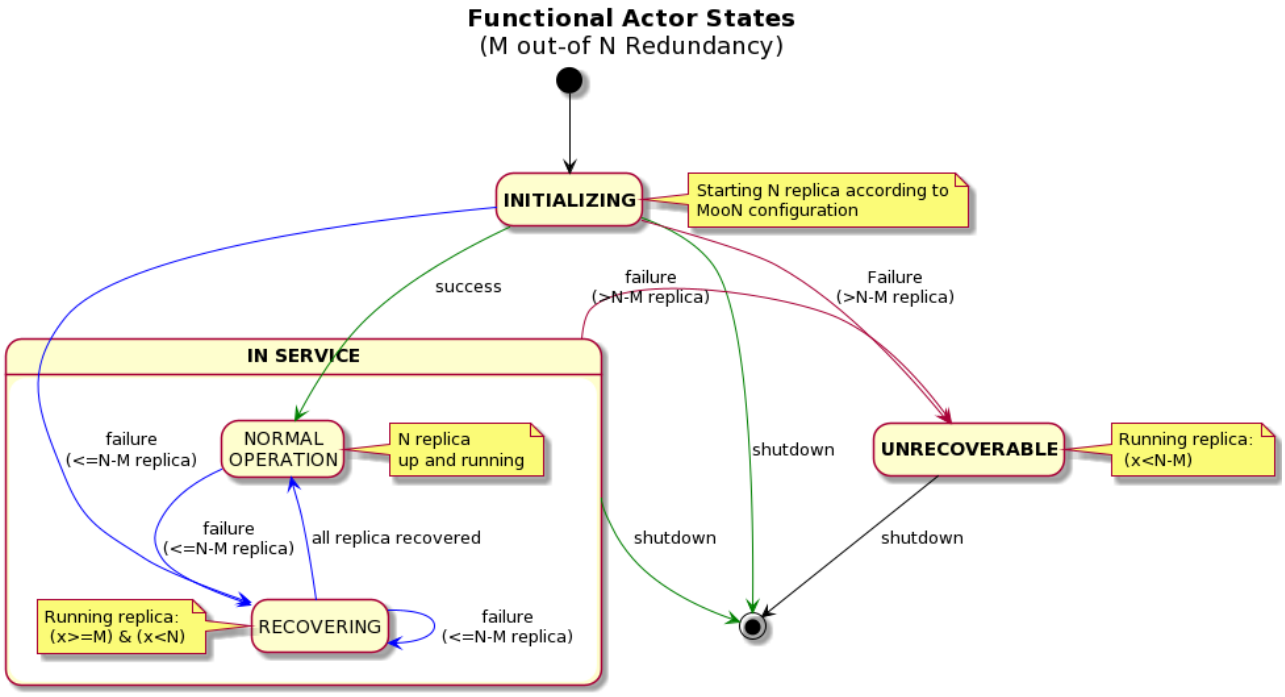

Figure 5. State Diagram for Functional Actors.

The UML code for the Functional Actor state diagram can also be found under: LINK.

6.3. Functional Application States

Functional Applications (potentially consisting of multiple Functional Actors, as described in Section 4) shall at all times be in one of the following states (see also Figure 6):

- **INITIALIZING:** The Functional Application is in the process of being initialized (i.e., the Functional Actors therein are in the process of being initialized). In this process, some Functional Actors may already be in state IN SERVICE, while others are still in state INITIALIZING

- **IN SERVICE:** The Functional Application is in service, in the sense that overall integrity is still guaranteed, any performance degradations (e.g., in terms of latency or reduction of functionality) are still acceptable, and no manual intervention is required. We here further differentiate between:

    - **NORMAL OPERATION:** All Functional Actors of the Functional Application are in state IN SERVICE;

    - **RECOVERING:** Some Functional Actors are not in state IN SERVICE, but are generally recoverable and in the process of being recovered, with delays that are still considered acceptable.

- **UNRECOVERABLE:** The Functional Application is in a state where failed individual Functional Actors cannot be recovered. In this case, the overall Functional Application has to be restarted by the Platform (with or without manual intervention).

