European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification Part 3: Mechanisms

BAE Ref No: IAWG-ECOA-TR-007
Dassault Ref No: DGT 144482-E

Issue: 5

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: *This specification represents the output of a research programme and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This version of documentation is recommended as appropriate for limited lab-based evaluation only. Product development should rely on the DefStan or BNAE publications of the ECOA standard.*

Contents

Figures

Tables

0 Introduction

0.1 Executive Summary

The European Component Oriented Architecture (ECOA®) programme represents a concerted effort to reduce development and through-life-costs of the increasingly complex, software intensive systems within military platforms.

ECOA® aims to facilitate rapid system development and upgrade to support a network of flexible platforms that can cooperate and interact, enabling maximum operational effectiveness with minimum resource cost. ECOA® provides the improved software architectural approaches required to achieve this.

The standard is primarily focussed on supporting the mission system software of combat air platforms - both new build and legacy upgrades - however the ECOA® solution is equally applicable to mission system software of land, sea and non-combat air platforms.

The ECOA® specification is documented in ten parts, collectively identified as the Architecture Specification.

0.2 Main Introduction

This Architecture Specification provides the specification for creating ECOA®-based systems. It describes the standardised programming interfaces and data-model that allow developers to produce ECOA® components and construct ECOA®-based systems. It uses terms defined in the Definitions (Architecture Specification Part 2). The details of the other documents comprising the rest of this Architecture Specification can be found in Section 3.

This document is Part 3 of the Architecture Specification; it acts as an introduction to ECOA®.

The document is structured as follows:

- Section 6 provides an overview of the mechanisms that are used for interactions between Modules in the system.
- Section 7 describes in details the behaviour of the interactions in an ECOA® system.
- Section 8 describes the System Management mechanisms that are provided by the Infrastructure.
- Section 9 describes the support for scheduling within an ECOA® system.
- Section 10 describes the mechanisms for managing Service Availability in an ECOA® system.
- Section 11 describes Service Link behaviour.
- Section 12 describes Module Operation behaviour.
- Section 13 describes the utilities provided by the ECOA® Software Platform.
- Section 14 describes how inter-platform communication occurs within an ECOA® system.
- Section 15 describes the concept of a composite (collection of Application Software Components)

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of software systems using a component oriented approach.

2 Warning

This specification represents the output of a research programme and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This standard of documentation is recommended as appropriate for limited lab-based evaluation only. Product development based on this standard of documentation is not recommended.

3 Normative References

Architecture Specification Part 1	IAWG-ECOA-TR-001 / DGT 144474 Issue 5 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	IAWG-ECOA-TR-012 / DGT 144487 Issue 5 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	IAWG-ECOA-TR-007 / DGT 144482 Issue 5 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	IAWG-ECOA-TR-010 / DGT 144485 Issue 5 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	IAWG-ECOA-TR-008 / DGT 144483 Issue 5 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	IAWG-ECOA-TR-006 / DGT 144481 Issue 5 Architecture Specification Part 6 – ECOA® Logical Interface
Architecture Specification Part 7	IAWG-ECOA-TR-011 / DGT 144486 Issue 5 Architecture Specification Part 7 – Metamodel
Architecture Specification Part 8	IAWG-ECOA-TR-004 / DGT 144477 Issue 5 Architecture Specification Part 8 – C Language Binding
Architecture Specification Part 9	IAWG-ECOA-TR-005 / DGT 144478 Issue 5 Architecture Specification Part 9 – C++ Language Binding

Architecture Specification Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 5 Architecture Specification Part 10 – Ada Language Binding
Architecture Specification Part 11	IAWG-ECOA-TR-031 / DGT 154934 Issue 5 Architecture Specification Part 11 – High Integrity Ada Language Binding
ISO/IEC 8652:1995(E) with COR.1:2000	Ada95 Reference Manual Issue 1
ISO/IEC 9899:1999(E)	Programming Languages – C
ISO/IEC 14882:2003(E)	Programming Languages C++
SPARK_LRM	The SPADE Ada Kernel (including RavenSPARK) Issue 7.3
sca-assembly-1.1-spec-cd03	Service Component Architecture Assembly Model Specification Version 1.1 Available at: http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd03.pdf

4 Definitions

For the purpose of this standard, the definitions given in Architecture Specification Part 2 apply.

5 Abbreviations

API	Application Programming Interface
ASC	Application Software Component
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
ELI	ECOA® Logical Interface
FIFO	First In, First Out
ID	Identifier
OS	Operating System
PINFO	Persistent Information
QoS	Quality of Service
UDP	User Datagram Protocol
XML	eXtensible Markup Language
XSD	XML Schema Definition

6 ECOA Mechanisms

The Architecture Specification Part 1 defines an architecture which uses Application Software Components and Services. This document describes the mechanisms defined by ECOA and the way that Components interact. Additionally, it describes the behaviour of other aspects of an ECOA system including management and utility functions along with how different ECOA Software Platforms interact.

Some of the mechanisms are described in detail within this document, whereas others are only discussed at a high level, as they are covered in greater depth in other documents. Where this is the case a reference will be provided.

The intended audience for this reference manual is:

- Component Developers:
  - To understand the mechanisms available for developing applications
- ECOA Platform Developers:
  - To understand the behaviour an ECOA Platform is required to provide for a given mechanism

This document describes the mechanisms available to an Application Software Component, but it is the Architecture Specification Part 4 which provides the abstract API for implementing the mechanisms described herein.

7 Interactions

7.1 Module Interactions

Interactions between Module Instances in an ECOA system rely on three primary mechanisms:

- Events
- Request-Response
- Versioned Data publication

The interactions between Module Instances can occur within a single Application Software Component, or between Module Instances of different Application Software Components, as a consequence of their Services. For detail on the behaviours, see sections 11 and 12 respectively.

In addition to the above mechanisms, Operations exist for Infrastructure Services to allow the management of the runtime lifecycle, properties, logging, faults, time services, service availability, persistent information and context management.

The following sections include numerous figures, which illustrate the interactions within an ECOA system and provide visual clarity. The key shown in Figure 1 offers guidance on the colouring and symbology used throughout these sections.

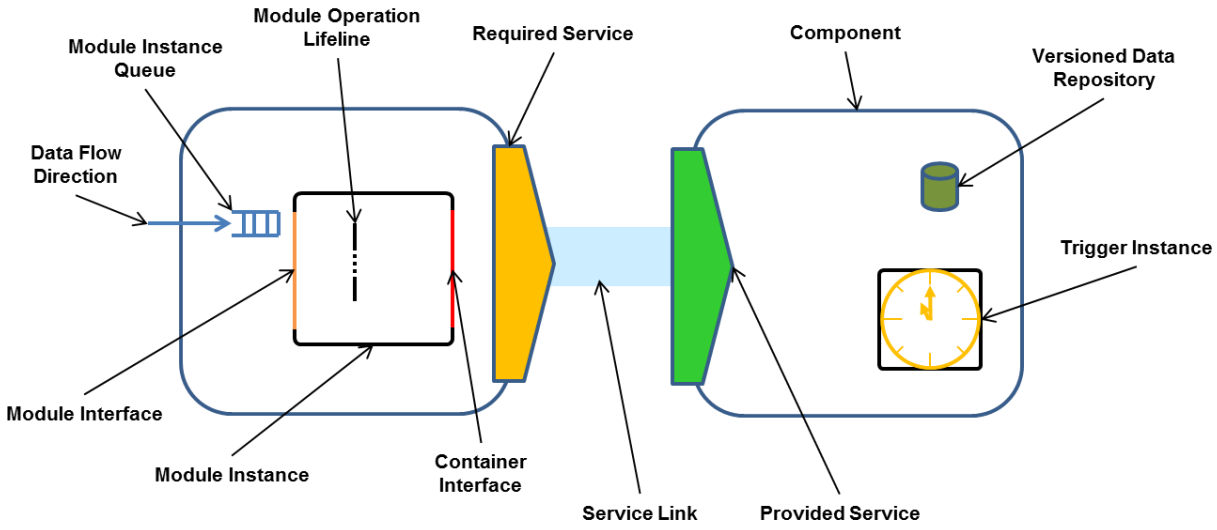

Figure 1 ECOA Interactions - Key

Note that the Component developer is only responsible for implementing the functionality within the Module Instance. The other infrastructure objects shown are the responsibility of the ECOA Platform Developer and comprise the Platform Integration Code (e.g. Trigger Instances, Module Instance Queues, Versioned Data repositories, Service Links etc.).

7.1.1 Timestamping

Freshness of data is an important consideration in a mission system and for this reason timestamping of operations (i.e. communication) is supported.

A timestamp point is related to the origin of the operation (Sender). The timestamp allows the user of the timestamp to rebuild a chronogram based on the same reference, the sender's clock. The ECOA infrastructure (i.e. container) will be able to record timestamps for operations as shown in Table 1.

Table 1 Timestamp Points

Operation API	Timestamp point	Description
`Event_Send`	When Module calls the Container API	When an event is sent by the requiring or providing Container
`Request_Sync`	When Module calls the Container API	When a request is sent by the requiring Container
`Request_Async`	When Module calls the Container API	
`Response_Send`	When the Module calls the Container API	When a response is sent by the provider Container
`Publish_Write_Access`	When Module calls Container API (publish)	When data is published by the provider Container
`Updated`	When Module calls Container API (publish)	When data is published by the provider Container

At present operation timestamps will always be provided by the infrastructure. In future timestamps may be made optional (for performance reasons) and a component may be able to specify whether it provides or requires a timestamp.

The timestamp associated with the sent event, sent request or sent response are available to the receiving module in the Module Instance context.

The timestamp associated with the publication of versioned data is available within the version data handle.

7.2 Module Instance Queues

Module run-time behaviour is dependent upon the Module Runtime Lifecycle state (see 8.1). A set of predefined Module Operations called Module Lifecycle Operations exist to allow the Container to inform the Module of changes to its Lifecycle. Module Lifecycle Operations are handled in any state (to enable the Lifecycle of a Module Instance to be managed), whereas normal Module Operations are only handled in the **RUNNING** state.

Module Operation calls are placed in the Module Instance Queue, and the corresponding entry-point for the Module Instance is invoked when the Operation reaches the front of the queue (if the Operation is specified as an activating Operation, see section 9.2 for further detail on activating and non-activating Operations).

Module Operation calls other than Module Lifecycle Operations are only queued if the Module Instance is in the **RUNNING** state and if, for a particular Module Operation, the maximum number of waiting operation calls does not reach the value given by the attribute fifoSize defined on the receiving part of the associated OperationLink.

If the Module Instance is not in the **RUNNING** state Module Operations are discarded. For Request operations arriving to a non-RUNNING Module, the Container will directly return a Response indicating that the Operation is not available.

7.3 Event

The Event mechanism is used for one-way asynchronous “push-style” communication between Module Instances and may optionally carry typed data.

When Events are used to implement a Service Operation, a Module Instance may be either the sender or receiver of an Event irrespective of whether it is designated as the Provider or Requirer of the Service.

Events are “wait-free” and “one-way”: the Sender is never blocked and does not receive any feedback from the Receiver. Events arriving on a full Receiver queue are lost, and the fault is reported to the fault-management Infrastructure.

There may be multiple receivers of an Event within a Component (e.g. other Module Instances or Service Instances), in which case instances of the Event are broadcast to all receivers.

If an Event arrives at a Service Instance that is declared unavailable, the Event is discarded silently.

For Events between Component instances, the behaviour is defined by the Rank and allEventsMulticasted attributes associated with the Service Link. If the Service Link is not identified as allEventsMulticasted; then only the Component instance connected to the Service Link with the lowest value of Wire Rank shall receive the Event. Further detail of this behaviour is described in section 11.

7.3.1 Event Sent by Provider

In the case of an Event sent by Provider, the providing Application Software Component initiates the sending. This behaviour is shown in Figure 2.

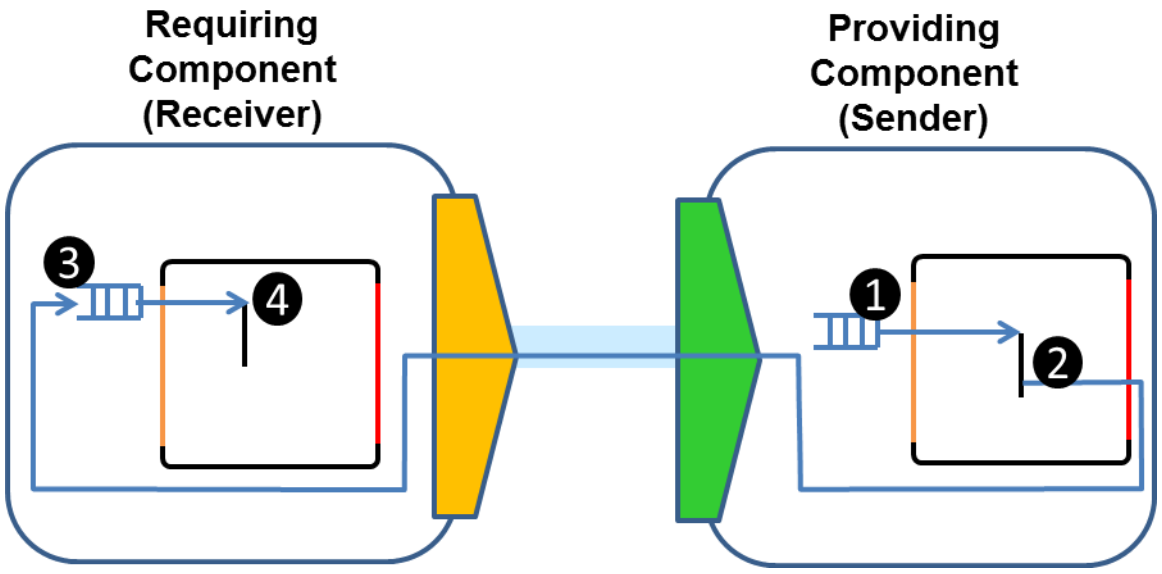

Figure 2 Event Sent by Provider

Figure 2 shows, at **point 1**, a Module Operation being invoked on the sender Module Instance (of the Providing Component instance) as a result of some other activity. During this execution, the Module Instance performs an Event Send Container Operation (Sent by Provider) at **point 2**. The Send operation returns immediately, allowing the Sender Module Instance to continue its execution. The Event will be queued on the Receiver Module Instance Queue, (of the Requiring Component instance) shown at **point 3**. The appropriate Event Received Module Operation will then be invoked on the Receiver Module Instance when the queue is processed and the Event reaches the front of the queue, at **point 4**.

7.3.2 Event Received by Provider

In the case of an Event received by Provider, the requiring Component initiates the sending. This behaviour is shown in Figure 3.