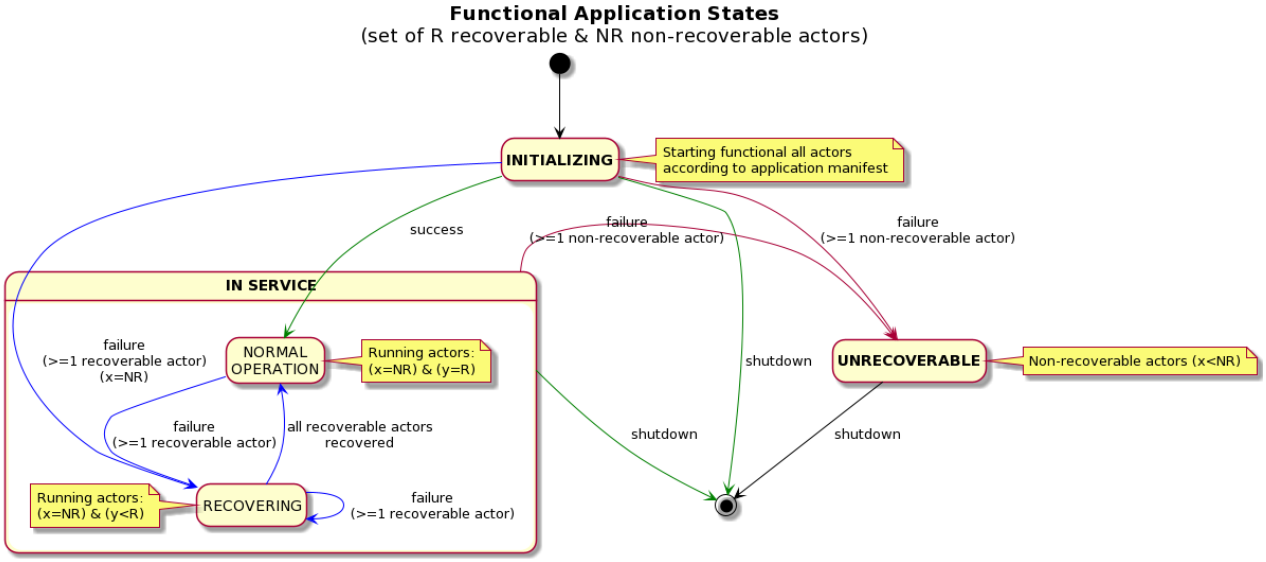

Figure 6. State diagram for Functional Applications.

The UML code for the Functional Application state diagram can also be found under: LINK.

6.4. Platform States

In the following, the state machine for the Platform is defined. Please note that here the term "Platform" here explicitly encompasses a setup of multiple RTE Instances running on multiple Compute Nodes, as introduced in Section 4. The exact interaction among multiple RTE Instances running on multiple Compute Nodes is not described here, as this is expected to be left to the discretion of the Platform vendor.

The Platform shall at all times be in one of the following states (see also Figure 7):

- **STARTUP:** Once the Platform is powered-on, it goes through startup phase which may include hardware initialization, POST, hypervisor startup, etc.;

- **NORMAL OPERATION:** The Platform is running normally in the sense that any errors that may occur (e.g., failure of Compute Nodes) are recovered by the Platform, without impairing the safety of the Functional Applications running on the Platform. This state may also include any maintenance while the Platform is running (e.g., upgrades and patches of Applications, RTE Instances, etc.), as long as the safety of the running Functional Applications is not affected. In this state, additional Functional Applications may be started, or Functional Applications may be terminated (e.g., triggered by a human operator);

- **UNRECOVERABLE:** The Platform encounters some failure which is not recoverable by the Platform itself and/or it cannot guarantee the integrity of the Platform anymore. A simple example of such failure could be a power source fault. The exact definition of the state is FFS.

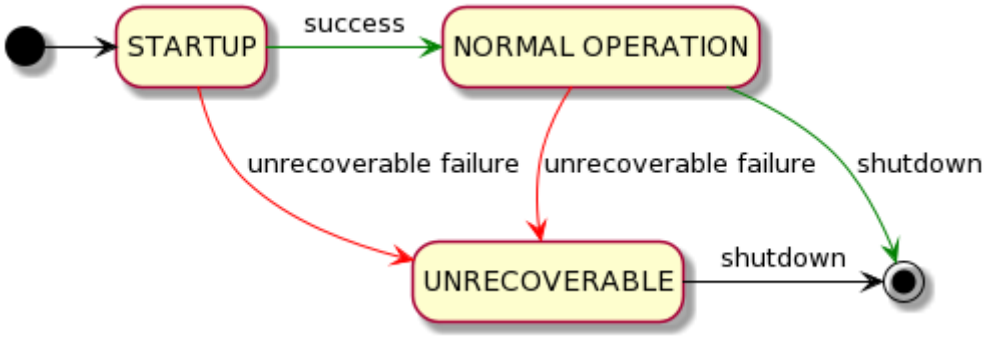

Figure 7. State diagram for the Platform (expected to encompass multiple RTE Instances on multiple Compute Nodes).

The UML code for Platform state diagram in Figure 7 can also be found under: LINK

7. Data Definitions

In the remainder of this specification, the types of data listed in Table 2 are assumed.

Table 2. Defined types of data.

Abbreviation	Description	Used by	Expected to be standardized
Init_cfg_data	**Platform initialization related configuration data:** Configuration data needed by the Platform to know how to initialize applications, for instance including information on • Functional Actors (incl. gateway functions) in the form of Functional Actor Replica to	Platform	Yes

	be created incl. their required MooN configuration; • Info, whether Functional Actor (Replicas) shall be able to access the extended PI API or only the PI API. Note: It is FFS whether this could be also covered in the SRAC's; • Info, whether and which safe communication protocol is to be applied to certain Functional Actor (Replicas) if these communicate to other actors outside of the Platform (unless such protocol is applied through a Gateway Function running on the Platform, see [1]); • Order in which Functional Actor (Replicas) are to be created (if there are any dependencies on the order); • Which application specific configuration data is to be passed to which Functional Actor (Replica).		
Interop_cfg_data	**Interoperability-related configuration data:** Configuration data used by both Platform and Functional Actor Replica which determines, e.g. • Which topic names are to be used by Functional Actor Replica.	Platform and Functional Actor Replica	Yes
Appl_cfg_data	**Application-specific configuration data**: Configuration data that the application needs for initialization. This data is assumed to be agnostic to the Platform, i.e. the Platform is not able to interpret this data. It is left to the discretion of the application vendor how to structure this data (e.g., whether the configuration data is specific to a single Functional Actor, or whether all Functional Actors belonging to a Functional Application receive the same.	Functional Actor Replicas	No
Appl_log_data	**Application-specific log data:** Any log data (including metrics) that may be generated by a Functional Actors (in the form of Functional Actor Replicas and using a function offered by the PI API, see Section 9). This type of log data is stored by the Platform (and in this context complemented with metadata such as timestamps and information on the source of the log data), but the content itself is transparent to the platform.	Functional Actor Replica	No

Platform_log_data	**Platform-specific log data:** Log data (including metrics) created by the Platform itself, which may comprise, e.g.: • Info on when Functional Applications and Functional Actors have been started or terminated; • Info on any issues that have been identified (e.g., different Functional Actor Replica have created inconsistent message output); • Juridical recording of the message exchange among Platform and Functional Actor Replica.	Platform	Yes

8. Procedures

In this section, key procedures involved in the defined states and state descriptions are described.

8.1. Initialization of a Functional Actor Replica

In this section, the procedure is described that would likely be involved in the initialization of a Functional Actor Replica, both in the case that a Replica is to be started from its initial state, and in the case where a Replica is to be started from a specific state (e.g., because it is created in replacement of a previous Replica that has crashed).

For illustration purposes, and to ensure that the procedure could work in at least one specific implementation of a Safe Computing Platform, the following description also details the interaction among a possible RTE Instance on a "Master Node" and the RTE Instance on "Node N" on which the Functional Actor Replica is to be created.