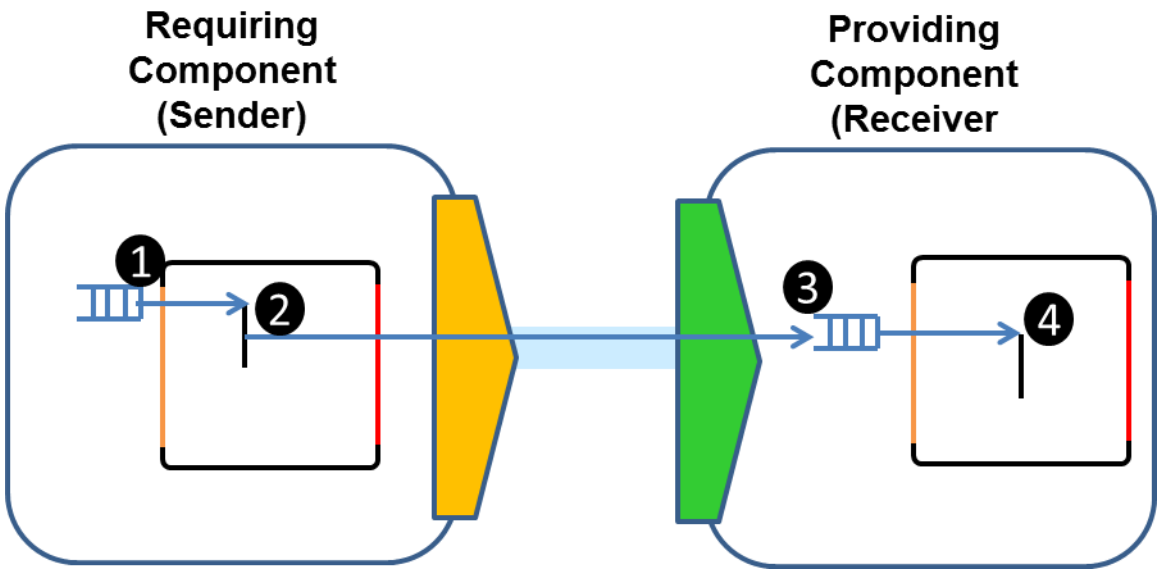

Figure 3 Event Received by Provider

Figure 3 shows, at **point 1**, a Module Operation being invoked on the Sender Module Instance (of the requiring Component instance) as a result of some other activity. During this execution, the Module Instance performs an Event Send Container Operation (sent by requirer) at **point 2**. The Event Send operation returns immediately, allowing the initiating Module Instance to continue its execution. The Event will be queued on the Receiver Module Instance Queue (of the Providing Component instance) shown at **point 3**. The Event Received Module Operation will then be invoked on the Receiver Module Instance when the queue is processed and the Event reaches the front of the queue, at **point 4**.

7.4 Request Response

The “Request-Response” mechanism is a two-way communication between Module Instances. The calling Module Instance Requests an operation and the called Module Instance provides a Response. The Requesting Module Instance (sender of the Request) is named the “Client”, and the providing Module Instance (sender of the Response) is named the “Server”.

A Request may carry data (“in” parameters) and the Response may also carry data (“out” parameters). All parameters are named and typed.

There are two mechanisms for Request operations which provide synchronous and asynchronous behaviour at the Client and one mechanism for Response operations at the Server. The details of these are described in the following sections. Note that the choice of mechanism for either a Request or a Response operation can be completely independent of each other.

For each Request-Response, the set of possible Clients and Servers are identified at design time, as is the type of the mechanism e.g. synchronous/asynchronous.

When a Client performs a Request, if the Server is a Module Instance within the same Component instance, then this Server is used. However, if the Request is connected to a Service, there may be multiple possible Servers available; only the Server connected with the Service Link with the lowest value of Wire Rank is used (known as the active Server, see Section 10). A Response from a Server is only sent to the particular Client that has issued the Request.

A call may fail if the Server is not available. The Client is notified of the failure of the call, and the fault reported to the fault-management Infrastructure.

The client Container instance implements a timeout in order to unblock the Client of a Synchronous Request-Response or to inform the Client of an Asynchronous Request Response if no Response is received within the given delay. The value of the timeout is defined at component implementation level; it can be determined by the maximum Response time defined by the required QoS. If the Response arrives after the timeout, the Response is discarded by the container and the fault is handled by the fault management. If the timeout is set to <0, it is considered as infinite; the Client of a Synchronous Request-Response remains blocked indefinitely.

The three types of Request-Response are detailed in the following sections.

7.4.1 Synchronous Request

In the case of a Synchronous Request, the Client Module Instance is blocked until the Response is received, as shown in Figure 4.

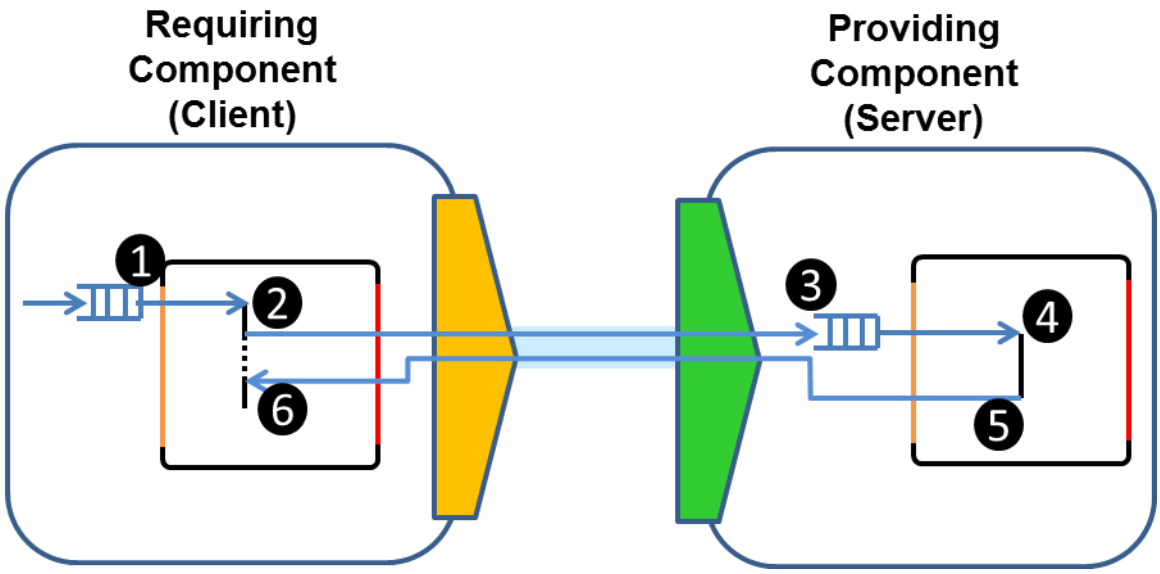

Figure 4 Synchronous Client Request-Response

Figure 4 shows, at **point 1**, a Module Operation being invoked on the Client Module as a result of some other activity. During this execution, the Module Instance performs a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked. When blocked, the Client Module Instance does not handle any other incoming Module Operation. From a module lifecycle point of view, this blocking situation is considered as a sub-state of the RUNNING state (see 8.1).

The Request operation is connected via a Service Link to the Server Module Instance, whereby the Request operation is queued in the Server Module Instance Queue, at **point 3**. The Request is accepted by the Server Module Instance as long as it has enough resources to handle the Response. If not, the Request is discarded.

The Request operation will be invoked on the Server Module Instance at **point 4**, which, in this example, will send the Response at **point 5** just before completing the Request operation. Once the Response is received by the Client Module Instance, it will become unblocked and can continue its execution at **point 6**.

7.4.2 Asynchronous Request

In the case of an Asynchronous Request, the Client is released as soon as the Request has been sent and may continue to execute other functionality. The Response results in the call of an operation on the Requesting Module Instance, as shown in Figure 5.

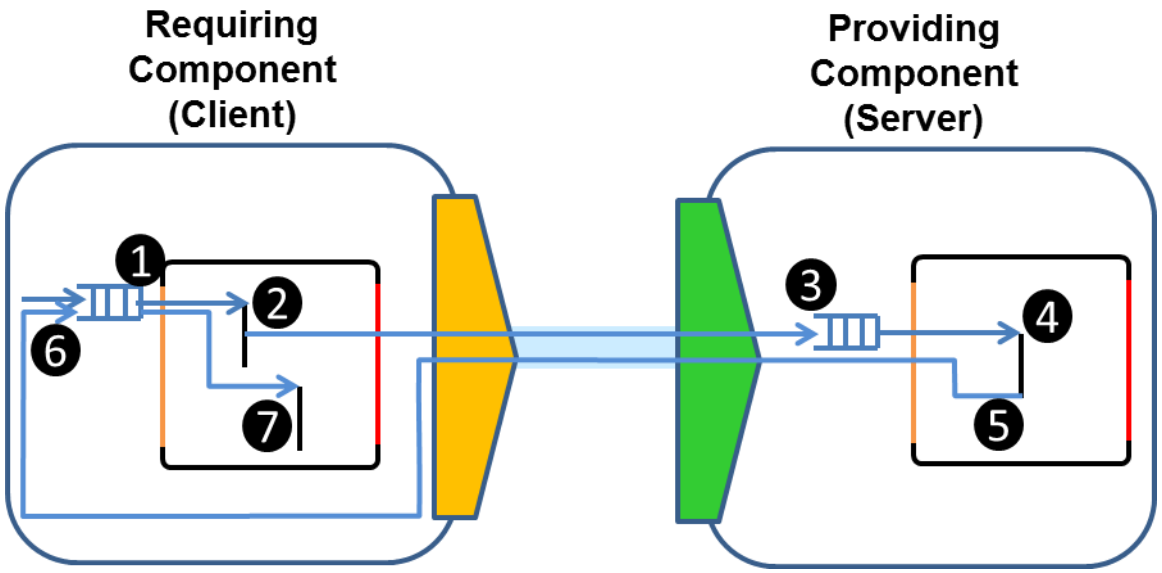

Figure 5 Asynchronous Client Request-Response

Figure 5 shows, at **point 1**, a Module Operation being invoked on the Client Module Instance as a result of some other activity. During this execution, the Module Instance performs an Asynchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance does **NOT** block meaning it can finish its execution of the invoked operation.

The Request operation is connected via a Service Link to the Server Module Instance, whereby the Request operation is queued in the Server Module Instance Queue, at **point 3**. The Request is accepted by the Server Module Instance as long as it has enough resources to handle the Response. If not, the Request is discarded.

The Request operation will be invoked on the Server Module Instance at **point 4**, which, in this example, will send the Response at **point 5** just before completing the Request operation. The Response will then be placed in the Client Module Instance Queue at **point 6**, and the Response call-back operation will be invoked on the Client Module Instance at **point 7**.

7.4.3 Response

The Server decides when it replies to the Client. It can be done in the same block of functionality associated to the Request (see 7.4.1 and 7.4.2) or the Server may defer the provision of the Response e.g. where it needs to invoke a Request-Response Service in order to provide the Response. In the second case the Server may continue to execute other functionality before providing the Response, as shown in Figure 6.

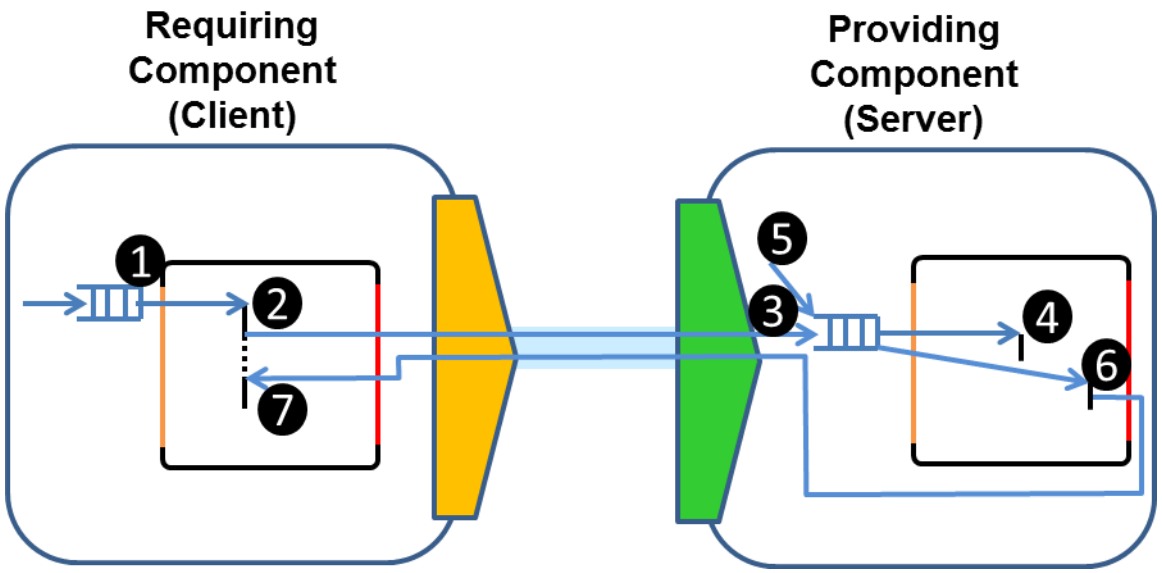

Figure 6 Deferred Response Server Request-Response

Figure 6 shows, at **point 1**, a Module Operation being invoked on the Client Module Instance as a result of some other activity. During this execution, the Module Instance performs, in this example, a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked.

The Request operation is connected via a Service Link to the Server Module Instance, wherein the Request operation is queued in the Server Module Instance Queue, at **point 3**. The Request is accepted by the Server Module Instance as long as it has enough resources to handle the Response. If not, the Request is discarded.

The Request operation will be invoked on the Server Module Instance at **point 4**. The Server may, by design, use the Response Container Operation to send the response at some later time. For example, in order to compute the Response, it may be necessary to invoke a further Asynchronous Request operation, meaning the original Response cannot be computed until receipt of this Response occurs.

Point 5, shows another Module Operation invoked on the Module Instance, during this execution, at **point 6**, the Response Container Operation is called to send the Response back to the Client. Once the Response is received by the Client Module Instance at **point 7**, it will become unblocked and can continue its execution.

7.5 Versioned Data Publication

The Versioned Data publication mechanism allows Module Instances to share typed data, according to a concurrency-safe read-write paradigm. A reading Module Instance is named a “Reader”, and a writing Module Instance is named the “Writer”.

The Writer can request a local copy of the data and subsequently commit or cancel any changes made. This write action is atomic. It means that the data is written entirely or not at all.

This write operation is independent of any other updates to the data-set. The Versioned Data writes are timestamped (timestamp is performed by the ECOA Software Platform).

A Reader can also request a local copy of the data, which will be the latest data value(s) at the time of the read request. This local copy will not be affected by any subsequent changes to the data-set (i.e. by a Writer updating the data-set). Note that although it is possible for the Reader to modify its local copy of the data, it is not able to update the global data-set.