It should be noted, however, that this concrete interaction is of course left to proprietary Platform implementation; the only aspect that ultimately has to be standardized is the interaction of the Platform as a whole (i.e., all entities in Figure 8 colored in blue) and the Functional Actor Replica (colored in orange), i.e., all API exchanges depicted in bold in the figure.

Table 1. Procedure: Initialization of a Functional Actor Replica

Pre-requisites or Pre-conditions	Functional Actor Replica in **Initial State**
Trigger	Triggered in context of procedure **Initialization of a Functional Actor**
Steps	The following steps are visualized in Figure 8. **Note: The fact that the API exchanges described in the following, and depicted in Figure 8, are denoted in "function call" notation does not mean that these necessarily have to be realized as function calls. It is here assumed that it is most important to first agree on the required API exchanges as such, and then define how these would be realized (also including how parameters are handled, etc.).**

	• The Platform triggers the creation of a Functional Actor Replica on a specific Compute Node by using API exchange **createReplica()**. The Replica is provided with its incoming and outgoing message queues to be used for communication;
	• An API exchange **readConfig()** is then used such that the Replica can obtain interoperability-related and application-specific configuration data, as defined in Section 7. Alternatively, this information may of course already be passed to the Replica in the context of API exchange createProcess();
	• The Replica may then register topics (i.e., topics on which it plans to publish messages to other Functional Actors) via API exchange **register()** and subscribe to topics (i.e., on which it expects to receive messages from other Functional Actors) via API exchange **subscribe()**, as needed;
	• The Replica may determine via **getSnapshot()** which exact state it is to be set to (e.g., in case it has been restarted by the Platform and is expected to start from a previously captured state), and assumes this state, if needed;
	• The Replica then publishes a control message MSG_CREATED via **publish()** to the Platform to indicate that it has correctly initialized and is ready to process messages;
	• The following steps then occur in parallel:
	  ○ Utilizing the API exchange **readMsg()**, the Replica consumes any control or application messages available in its incoming message queue. It is assumed that the Replica is inactive until a message is available in its incoming message queue. Upon receipt of any messages, it processes these and publishes messages as needed according to its application logic;
	  ○ The Platform, after receipt of MSG_CREATED from the Replica, provides the Replica with any application messages from the past that it needs to catch up to the state of the other Replica (e.g., in case the Replica had to be restarted). If the Replica could not be successfully initialized, initiates a Replica shutdown via API exchange **shutdown()**;
	  ○ The Platform continuously provides the Replica with new application messages (via the replica's incoming message queue) as these arrive;
	  ○ The Platform also continuously processes application messages sent by the Replica. While doing so, it checks whether the newly created Replica has already caught up with the other Replicas of the same Functional Actor, e.g., by comparing message sequence numbers. If the replica is still catching up, its output is not further processed. Once the newly created Replica has caught up, it is set to state IN SERVICE, meaning its message output is from now on included in voting.

Post-Scenario	Functional Actor Replica in state **IN SERVICE**

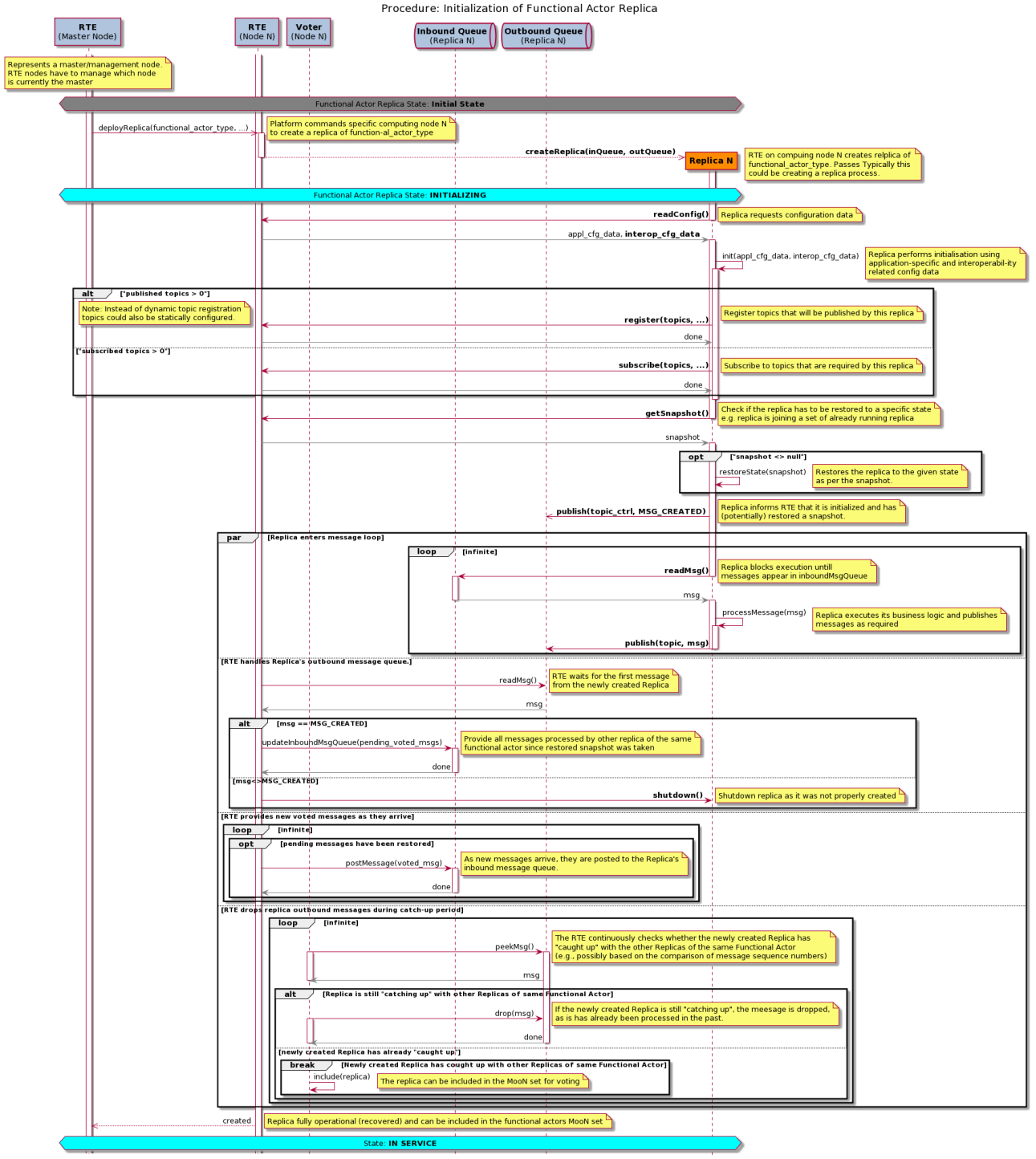

Figure 8. Procedure: Initialization of a Functional Actor Replica (click on figure to open digitally).

The UML code of the sequence diagram shown in Figure 8 can also be found under: LINK

8.2. Functional Actor Replica in Service

In this section, the procedure is described that would likely be involved when a Functional Actor Replica is in state IN SERVICE, i.e., when it is processing and publishing messages according to its application logic.

As in Section 8.1, the following description also details the possible interaction among entities within the Platform (e.g., among different RTE Instances running on different Compute Notes). Further, the description is based on the assumption that message queues and voters are replicated on the different Compute Nodes. Both aspects are of course design decisions that would be left to the discretion of the Platform vendor, and which would not be standardized. What has to be ultimately standardized, are again only the API exchanges among the Platform as a whole (i.e., all blue entities in Figure 9) and the Functional Actor Replicas (orange entities) – in the figure marked in bold.