Writers and Readers have to specify the beginning and the end of each (read or write) access to the data. The mechanism is "wait-free": no Writer or Reader is blocked waiting for another Writer or Reader to release the data or waiting for the underlying synchronisation mechanism to update the local data-set.

It is possible for multiple instances of the same Versioned Data repository to exist in an ECOA system e.g. where the Readers are on different Computing Nodes. When the access begins, the Writer or the Reader always gets the latest copy of data available locally. The timestamp enables the caller to determine the freshness of the data. The Reader gets an error if data has never been received or initialized.

Note that at the moment the data is requested, there is no guarantee that it is synchronised with the latest update, particularly in a distributed system. Where there are multiple Writers the timestamps can be used by any Readers to determine the order in which the data was published.

Any copy that is made to enable a read or write access is isolated, in that it will not be changed by any concurrent modifications of the data, and local changes will not cause the Versioned Data repository to be updated. The local copy is discarded after the read or write has been completed.

A write access ends with two possible alternatives:

- "publish" – modifications made by the Writer are published to the Versioned Data repository
- "cancel" – the modified local copy of the data is discarded without making any modifications to the Versioned Data repository.

Data publications are atomic; "simultaneous" publications of the same dataset cannot corrupt the data content. This behaviour may require support from the Infrastructure.

An optional attribute (maxVersions) may be set in the Component Implementation as an attribute of the data read/write operation to specify the maximum number of concurrent read or write accesses that may occur. The default is one access per operation e.g. a Read must be released before the next Read starts. A read or write access Request may fail if a new local copy of the data cannot be created. In this case, a null Data Handle is returned, the Client is notified of the failure of the call, and the fault reported to the fault-management Infrastructure.

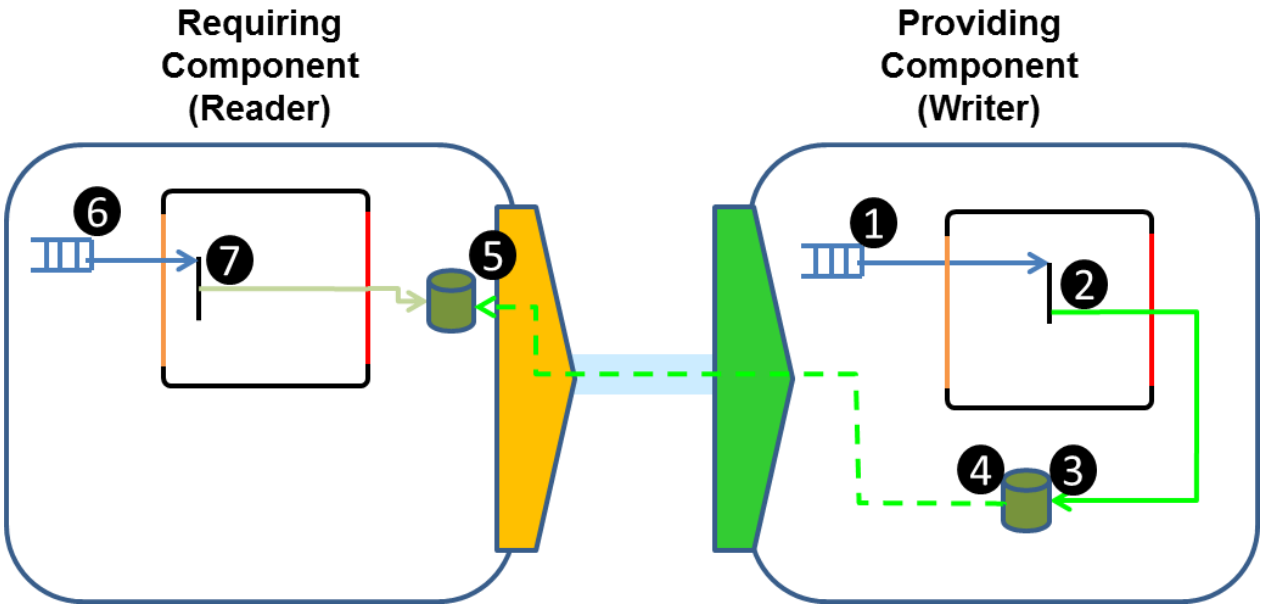

Figure 7 Versioned Data Behaviour

Figure 7 shows, at **point 1,** a Module Operation being invoked on the Writer Module Instance as a result of some other activity. During this execution, the Module Instance performs a publish Container Operation at **point 2**. The data is written to the Component instance local copy of the repository at **point 3**. The Infrastructure is then responsible for copying the data to any requiring Components (which may not be immediate depending upon the implementation of the Infrastructure) at **point 4** and **point 5** respectively. Also note that the Infrastructure may optimise this database management such that the 'local' copies are one in the same where the components are deployed in the same protection domain.

Independently, of the Writer, the Reader Module Instance can read from the Versioned Data repository. **Point 6** shows a Module Operation being invoked on the Reader Module Instance by some means (e.g. an Event Received). During this execution, the Module Instance performs a read Container Operation at **point 7**.

7.5.1 Notifying Versioned Data

Versioned Data Readers can also specify an optional attribute (notifying) to receive a notification of any updates to Versioned Data. This behaviour is achieved by queuing a notification Event on the Reader Module Instance, which also contains a reference to the latest version of the data.

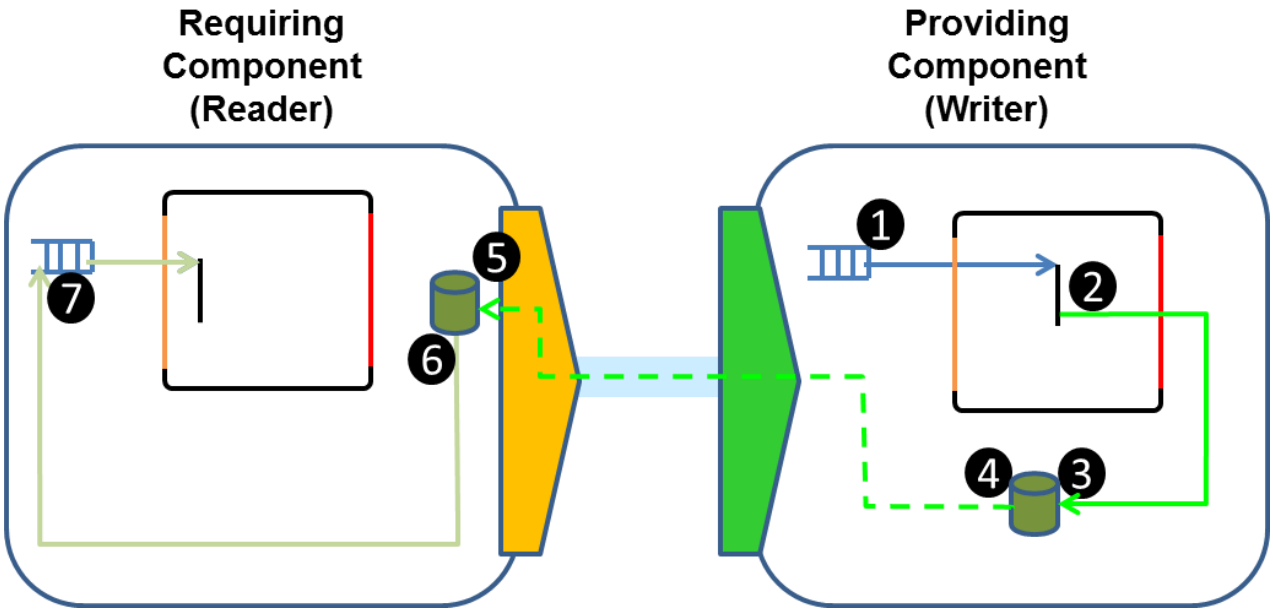

Figure 8 Notifying Versioned Data Behaviour

Figure 8 shows, at **point 1,** a Module Operation being invoked on the Writer Module Instance as a result of some other activity. During this execution, the Module Instance performs a publish Container Operation at **point 2**. The data is written to the Component instance local copy of the repository at **point 3**. The Infrastructure is then responsible for copying the data to any requiring Components (which may not be immediate depending upon the implementation of the Infrastructure) at **point 4** and **point 5** respectively.

When the requiring Component receives the updated data (and as the Reader Module Instance has defined the operation to be notifying) a notification Event is generated at **point 6** which is queued on the Reader Module Instance Queue at **point 7**. This invokes the notification Module Operation, which also includes a copy of the updated Versioned Data.

Note that specifying a Versioned Data Read operation as notifying does not preclude the use of standard Container Operations for getting a read-only copy of the data at any time, as detailed in section 7.5. Both mechanisms share the same underlying memory buffers.

7.6 Trigger

Triggers generate periodic Events which can be used to invoke some functionality provided by a Module Instance or set of Module Instances. The Trigger can generate Events which are queued to Module Instances within the same Application Software Component and/or generate Events which are queued to Module Instances in different Application Software Components via a Service e.g. where a single central Trigger is used to coordinate the execution of functionality of multiple Components, in the manner of a “central clock”. The Trigger behaviour for the case of generating events within a same Component is shown in Figure 9.

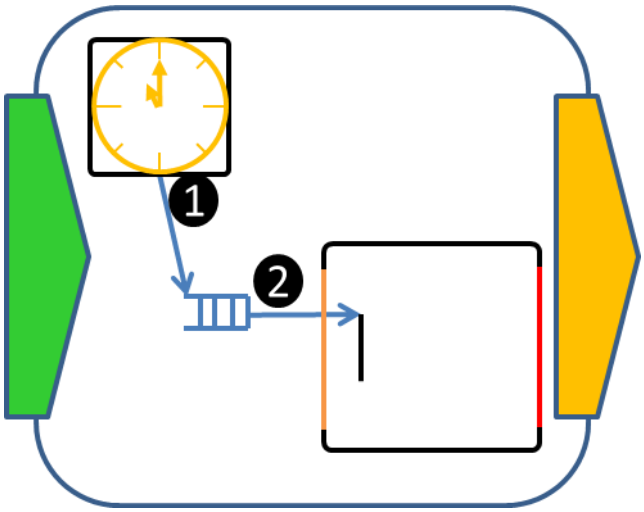

Figure 9 Trigger Behaviour

Figure 9 shows, at **point 1**, the Trigger Instance sending an Event to the Receiver Module Instance Queue. This causes the Module Operation connected to the Trigger to be invoked on the Receiver Module Instance at **point 2**. The Trigger Instance will generate the Event at a periodic interval as defined by the Component implementer. The first Event will be generated one period after the Trigger Instance has been started by the Supervision Module.

The Trigger Instance is provided by the underlying Infrastructure to generate an Event at a set time interval. The Trigger Instance exhibits a sub-set of the Module interface, to enable the Supervision Module to control the Module Lifecycle of the Trigger.

NOTE: the Trigger can be connected to one or more Module Operations and/or Service Operations. Each operation link may specify a different trigger period, so a single trigger may generate events at different periods.

7.7 Dynamic Trigger

A Dynamic Trigger sends an Event after a given delay (known as the `out` Event) from the receipt of an input Event (known as the `in` Event). The `in` Event specifies the delay time. A Dynamic Trigger may also receive a `reset` Event, which will purge all unexpired delays.

It is possible for multiple Module Instances to:

- Send `in` and `reset` Events to the same Dynamic Trigger.
- Receive the same `out` Event.

As with the periodic Trigger, the Dynamic Trigger can generate Events which are queued to Module Instances within the same Application Software Component and/or generate Events which are queued to Module Instances in different Application Software Components via a Service.

Multiple occurrences of the `in` Event may be processed such that a number of delays are waiting to expire. A `reset` Event is used to purge all currently active delays.

The first parameter of a Dynamic Trigger is the delay. The remaining parameters can be any pre-defined type. The `out` Event is generated with exactly the same parameters as the `in` Event, except that the first (delay) parameter is omitted. The `out` Event is sent at the time resulting from adding the timestamp of the `in` Event and the delay time. The timestamp of the `out` Event is the time at which the Event is sent by the Dynamic Trigger.

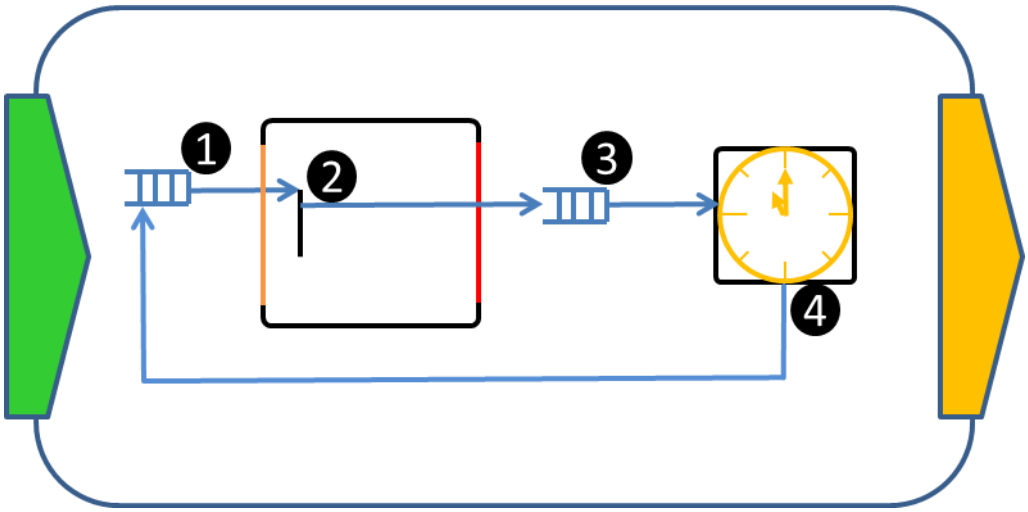

Figure 10 Dynamic Trigger Behaviour

Figure 10 shows, at **point 1,** a Module Operation being invoked on a Module Instance as a result of some other activity. During this execution, the Module Instance performs, at **point 2**, a Container Operation to Request the Dynamic Trigger Instance to send a Trigger Event after a given period (`in` Event). The `in` Event is queued in the Dynamic Trigger Instance Queue, at **point 3**, otherwise if the queue is full, the event is discarded and an error raised to the Fault Handler.

The queue is processed by the Dynamic Trigger, and if the maximum number of concurrent delays has been reached, the delay is discarded and an error raised to the Fault Handler, otherwise the delay is started. After the delay time has expired, the Dynamic Trigger Instance will generate an `out` Event to the Receiver Module Instance Queue, shown at **point 4**.

The Dynamic Trigger Instance exhibits a sub-set of the Module interface, to enable the Supervision Module to control the Module Lifecycle of the Dynamic Trigger.