Table 2. Procedure: Functional Actor Replica in Service

Pre-requisites or Pre-conditions	Functional Actor Replica in state **IN SERVICE**
Trigger	Continuously happening when a Functional Actor Replica is in state IN SERVICE
Steps	The following steps are also illustrated in Figure 9. **Note: The fact that the API exchanges described in the following, and depicted in Figure 8, are denoted in "function call" notation does not mean that these necessarily have to be realized as function calls. It is here assumed that it is most important to first agree on the required API exchanges as such, and then define how these would be realized (also including how parameters are handled, etc.).** The following steps are expected to happen in parallel: • The Functional Actor Replica continuously processes incoming messages using API exchange readMsg(), and publishes messages using publish() according to its application logic; • The Platform constantly replicates the outbound messages of all Functional Actor Replicas belonging to a Functional Actor to the outbound message queues of the other Functional Actor Replicas (assuming that there is a redundant Voter instance on each Compute Node that requires that the messages are replicated prior to voting). It is noted that there may of course be other means to realize this, left to the discretion of the Platform vendor; • The Platform also continuously ensures that published messages of the Functional Actor Replica are voted on. In case the voting result is such that M or more Replica (in a MooN configuration) have published identical output, these messages are distributed to the Functional Actor Replicas of the Functional Actors that have subscribed to the related topics. In case up to N-M Replica have published inconsistent messages, the Platform triggers a shutdown of these Functional Actor Replica(s) via API exchange shutdown() and deploys/creates new Functional Actor Replica(s) as needed (via the process described in Section 8.1). If more than N-M Replica have published inconsistent messages, the Functional Actor is set to state UNRECOVERABLE.

	• The Platform also continuously distributes any incoming (voted) messages from other Functional Actors to the Functional Actor Replicas based on their topic subscriptions.
Post-Scenario	Functional Actor Replica in state **IN SERVICE**

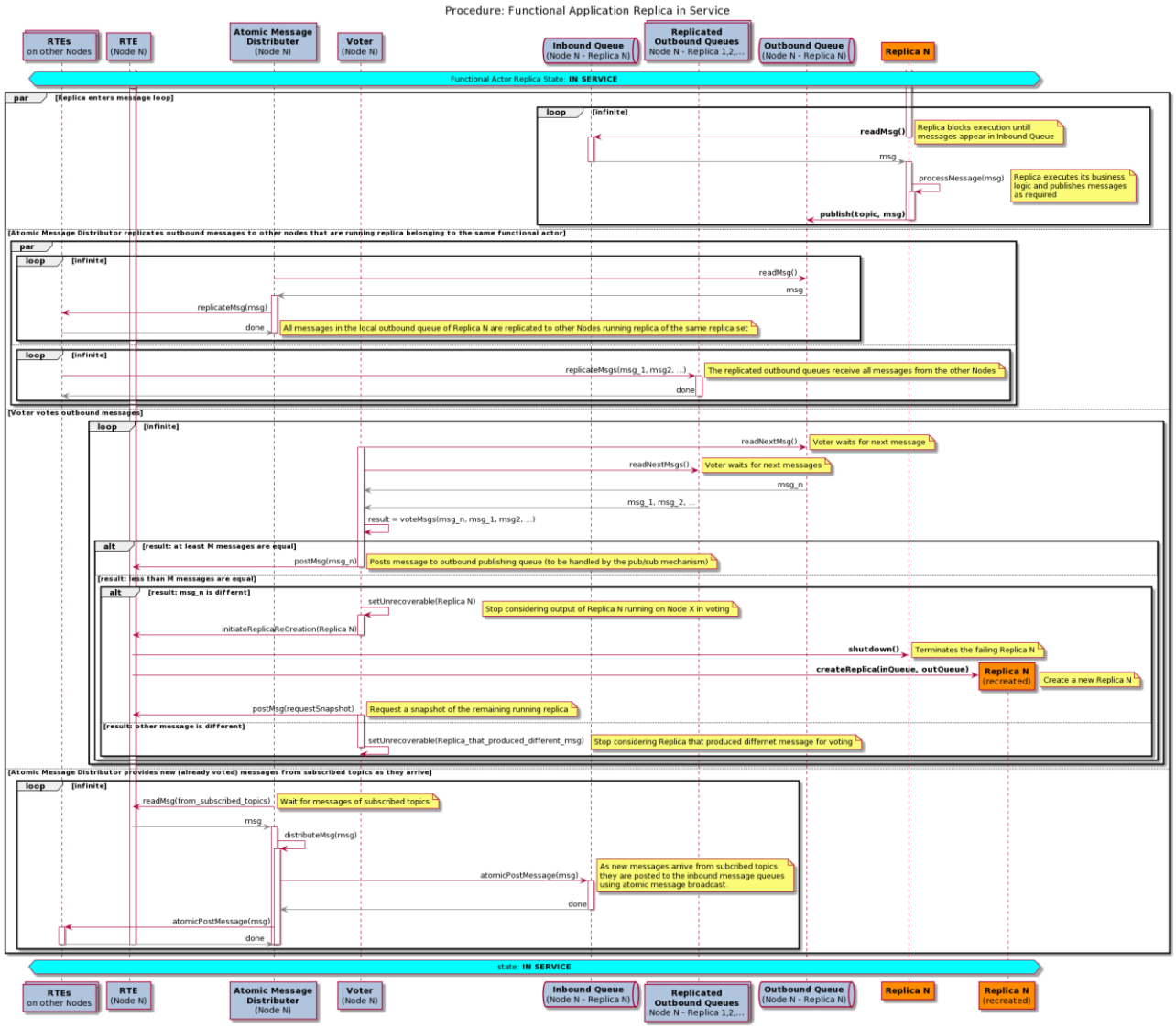

Figure 9. Procedure: Functional Actor Replica in Service (click on figure to open digitally).

The UML code of the sequence diagram shown in Figure 9 can also be found under: LINK

8.3. Shutdown of a Functional Actor Replica

To be written

8.4. Initialization of a Functional Actor

To be written

8.5. Functional Actor in Service

To be written

8.6. Shutdown of a Functional Actor

To be written

8.7. Initialization of a Functional Application

To be written

8.8. Functional Application in Service

To be written

8.9. Shutdown of a Functional Application

To be written

8.10. Initialization of a Platform

To be written

9. API Exchanges

In this section, all API exchanges are listed and defined in more detail that are required for the realization of the procedures described in Section 8. It is expected that these API exchanges are a key part of the PI API specification.

9.1. API Exchanges related to Control

The API exchanges described in Table 3 are required for the basic interaction among Platform and Functional Application Replica in the context of the initialization or shutdown of a Replica, or when an error has occurred.

Table 3. List of API exchanges related to control.

Sequence Diagrams	API Exchange	Description	Information Exchanged	Trigger
Figure 8, Figure 9	**createReplica** ()	Creates a Replica running in a dedicated Partition	**Platform -> Replica: inQueue:** Inbound message queue to be used by the Functional Actor Replica	Platform

			outQueue: Outbound message queue to be used by the Functional Actor Replica	
Figure 8	**readConfig()**	Acquire configuration data provided to the Functional Actor Replica by the Platform at startup.	**Platform -> Replica:** **appl_cfg_data:** Configuration data that the application needs for initialization **interop_cfg_data:** Configuration data used by both Platform and Functional Actor Replica	Functional Actor Replica
Figure 8	**getSnapshot()**	Requests the Platform to provide the snapshot to be restored	**Platform -> Replica** **snapshot:** complete information required to restore the Functional Actor Replica to a specific state	Platform
Figure 8, Figure 9	**shutdown()**	Initiates a Functional Actor Replica shutdown.	**-**	Platform
-	**reportIssue()**	Allows a Functional Actor Replica to report that some error has been observed (e.g., some inconsistency in state), based on which the Platform could take similar measures as if it would have realized itself that there is an issue with the Functional Application (e.g., during voting). The exact necessity and usage of such function is FFS.	**Replica -> Platform** **issue**: information identifying the problem that the Functional Actor Replica encountered	Functional Actor Replica

9.2. API exchanges related to IO

The API exchanges listed in Table 4 relate to the exchange of control or application messages among Functional Actor Replicas and the Platform.

Table 4. List of API exchanges related to IO.