7.7.1 Dynamic Trigger Operations

The Dynamic Trigger can be considered as a Module whose operations are:

- EventReceived in
  - On reception, the Dynamic Trigger sets the trigger for the time calculated from the timestamp at which the “in” was sent+delay
  - Parameters:
    - delay : ECOA:duration (seconds, nanoseconds)
    - p1, p2, etc: ECOA types
- EventSent out
  - The Dynamic Trigger sends an “out” Event at the time calculated from the timestamp at which the corresponding “in” was sent+delay. The timestamp of "out" is the time at which "out" is sent.
  - Parameters:
    - p1, p2, etc: are identical (number, types and position identical to those of the “in” Event)
- EventReceived reset
  - On reception, the Dynamic Trigger cancels the trigger settings for all "in" Events already previously received and not yet expired.

The transmitted Events ‘in’ and ‘out’ can have several other parameters (p1, p2, etc.):

- These parameters are sent unchanged by the Dynamic Trigger.
- The number of parameters and their types are defined in the Component implementation model at instance definition level.

7.8 Interactions within Components

Although the majority of examples shown in the preceding sections display interactions between Module Instances in multiple Components (using Services), the exact same interactions can occur within a Component boundary (Module to Module communications).

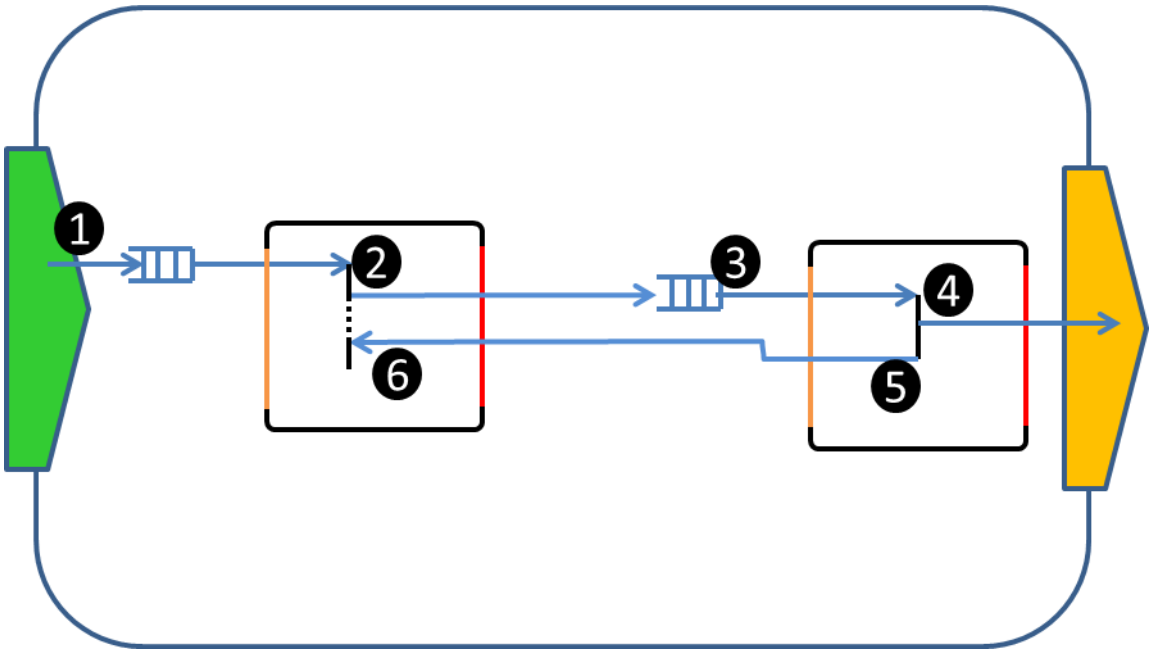

Figure 11 Interactions within Components – Synchronous Request-Response

Figure 11 shows the interactions of a Synchronous Request –Response operation between two Module Instances within a Component. At **point 1**, a Module Operation is invoked on the Client Module Instance from a Service Operation.

During this execution, the Module Instance performs a Synchronous Request operation at **point 2**. At the point the Request is made, the Client Module Instance becomes blocked.

The Request operation is connected to another Module Instance within the Component. The Request operation is queued in the Server Module Instance Queue, at **point 3**. The Request operation will be invoked on the Server Module Instance at **point 4**, which can peform any processing required in order to produce the Response.

In this example, at **point 4**, the Module Instance invokes a Container Operation to perform an Event Send. The Server Module Instance sends the Response at **point 5** just before the Module Operation completes and returns control to the Container. Once the Response is received by the Client Module Instance, it will become unblocked and can continue its execution at **point 6**.

7.9 Assembly, Component and Module Properties

Components may be instantiated multiple times within a system. Component Properties provide a means to tailor the behaviour of a Component Instance. A Component Property is declared as part of the Component Definition.

Within the Assembly Schema the Component Instance Property Values are assigned for each Component Instance. These values are assigned at design time and cannot be changed during execution.

A Component Instance Property Value may either be a literal value or a reference an Assembly Property Value. An Assembly Property Value is a property which is accessible by all Components within the Assembly.

As part of a Component Implementation a Module Type can have Module Properties declared, which can then be used to tailor different behaviour for each Module Instance.

For each Module Instance, a Module Property can either reference a Component Property, or be assigned a value at Component Implementation time (design time) which cannot be changed during execution. Note that in order for Component Properties to be accessed; a Module Property must be created to reference the Component Property.

Module Instances can then access Module Properties, and consequently Component Properties if referenced, via the Container Interface at run-time. The Architecture Specification Part 4 contains more detail regarding Properties.

7.10 Persistent Information

Persistent information is broken down into two main categories related to the extent of its accessibility as follows:

- Private PINFO – is data accessible by Module Instances within the same ASC Instance.
    - Private PINFO can be defined in Read-Only or Read-Write mode
- Public PINFO – is data accessible by any Module Instance in the Assembly Schema.
    - Public PINFO is Read-Only

PINFO defined in Read-Only mode can be referenced (i.e. accessed) by multiple Module Instances.
PINFO defined in Read-Write mode can only be referenced (i.e. accessed) by a single Module Instance.

Figure 12 illustrates the aspects of PINFO and how they are being declared in an ECOA system.

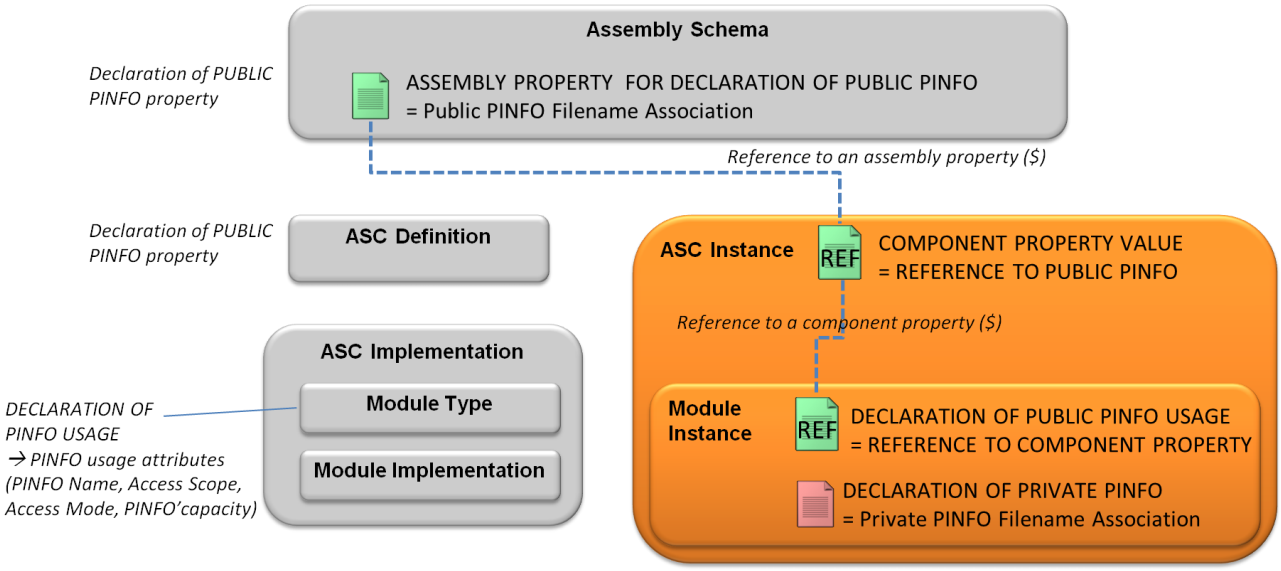

Figure 12 Aspects of PINFO

7.10.1 Private PINFO

7.10.1.1 Private PINFO Attributes

Private PINFO has the following attributes associated with it:

- PINFO name:
  - This is the name used at Module level for accessing the PINFO. It is NOT the filename associated to the PINFO.
- Access Mode:
  - 'Read-Only' or 'Read-Write'.
- PINFO'capacity:
  - The maximum size of PINFO (applicable only in 'Read-Write' access mode).
- Filename Association: (Path and filename)
  - This is where the file containing PINFO data is stored prior to deployment on the target ECOA Platform.

Declaration of private PINFO is bounded by the following rules:

- PINFO Filename Association is declared as a string:
  - The maximum length of that string is 256 characters,
  - Regarding PINFO path, the characters can be alphabetical, numerical or special characters among "_", ".", "/"
  - Regarding PINFO name, the characters can be alphabetical, numerical or special characters among "_"
- PINFO Filename Association is not case sensitive.
- A Filename Association can be associated with only one Private PINFO in read-write mode, among all Module Instances of a given Component Implementation.

- A Filename Association can be associated with more than one Private PINFO in read-only mode, among all Module Instances of a given Component Implementation.
- Private PINFO are stored in '4-ComponentImplementations/<Component Implementation Name>/Pinfo' directory, and their Filename Association is declared as a relative path to that directory (sub-directories are allowed).

NOTE: This follows the interim data organisation structure as defined in Architecture Specification Part 7.

7.10.2 Public PINFO

7.10.2.1 Public PINFO Attributes

Public PINFO has the following attributes associated with it:

- PINFO name:
    - This is the name used at Module level for accessing the PINFO. It is NOT the filename associated to the PINFO.
- Filename Association: (Path and filename)
    - This is where the file containing PINFO data is stored prior to deployment on the target ECOA Platform.

Declaration of public PINFO is bounded by the following rules:

- PINFO Filename Association is declared as a string:
    - The maximum length of that string is 256 characters,
    - Regarding PINFO path, the characters can be alphabetical, numerical or special characters among "_", ".", "/"
    - Regarding PINFO name, the characters can be alphabetical, numerical or special characters among "_"
- PINFO Filename Association is not case sensitive.
- Public PINFO are stored in '5-Integration/Pinfo' directory, and their Filename Association is declared as a relative path to that directory (sub-directories are allowed).

NOTE: This follows the interim data organisation structure as defined in Architecture Specification Part 7.

7.10.3 PINFO organization

Prior to deployment on the target ECOA Platform(s), PINFO is organised as follows:

- PINFO are stored in directories specified in §7.10.1.1 and §7.10.2.1.
- Private PINFO is common to all ASC Instances of a given ASC Implementation (as it is stored at ASC Implementation level)

This organisational structure is illustrated in Figure 13.

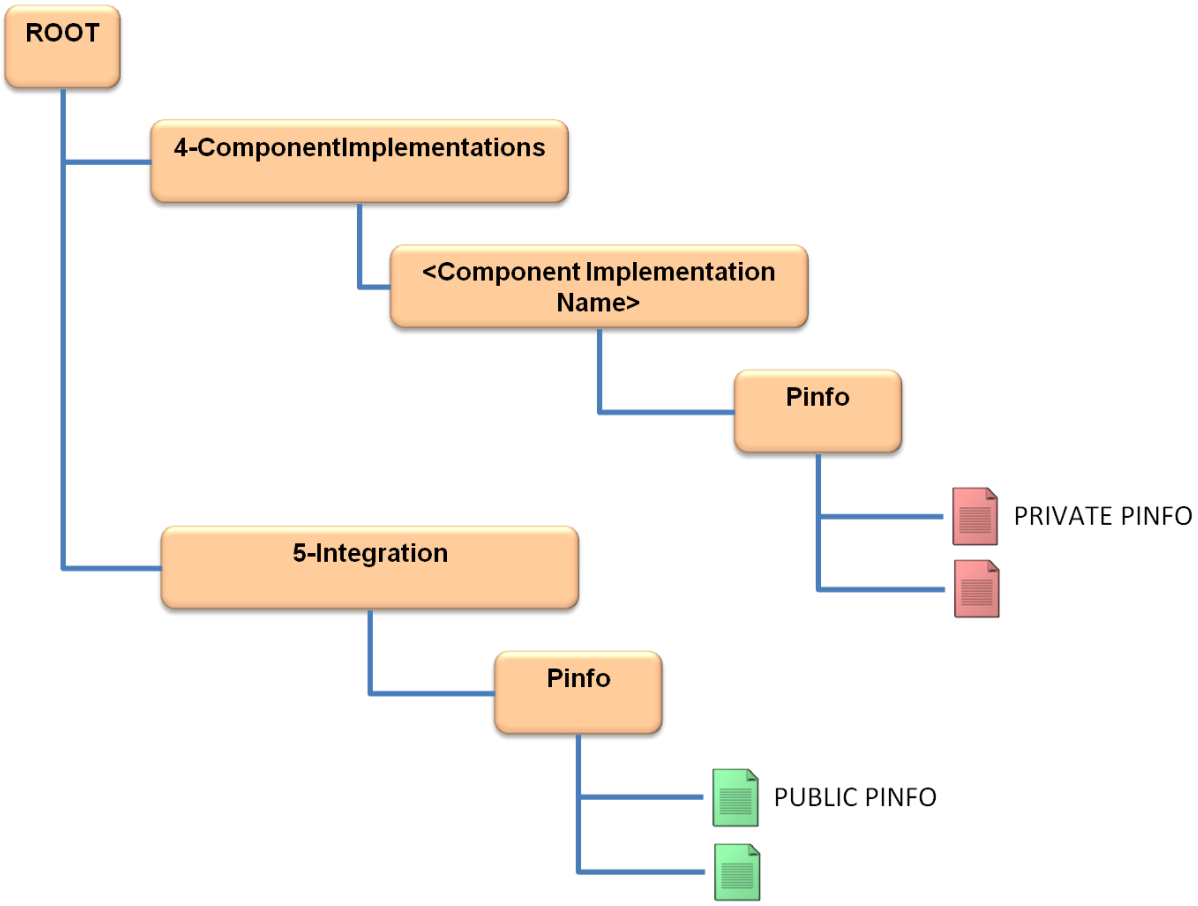

Figure 13 PINFO organization prior to deployment

- Subsequent to deployment on the target ECOA Platform(s), PINFO is organised as follows:
  - PINFO are stored in the ECOA Infrastructure, which is specific to each ECOA Platform.
  - For each ASC Implementation, the ECOA Infrastructure shall duplicate all Private PINFO into as many copies as there are ASC Instances of that ASC Implementation.
  - The ECOA Infrastructure shall only let Module Instances of a given ASC Instance access to the Private PINFO copies related to that ASC Instance.
    - Rationale: this is aimed at complying with Access Scope as defined in §7.10

The organisational structure after deployment is illustrated in Figure 14

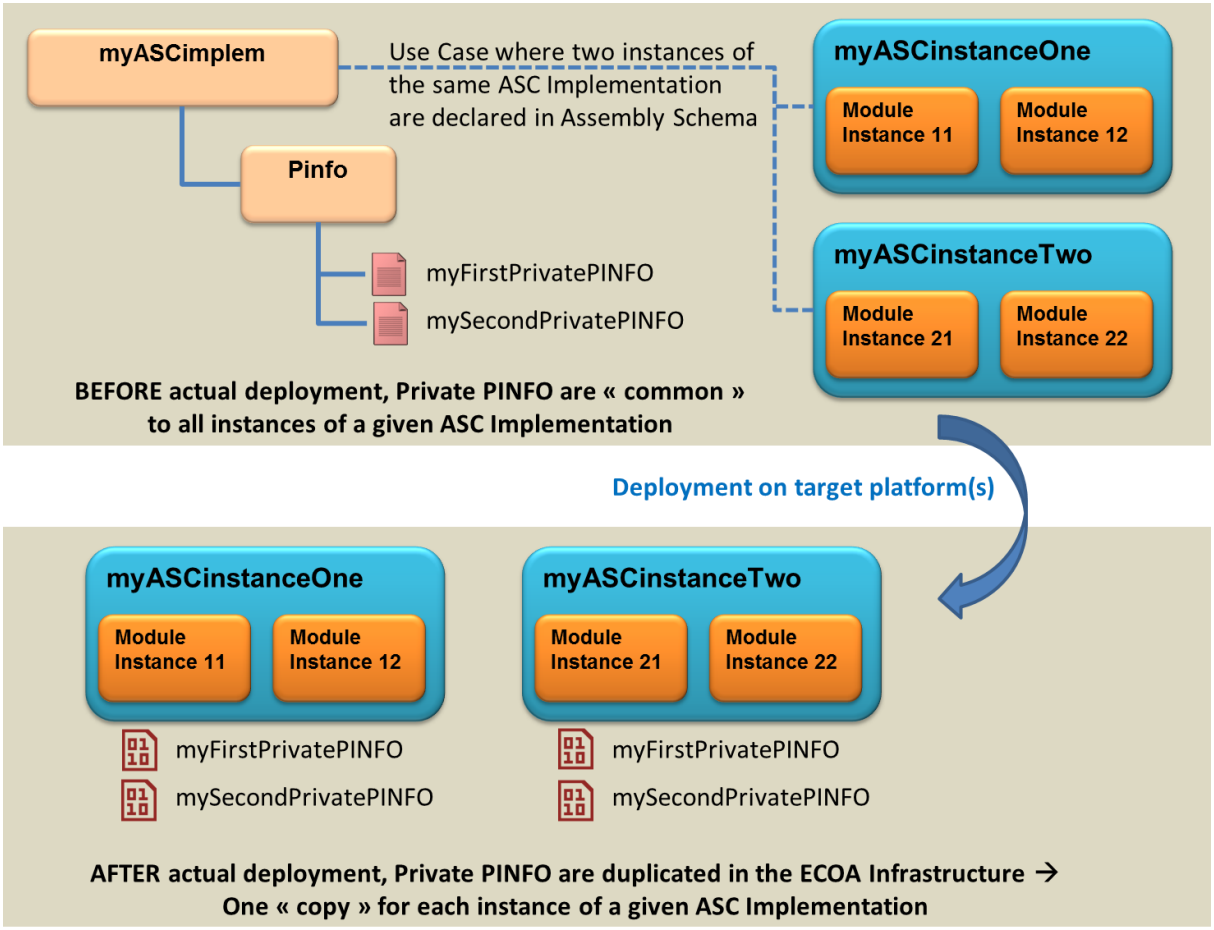

Figure 14 Focus on Private PINFO organization before and after deployment

7.10.4 PINFO API

The ECOA Infrastructure is responsible for making PINFO accessible to Module Instances:

- PINFO used by a Module Instance should be made accessible to that Module Instance prior to the INITIALIZE operation being invoked when it is in the IDLE state,
- PINFO used by a Module Instance should cease to be accessible to that Module Instance when it goes to the 'IDLE' state and any ongoing SHUTDOWN operation has returned.

From a Module Instance point of view, PINFO has the following form, illustrated by Figure 15:

- PINFO is an array of bytes,
- PINFO has an index, denoted by PINFO'index which indicates the current position in the array of bytes at which the next access will occur:
  - PINFO'index is expressed as a number of bytes,
  - PINFO'index starts from zero.
- PINFO has an upper bound denoted by PINFO'capacity (for 'Read-Write' access mode, as defined in §7.10.1.1),
- PINFO has a current size denoted by PINFO'size.

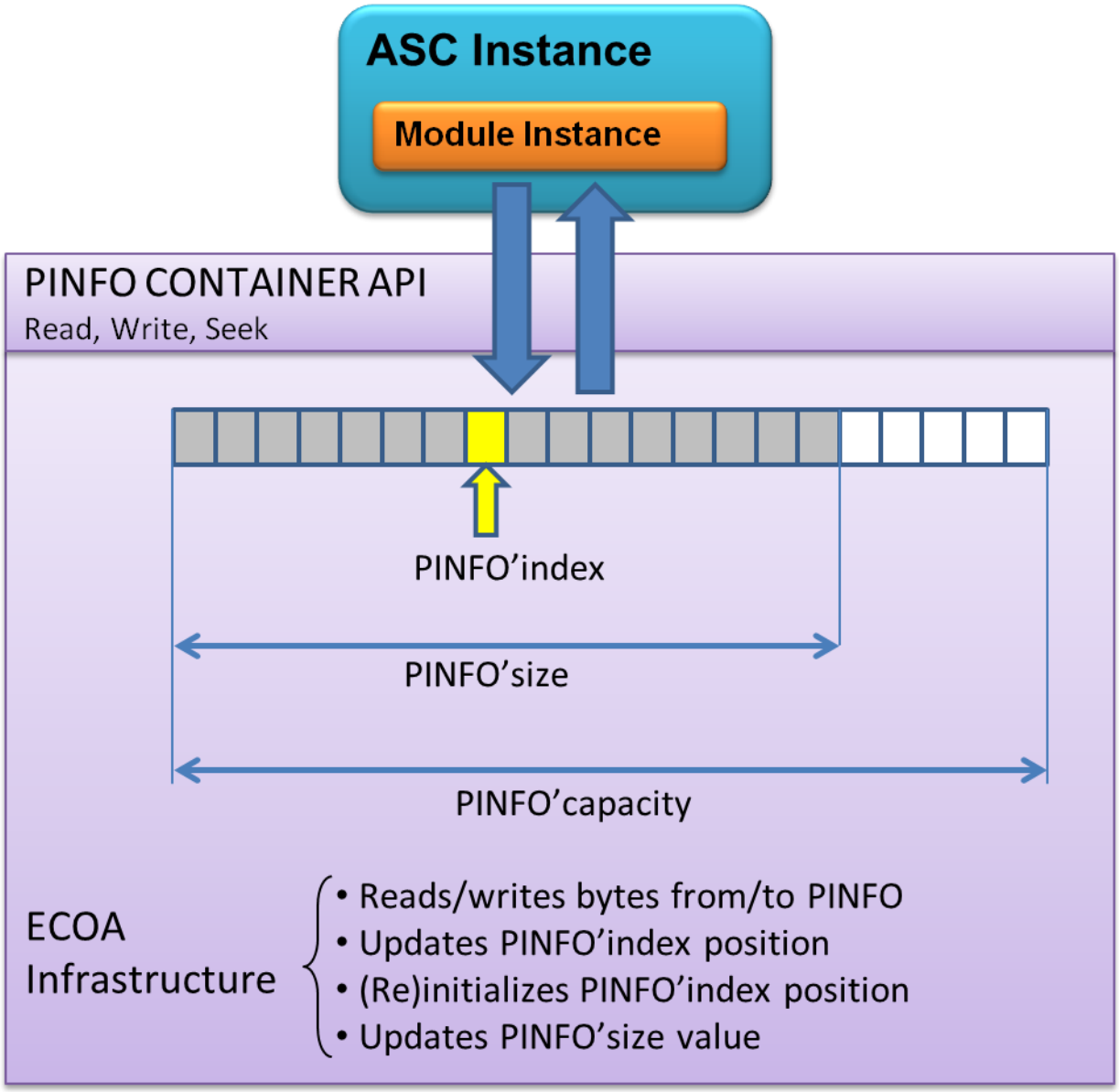

Figure 15 PINFO API

Module Instances invoke the following Container API for accessing PINFO:

- PINFO Read:
  - The Module Instance asks its Container to read a number of consecutive bytes in the PINFO, starting at PINFO'index position.
  - The Container provides the bytes read to the Module Instance.
  - The Container ensures that PINFO'size is not exceeded.
  - The Container increments PINFO'index according to the number of read bytes.
- PINFO Write:

- The Module Instance asks its Container to write a number of consecutive bytes into the PINFO, starting at PINFO'index position.
- The Container writes these bytes into the PINFO.
- The Container ensures that PINFO'capacity is not exceeded.
- The Container increments PINFO'index and PINFO'size according to the number of written bytes, with regard to PINFO'capacity upper bound.

- PINFO Seek:
  - The Module Instance asks its Container to move PINFO'index forward or backward by a specified number of bytes relative to a starting position.
  - The starting position is one of 'start of PINFO', 'current index', 'size of PINFO'.
  - The Container moves PINFO'index and returns its new position to the Module Instance.
  - The Container ensures that the new PINFO'index position is greater than or equal to zero.
  - The Container ensures that PINFO'size is not exceeded.

For a given Module Instance and for a given PINFO, the PINFO API available in the Container depends upon the declaration of PINFO Access Scope, as defined in §7.10.

The ECOA Infrastructure sets the PINFO'index for a Module Instance to zero just prior to it invoking an INITIALIZE or REINITIALIZE Module operation.

7.11 Driver Components

Driver Components provide a means of allowing interactions between the ECOA and non-ECOA domains. This interaction may be with hardware devices or with non-ECOA software. As a result, driver components may be less portable than other ECOA components.

Any Driver Component must comply with any specified provided and required Quality of Service.

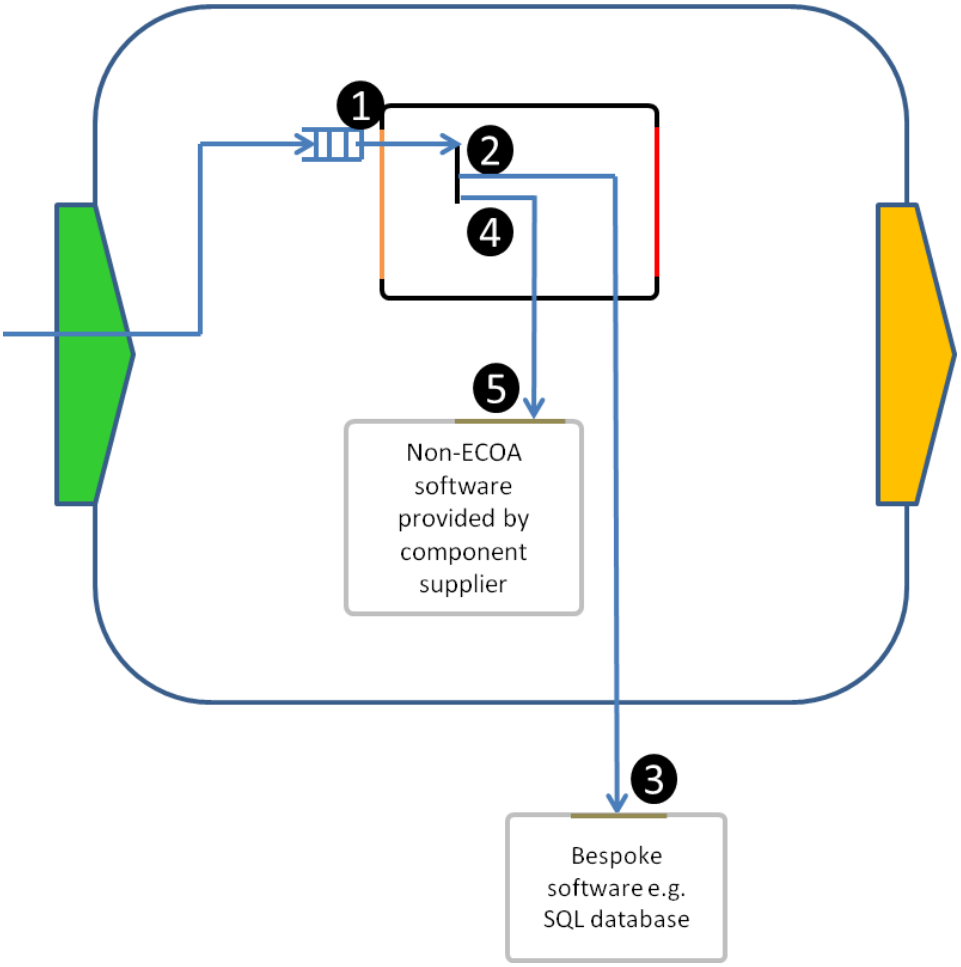

Figure 16 Driver Component Example – Interaction with Bespoke APIs

Figure 16 shows an example Driver Component whose module interacts with non-ECOA software through two bespoke APIs.

At **point 1** a Module Operation is invoked on the Client Module Instance from a Service Operation.

During this execution, the Module Instance invokes some non-ECOA functionality via a bespoke API at **point 2**.

The non-ECOA functionality is implemented by some bespoke software at **point 3** (e.g. an SQL database), and performs some functionality prior to returning control.

The Module Instance continues operation and invokes some alternate non-ECOA functionality via a second bespoke API at **point 4**.

This alternate functionality is implemented at **point 5** by some non-ECOA software provided by the component supplier. This non-ECOA software may interact with other software or hardware as required by the component provider. Once complete, control is returned back to the Module Instance.

Any non-ECOA software invoked by an ECOA Driver Component must ensure that it also complies with the ECOA Inversion of Control principle, as it is effectively part of the Module Instance thread of control. If any

bespoke software requires an interaction that would block the Module Instance from continung, then this must be implemented by a separate thread of control.