Sequence Diagrams	API Exchange	Description	Information Exchanged	Trigger
Figure 8	**register**()	Allows the Functional Actor Replica to register a topic it intends to post messages to	**Replica > Platform:** **topic:** unique name of topic to register	Functional Actor Replica
-	**unregister**()	Allows the Functional Actor Replica to unregister a topic (e.g., when the related Functional Actor is being shut down)	**Replica -> Platform:** **topic:** unique name of topic to unregister	Functional Actor Replica
Figure 8	**subscribe()**	Allows the Function Actor Replica to subscribe to a topic for which it wants to receive messages	**Replica -> Platform:** **topic:** unique name of topic to subscribe to	Functional Actor Replica
-	**unsubscribe()**	Allows the Function Actor Replica to unsubscribe from a topic	**Replica -> Platform:** **topic:** unique name of topic to unsubscribe from	Functional Actor Replica
Figure 8, Figure 9	**publish()**	Allows the Functional Actor Replica to publish a message (control message or other) on a certain topic.	**Replica -> Platform:** **topic:** unique name of topic to publish to **msg:** message to be published	Functional Actor Replica
Figure 8, Figure 9	**readMsg()**	Provides the Functional Actor Replica with the next message (control message or other) from its incoming message queue.	**Platform -> Replica:** **topic:** unique name of topic the message belongs to **msg:** message data	Platform

9.3. API exchanges related to logging

The API exchanges listed in Table 5 refer to logging.

Table 5. List of API exchanges related to logging functions.

Sequence Diagrams	API Exchange	Description	Information Exchanged	Trigger
-	**log** ()	Allows the Functional Actor Replica to write log information of a certain log level (e.g., INFO, WARNING, ERROR). As described in Section 7, the actual log information is expected to be application-specific and transparent to the platform.	**Replica -> Platform:** **level:** identifier of the log level (e.g., INFO, WARNING, ERROR) **content:** application-specific log information	Replica

10. Desired Platform Capabilities not requiring detailed Standardization

In this section, Platform capabilities are described that could be realized based on the API exchanges defined in Section 9 without requiring detailed standardization. It is expected that these capabilities would provide large room for Platform vendor differentiation. The capabilities are here mainly described to demonstrate the large range of possibilities that already a small list of standardized API exchanges could provide.

Table 6. Possible Platform capabilities based on the defined API exchanges.

Possible Platform capability	Possible realization (based on defined API exchanges)
1) Capability to allow replacement of hardware while a functional actor is running	The Platform could create more Replicas of a Functional Actor than actually needed by the required MooN scheme. This would enable that some Replicas could be shutdown and the underlying Compute Nodes could be replaced, followed by the recreation of the replicas without effecting the operation of the Functional Actor. This approach would be based on proprietary variations of the procedures defined in Section 8, reusing the same set of API exchanges.

2) Updating / patching a Functional Application while Functional Application is in operation	The Platform could in principle allow to update or patch a Functional Application while this is in operation (assuming that the previous and new version of the Functional Application produce exactly the same output during the period when the updates/patches are performed). For this, the Platform could create a larger number of Functional Actor Replica than needed (same as in capability 1), shutdown some Replica and then restart these based on the updated/patched version. Also this approach would be based on variations of the procedures detailed in Section 8, and the same API exchanges.
3) Updating / patching the RTE while Platform is in operation	The Platform could in principle enable updates or patches on RTE Instances while the Platform is in operation. This could be done by initially creating more Functional Actor Replicas than needed (same as in capability 1), deactivating some of the Functional Actor Replicas and the Compute Nodes these have been running on, updating/patching the RTE on these Compute Nodes, and then re-creating the Functional Actor Replicas on these Compute Nodes again.

11. Aspects for further discussion

In the course of the work on the PI API specification, the following points could not be resolved and are left for further discussion:

- For the communication between Function Actor Replica and RTE, means for authentication and confidentiality may be necessary;
- It is not sure whether there has to be an overall (standardized) timing for Functional Actor Replica and the Platform (e.g., with fixed time slots).

References

[1] 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

[2] RCA/OCORA, “Generic Safe Computing Platform High-Level Requirements”, OCORA-TWS03-020, Version 2.01, July 2021, see https://github.com/OCORA-Public/Publication/blob/master/04_OCORA%20Delta%20Release/OCORA-TWS03-020_Computing-Platform-Requirements.pdf

[3] RCA architecture, see https://www.eulynx.eu/index.php/documents/published-documents

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

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