Within a Driver Component, an ECOA Module Instance can make use of a bespoke API to invoke non-ECOA software as described above. An external interface is defined [Architecture Specification Part 4] to allow non-ECOA software to asynchronously interact with one or more ECOA Module Instances or ECOA dynamic Trigger Instances. The externally generated event may be connected to one or more Module Instance operations and/or Dynamic Trigger instance 'event' operations following the normal rules of Event Links. This allows an externally generated event to be delivered to multiple Module Instances and/or Dynamic Trigger Instances within the Component Instance. An external interface may not be connected to a Provided or Required Service Operation.

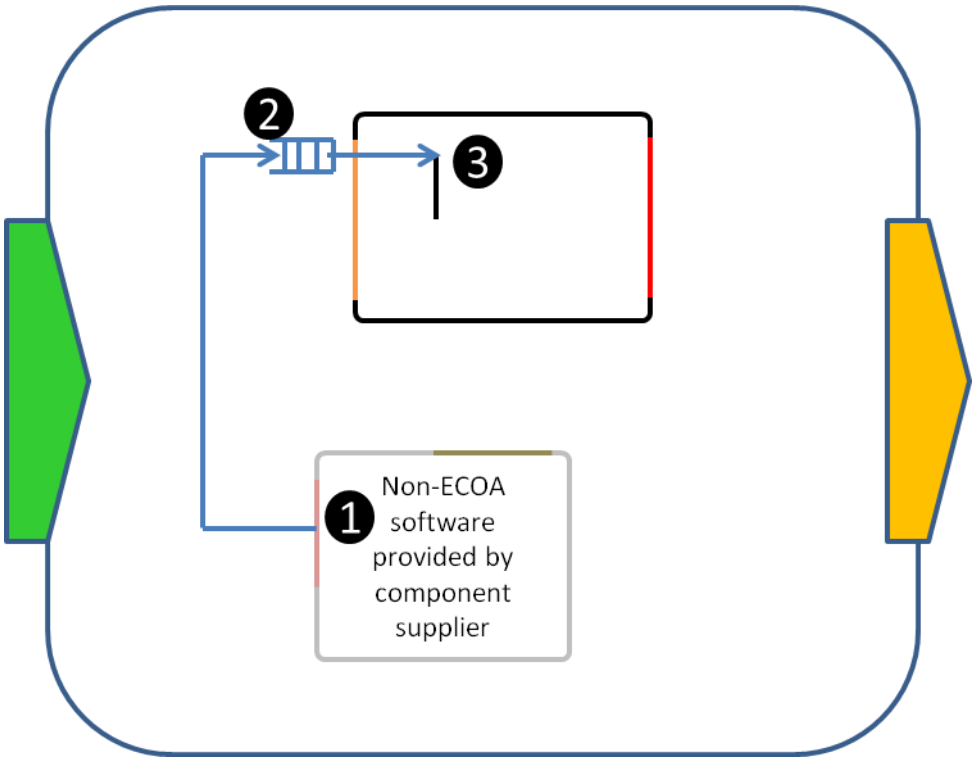

Figure 17 Driver Component Example – External Asynchronous Interaction

Figure 17 shows an example of how non-ECOA software asynchronously interacts with one Module Operation of a single Module Instance through a defined external interface (**point 1**).

When the ECOA external interface (implemented by the container) is invoked by the non-ECOA software at **point 1**, an event is placed into the Module Instance queue at **point 2**. The event will be processed as usual, and the associated operation will be invoked on the Module Instance at **point 3**. The Module Instance can then perform the appropriate processing for the received event.

Due to the chosen approach, only one single instance of one given driver component implementation can be deployed per protection domain; this limitation does not preclude the deployment of several driver component instances within the same protection domain if they refer to distinct driver component implementations. Any Module Instances of the same ASC Instance that are linked to the same external operation must be deployed in the same Protection Domain along with the non-ECOA software that invokes the associated External API.

8 ECOA System Management

8.1 Lifecycle

A Component Instance is composed of Module Instances. The Runtime Lifecycle of a Module Instance defines its runtime state.

Figure 18 illustrates the Module Runtime Lifecycle.

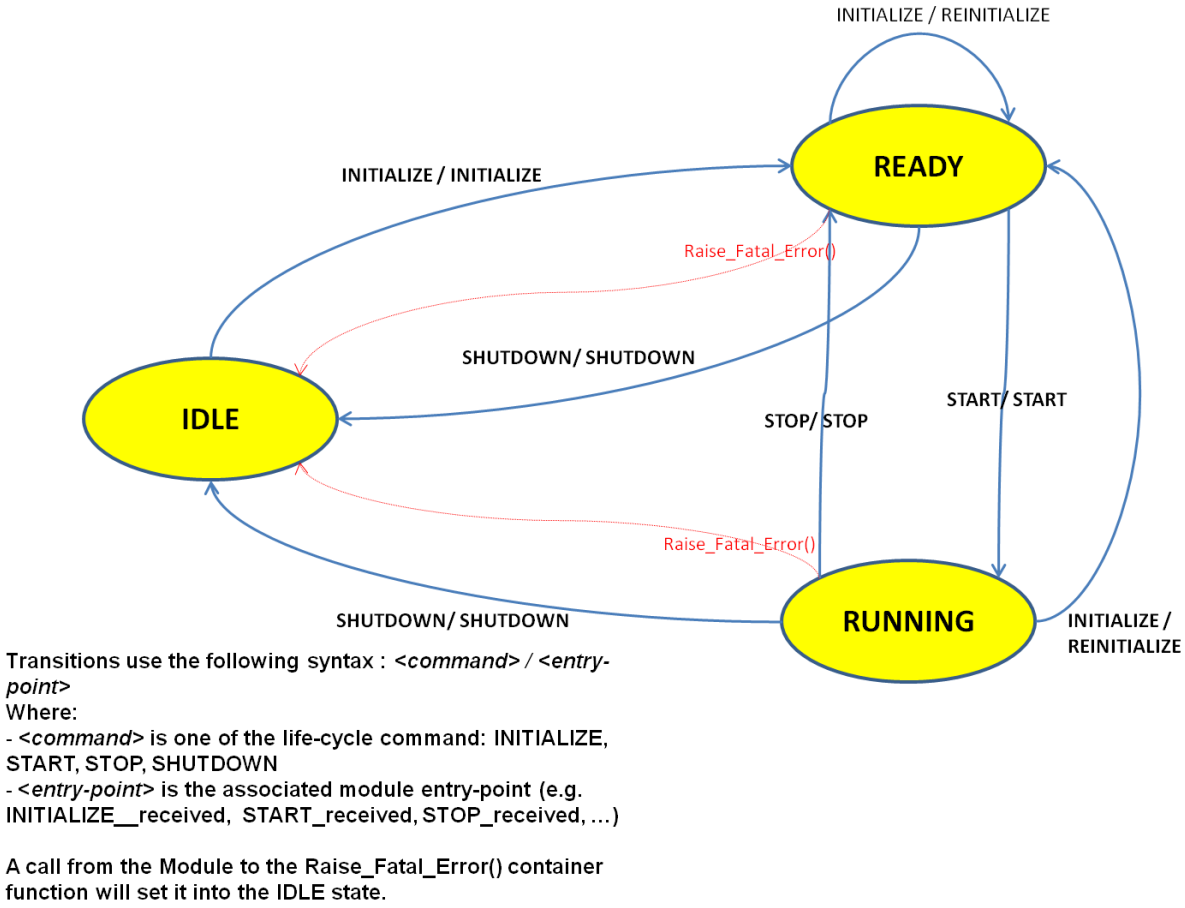

Figure 18 Module Runtime Lifecycle

A Module Instance has three possible logical states:

- IDLE – state reached after instantiation, fatal error raising or shutdown.
- READY – state reached when the module has been initialized.
- RUNNING – state where the Module Instance is handling incoming Module Operations. In the RUNNING state, the Module Instance may be blocked when invoking Container Interface blocking operations such as synchronous request-responses.

A Module Instance can transition between these states as shown in Figure 18. The states and transitions are managed by the Container, which invokes Module Interface entry points for each state change. The Module Instance states do not necessary reflect the states of the underlying OS tasks which support the Module Instances, e.g. a Module Instance may be in a RUNNING state while the underlying task may be in a ready state waiting for the CPU resource.

The Module Interface contains entry points that are invoked by the Container as a result of a Module Lifecycle state change. They are:

- INITIALIZE
- REINITIALIZE
- START
- STOP
- SHUTDOWN

NOTE For an INITIALIZE command, the Container will invoke the INITIALIZE entry point if the Module is in the IDLE state and the REINIITIALIZE entry point if it is in the READY or RUNNING state.

The lifecycle of non-Supervision Module Instances within a Component instance are managed by the Supervision Module with the assistance of the Container:

In order to achieve this, the Supervision Module Container Interface provides the following operations (one set per non-Supervision Module Instance):

- INITIALIZE
- START
- STOP
- SHUTDOWN

8.1.1 Module Startup

Upon start-up of the ECOA System, the ECOA Infrastructure is responsible for the allocation and initialization all the resources (threads, libraries, objects, etc.) needed to execute the functionality of the Module Instances and their Containers.

Following allocation and initialisation of resources each Module Instance is brought to the **IDLE** state.

8.1.2 Supervision Module Startup

When all Module Instances have been brought to the **IDLE** state, the Container commands all Supervision Module Instances to **INITIALIZE** and **START** by performing the following:

- the Container calls the **INITIALIZE** entry point in the Supervision Module Interface
- when the entry point returns, the Container changes the state of the Supervision Module Instance from **IDLE** to **READY**, and calls the **START** entry point in the Supervision Module Interface.
- when the entry point returns, the Container changes the state of the Supervision Module Instance from **READY** to **RUNNING**.

NOTE: Although the Supervision Module is able to manage non-Supervision Modules at any time; it is recommended that it only manages them in the **RUNNING** state (i.e. during or after the **START** entry point).

8.1.3 Non-Supervision Module Startup

Within a Component, the Supervision Module can Request non-Supervision Module Instances to **INITIALIZE** (or **REINITIALIZE** depending on the state of the Module Instance). The Module Instance then performs any actions required to initialize, or re-initialize, such as allocating further resources, setting its internal variables and/or reading its Properties to reach a functionally coherent and initialized internal state.

Following initialisation the Module Instance is **READY** and the Supervision Module can Request it to **START**. At this point, the Module Instance has the opportunity (via its **START** operation) to perform any actions relevant to its transition to the **RUNNING** state (these are likely to be specific to the functionality of the design.). Once started the Module Instance enters the **RUNNING** state.

The following are the steps taken to change the state of a Module Instance:

- the Supervision Module Requests a Module Instance lifecycle state change via the Container Interface
- the Container invokes the appropriate entry point in the non-Supervision Module Interface
- when the entry point returns the Container changes the state of the Module Instance
- the Container then notifies the Supervision Module – indicating the state change is complete.

8.1.4 Module Run-time Behaviour

Lifecycle Events that do not represent a valid transition from the current state, as shown by the Module Lifecycle state diagram in Figure 18, are discarded, and the fault management Infrastructure will be notified (see section 8.3).

Lifecycle Events are activating operations.

Lifecycle Events have no priority over other operations:

- On receipt of **STOP**, **SHUTDOWN** or **INITIALIZE** Events, operations already executing are allowed to complete and operation calls already queued will be executed.
- Operations arriving when a Module Instance is entering the **RUNNING** state will be queued and executed after the **START** entry-point has returned.

A Module cannot invoke any Request-Response operations in its Container Interface as part of the implementation of a Module Lifecycle operation. The Module may however invoke Event or Versioned Data operations in this case.

8.1.5 Module Shutdown

When its **SHUTDOWN** entry point is called the Module Instance may de-allocate any associated resources (those previously allocated during **INITIALIZE**).

When its **SHUTDOWN** entry point returns the Infrastructure de-allocates the resources used by the Module Instance, after which the Module Instance enters the **IDLE** state.

Whenever a Module Instance enters the **IDLE** state, its queue is cleared and the Infrastructure releases all Versioned Data handles associated with it, including those involved in pending update notifications.

8.1.6 Module Runtime Lifecycle Example

Figure 19 provides an example of how the Module lifecycle can be used in a Component by a Supervision Module to manage other modules

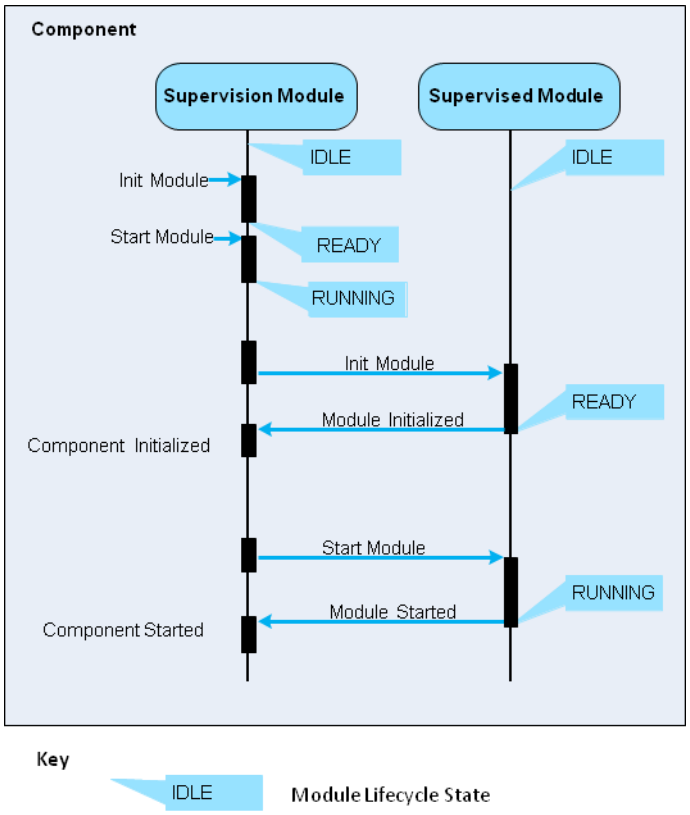

Figure 19 Lifecycle Example

Here the Supervision Module, once it has been initialized and started by the platform, automatically initializes and starts the other module in the Component. At each step the Supervision Module is notified of the Supervised Module state transition to ensure that it correctly sequences the commands.

Once the Supervised Module is Initialized, the component can be thought of as 'initialized', and once the Supervised Module is Running, the Component can be thought of as being in a 'running' state.

This is a very simple example, and more complex components may not automatically initialize and start all Supervised Modules immediately. This could be related to some specific management policy for a component, or may be related to a hierarchical component lifecycle defined by the system being developed.

Each Application Software Component manages internally its own state based on its own Module Runtime Lifecycle states. By default, this internal component-level state is not shared with other components.

8.2 Health Monitoring

Unlike Fault Handling, ECOA assumes that Health Monitoring is related to application and functional system design and that it can be addressed by designing ASCs for this purpose.

8.3 Fault Handling

8.3.1 Error Categorization

Error is a global term that encompases:

- Application errors: consequences of faults occurring at application Module or Supervision Module level. Their management is the responsibility of the component provider. An Error can be:
  - A fatal Error: the Module knows it cannot recover on its own and uses mechanisms provided by the Infrastructure to report the Error
  - A non-fatal Error: the Module may be able to recover on its own (in case of minor Errors) or it may report the Error
- Infrastructure errors: consequences of faults occurring at ECOA Infrastructure level, i.e. Module Container level, Protection Domain level, Computing Node level or Computing Platform level, or faults raised by a Supervision Module. Their management is the responsibility of the system architect and system integrator.

8.3.2 Error Propagation and Recovery Actions

Errors can be detected at several levels:

- Module level (application errors)
- Module Container level (Infrastructure errors)
- Protection Domain level (Infrastructure errors)
- Computing Node level (Infrastructure errors)
- Computing Platform level (Infrastructure errors)

Depending on the nature of the error (application error or Infrastructure error) and the level of detection, fault management may provide recovery procedures at:

- Module level
- Component level
- Protection Domain level
- Computing Node level
- Computing Platform level

The recovery procedures for Infrastructure errors, provided by the ECOA Infrastructure, are done by an entity named Fault Handler. ECOA Fault Handling at infrastructure level may be implemented into one or several such entities on a single platform. The scope of the ECOA Fault Handler(s) cannot be wider than platform level (e.g. in a system made of two platforms connected through ELI, there shall be at least one ECOA Fault Handler entity per platform).

8.3.2.1 Application Errors Propagation and Recovery Actions

Application errors may be detected by an application Module or a Supervision Module. The Component provider determines the recovery actions (if any) to be applied by an application Module, or by a Supervision Module when an application error occurs.

If an application error is detected by an application Module, the recovery procedure may be:

- In case of non fatal application error, the Module may
  - Handle the application error if it has been coded to recover from that type of error,
  - Raise the application error to its Supervision Module through *raise_error*.
- In case of fatal application error, the application Module uses the *raise_fatal_error* API. The error is automatically sent to the Supervision Module and the application Module is immediately placed into the IDLE state by the ECOA middleware without the associated lifecycle notification being sent to the

Supervision Module. The Supervision Module may then (as an example) change the availability of one or more services of the Component (related to the faulty Module).

Figure 20 illustrates the propagation of Non-fatal application errors.

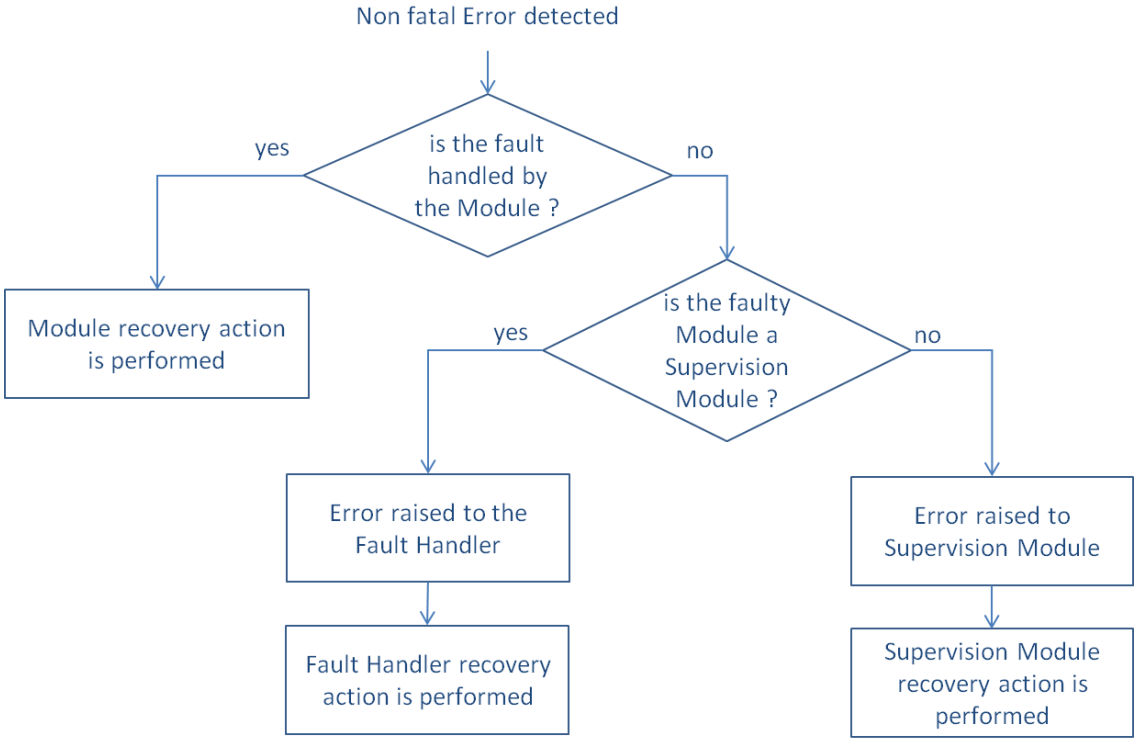

Figure 20 Propagation path of non-fatal application error

When an application error is detected by a Supervision Module or received from an application Module, the procedure may be:

- Handle the application error, where the recovery actions may be:
  - Shutdown the faulty Module or all the Modules of the faulty Component (using the normal Module Lifecycle operations)
  - Restart the faulty Module or all the Modules of the faulty Component (using the normal Module Lifecycle operations)
  - Change the availability of one or more Component Services
- The Supervision Module cannot handle the application error and raises it to the Fault Handler through *raise_error* , the Fault Handler will determine the Recovery Action
- The Supervision Module is the faulty Module, or it cannot handle the application error and raises it to the Fault Handler through *raise_fatal_error*:
  - The ECOA middleware sets the Component services as unavailable and immediately places the Supervision Module and all its application Modules into the IDLE state without the associated lifecycle notification being sent to the Supervision Module.
  - The Fault Handler will determine the Recovery Action

Figure 21 illustrates the propagation of fatal application errors.

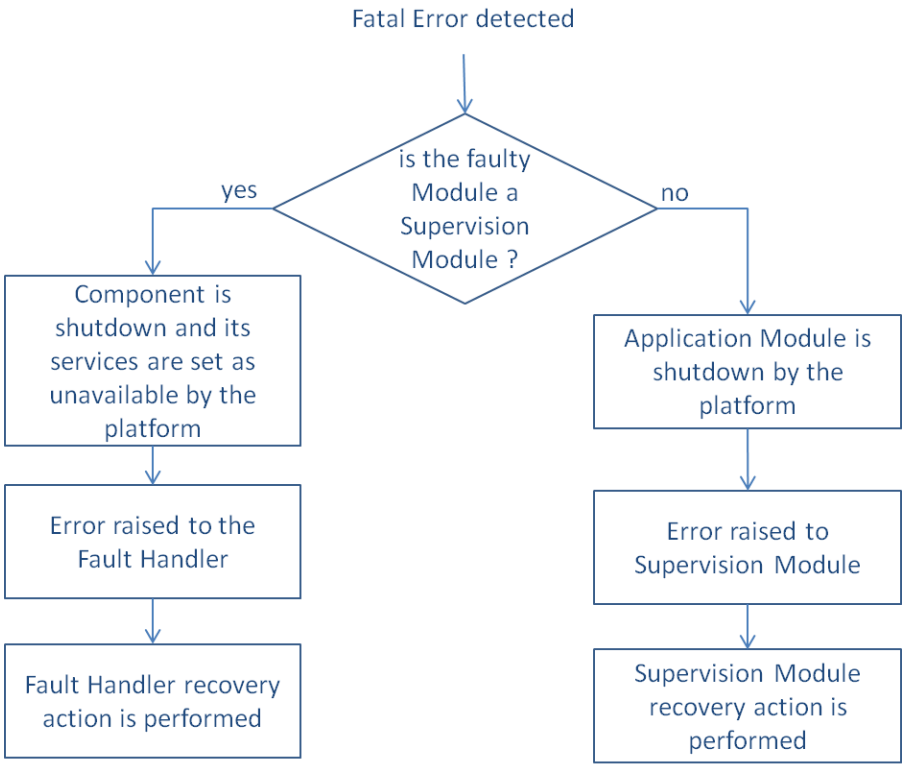

Figure 21 Propagation path of fatal application errors

8.3.2.2 Infrastructure Errors propagation and Recovery Actions

An Infrastructure error is a fault detected at Infrastructure level, i.e. at Module Container level, Protection Domain level, Computing Node level, Computing Platform level or raised by a Supervision Module. Figure 22 illustrates Infrastructure error propagation:

- If the Infrastructure error involves the Fault Handler (itself or its Protection Domain or its Computing Node):
  - If there is only one Fault Handler instance on the ECOA platform, it is handled by the ECOA platform that applies a default recovery action
  - Otherwise it may be handled by another Fault Handler (or, still, by the ECOA platform) , depending on the implementation chosen by the platform supplier
- Otherwise:
  - If the Infrastructure error is detected by a Module Container, it is raised to the Fault Handler
  - If the Infrastructure error comes from a Protection Domain, a Computing Node, or another Computing Platform, it is directly detected by the Fault Handler

  Then, the Fault Handler logs the fault and applies the corresponding recovery procedure.

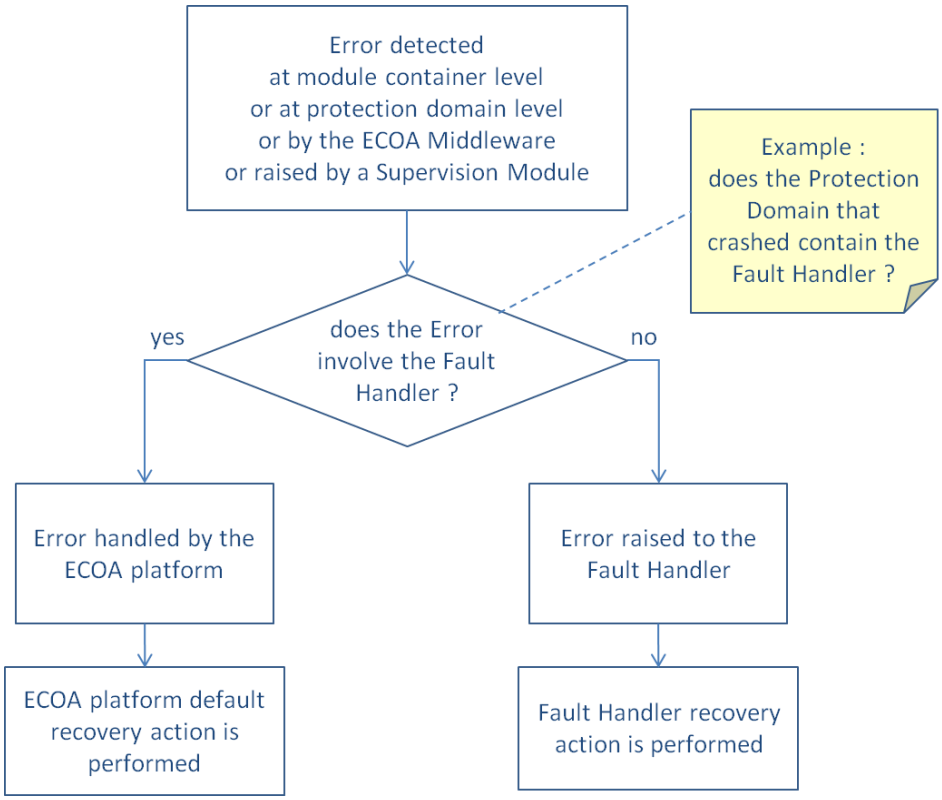

Figure 22 Propagation path of infrastructure errors

The parameterization of the Fault Handler recovery routines (i.e. the recovery actions to be applied) is the responsibility of the system architect and the system integrator. Recovery actions for Infrastructure errors may be applied at some levels:

- At Component level, the Fault Handler may:
  - Shutdown the faulty Component
  - Warm/cold restart the faulty Component
- At Protection Domain level, the Fault Handler may:
  - Shutdown the Protection Domain
  - Warm/cold restart the Protection Domain
- At Computing Node level, the Fault Handler may:
  - Shutdown the Computing Node
  - Warm/cold restart the Computing Node
- At Computing Platform level, the Fault Handler may:
  - Shutdown the Computing Platform
  - Warm/cold restart the Computing Platform
  - Change the deployment

The component shutdown action consists in the Infrastructure putting all Module Instances of the targeted Component Instance in idle state (with the Supervision Module Instance being the first). The Infrastructure frees associated resources, without calling the shutdown entry point of these potentially faulty Module Instances. Therefore, this is not a graceful functional shutdown and is not part of the Module Lifecycle.

The component restart action consists in the Infrastructure of triggering a component shutdown action, followed by initialize and start lifecycle actions targeted at the Supervision Module Instance of the Component Instance.

In order to allow fault filtering, the Fault Handler has a context that can be used as a memory to store the number of occurrences for each fault on a slipping time frame. As an example, it allows to distinguish a sporadic fault from a recurrent fault, and then to apply recovery action only if a given fault occurs more than X times in the given time frame.

Figure 23 summarizes the error propagation path and applicable recovery actions.

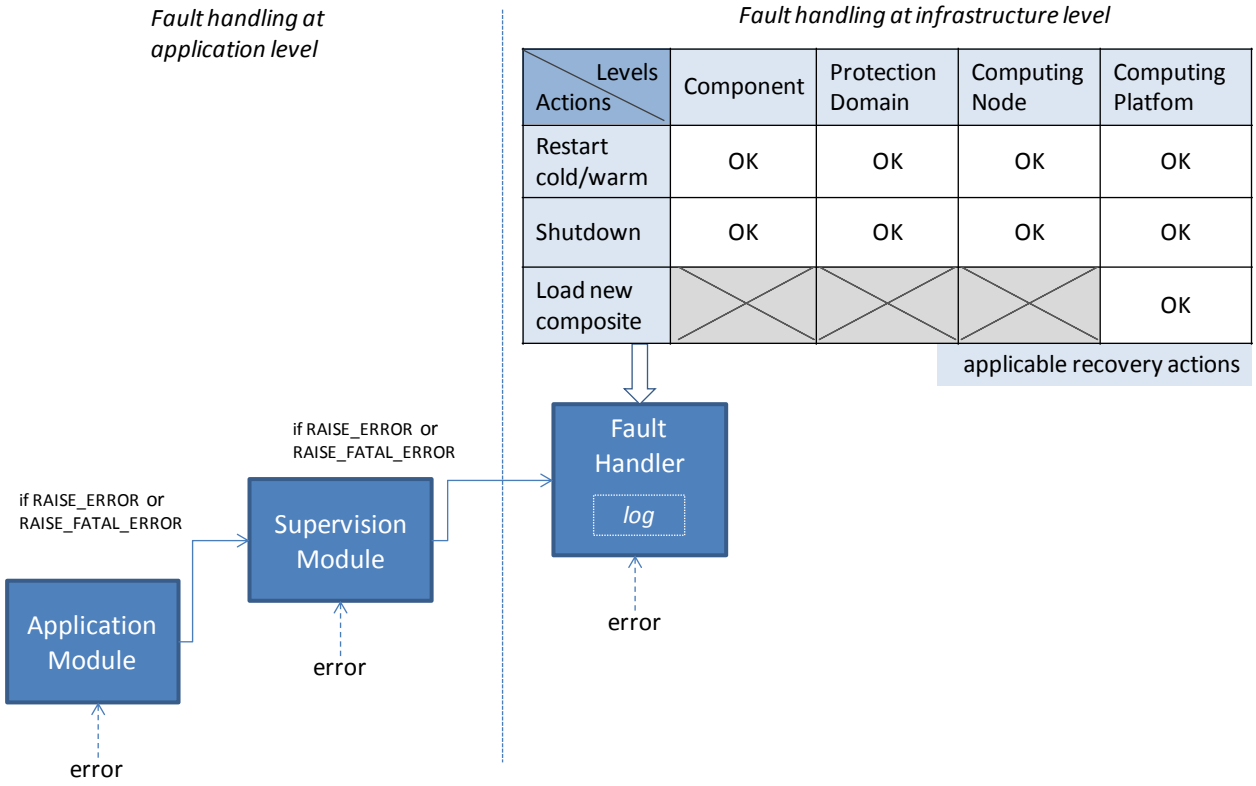

Figure 23 Error propagation path

8.3.3 Operations and faults

ECOA allows the component supplier to use three types of operations in his application modules:

- Event
- Versioned Data
- Synchronous and Asynchronous Request/Response

Each of these operations has specific behaviours regarding identified errors and infrastructure errors.

Event:

- No error code is returned to the Module, due to fire-and-forget schema
- Infrastructure errors handled by the Fault Handler:
  - RESOURCE_NOT_AVAILABLE client container unable to send the event

- OPERATION_NOT_AVAILABLE server cannot handle the event
- UNKNOWN_OPERATION requested operation ID is invalid

Figure 24 illustrates these behaviours.

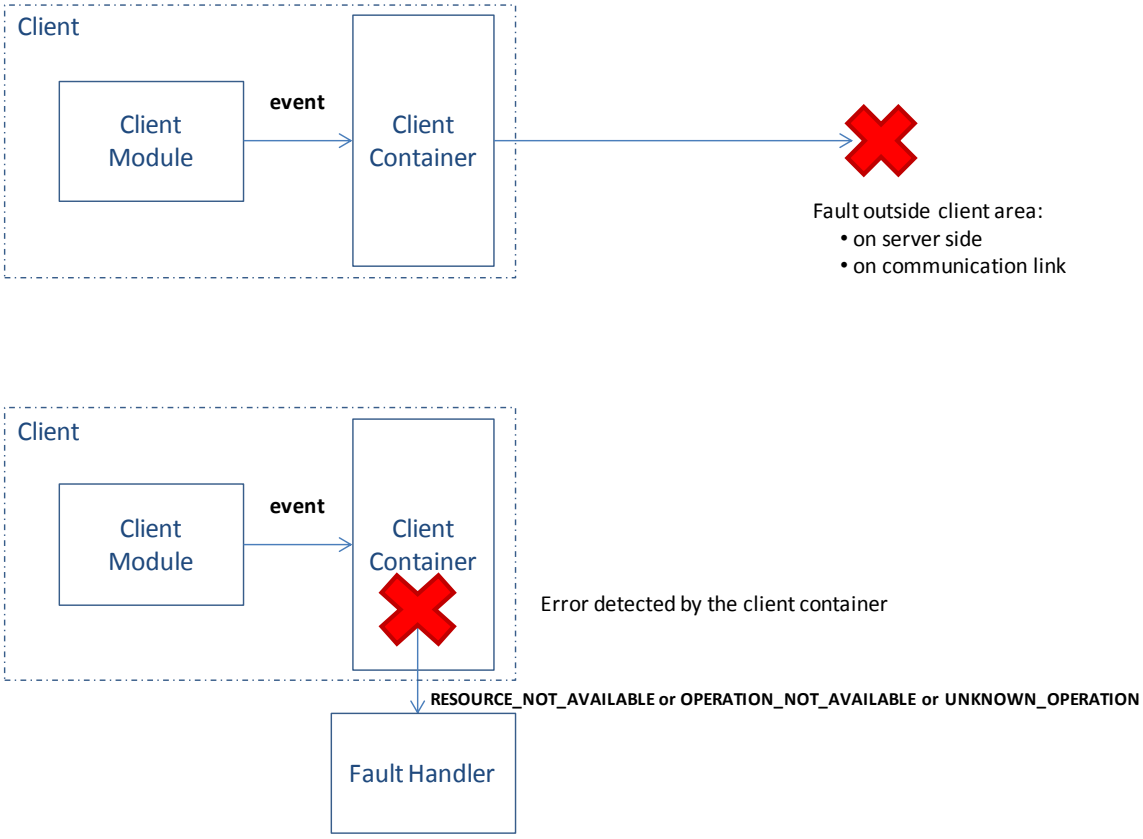

Figure 24 Event faults propagation behaviour

Request-Response:

- Request faults:
  - Error codes returned to the Module:
    - OK no error
    - NO_RESPONSE QoS timeout
  - Infrastructure errors handled by the Fault Handler:
    - RESOURCE_NOT_AVAILABLE client container unable to send the request
    - OPERATION_NOT_AVAILABLE server cannot handle the request
    - UNKNOWN_OPERATION requested operation ID is invalid
- Response faults:

- Error codes returned to the Module:
  - OK — no error
  - INVALID_IDENTIFIER — API called with invalid request-response ID
- Infrastructure errors handled by the Fault Handler:
  - RESOURCE_NOT_AVAILABLE — server container unable to send the response
  - OPERATION_NOT_AVAILABLE — client cannot handle the response
  - OVERFLOW — server container unable to retain the request (maxConcurrentRequests has been reached)

The Figure 25 and Figure 26 illustrate these behaviours.

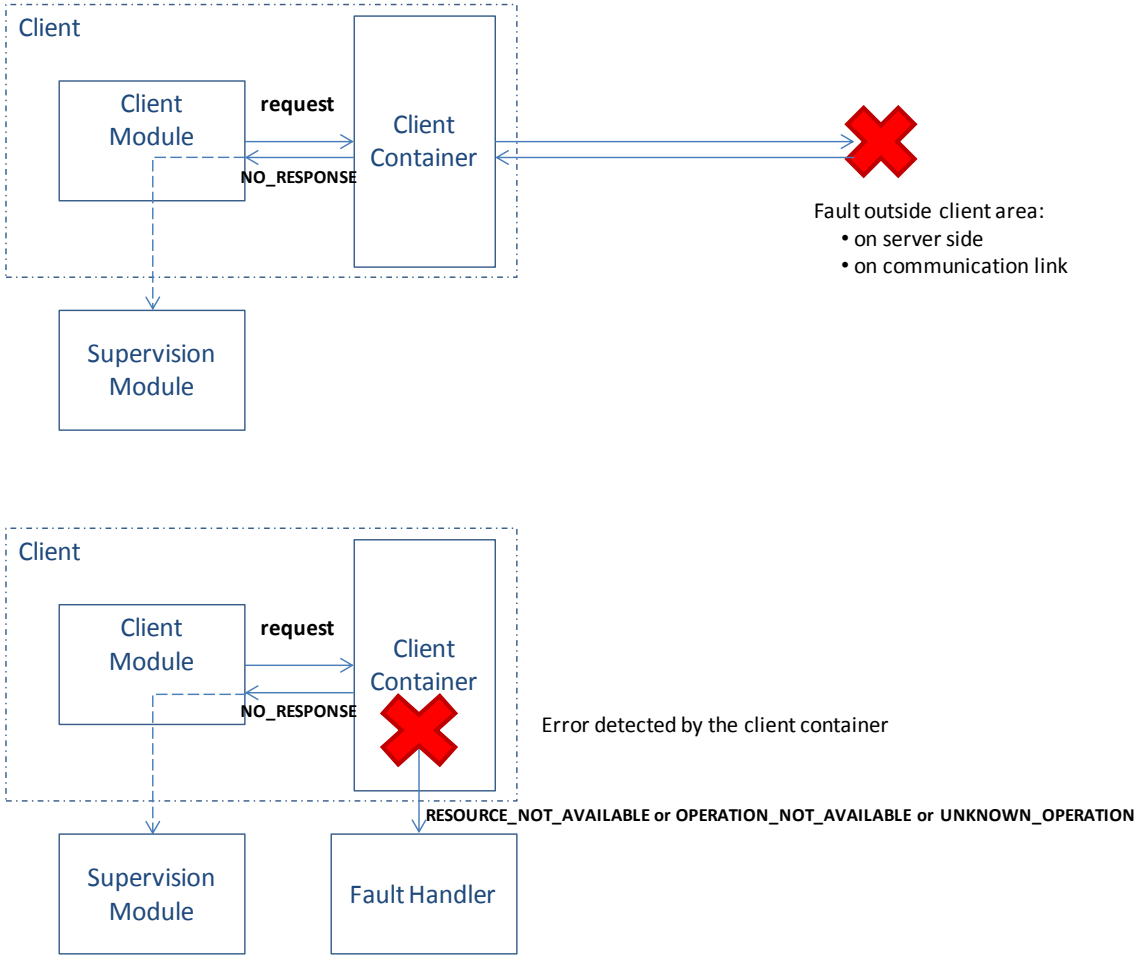

Figure 25 Request-Response faults propagation behaviour part 1

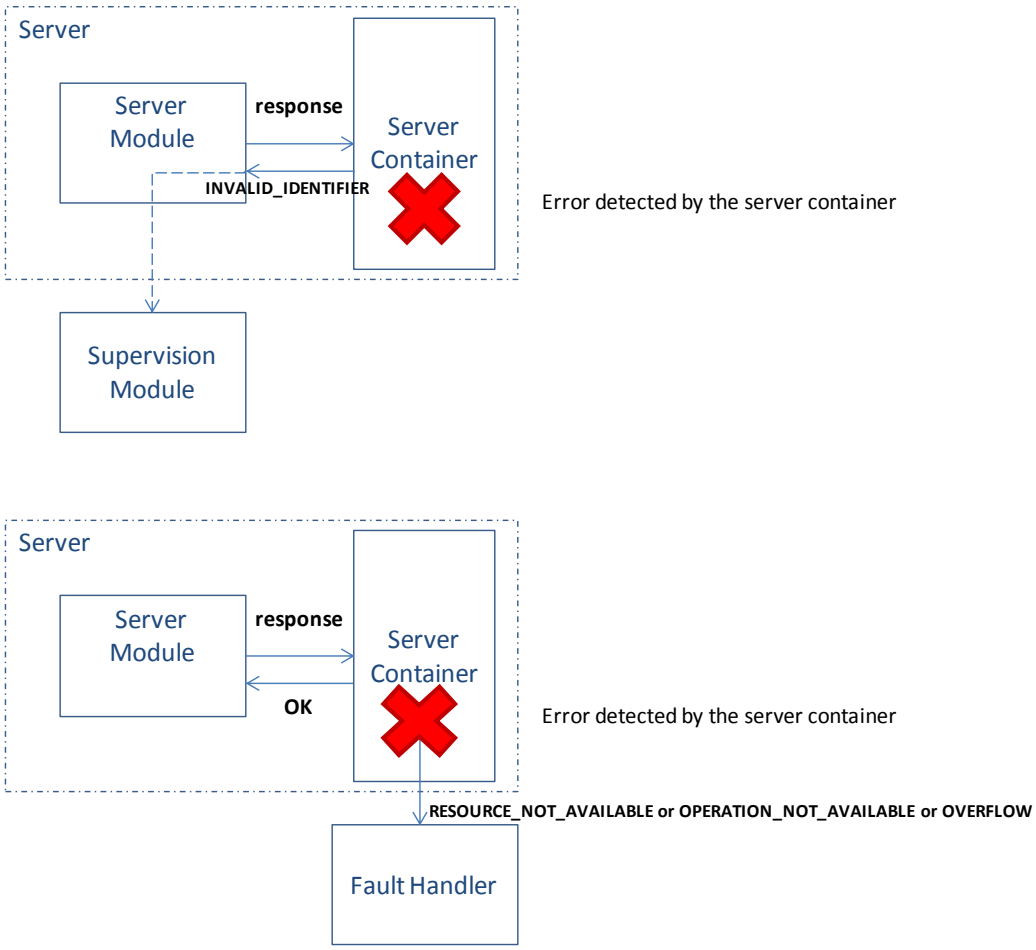

Figure 26 Request-Response faults propagation behaviour part 2

Note that in case of NO_RESPONSE returned to the Module due to an Infrastructure error detected by the client Container after a request call, the Infrastructure error is handled by the Fault Handler. At the same time, the Module has the possibility to raise an error to its Supervision Module, which potentially may raise it to the Fault Handler too, causing two recovery actions for a same original fault.

Note also that when a fault is detected outside of the client area after a request call, the platform supplier has the possibility to choose between two implementations: he can choose to send NO_RESPONSE to the client though the ECOA middleware, or do nothing and wait for the timeout to happen on the client side (which causes the client Container to send a NO_RESPONSE to the client Module).

Versioned Data:

- Get_read_access:
  - Error codes returned to the Module:
    - OK no error
    - INVALID_HANDLE API called with a invalid versioned data handle (e.g. null pointer)
    - NO_DATA not an error ; the data has never been written (reader side)

- Infrastructure error handled by the Fault Handler:
  - RESOURCE_NOT_AVAILABLE client container unable to access the memory slot

- Release_read_access
  - Error codes returned to the Module:
    - OK no error
    - INVALID_HANDLE API called with an invalid versioned data handle (e.g. null pointer)
  - Infrastructure error handled by the Fault Handler:
    - RESOURCE_NOT_AVAILABLE client container unable to access the memory slot

- Get_write_access:
  - Error codes returned to the Module:
    - OK no error
    - INVALID_HANDLE API called with an invalid versioned data handle (e.g. null pointer)
    - DATA_NOT_INITIALIZED not an error ; the data has never been written (writer side)
  - Infrastructure error handled by the Fault Handler:
    - RESOURCE_NOT_AVAILABLE client container unable to access the memory slot

- Publish_write_access:
  - Error codes returned to the Module:
    - OK no error
    - INVALID_HANDLE API called with an invalid versioned data handle (e.g. null pointer)
  - Infrastructure error handled by the Fault Handler
    - RESOURCE_NOT_AVAILABLE client container unable to access the memory slot

- Cancel_write_access:
  - Error codes returned to the Module:
    - OK no error
    - INVALID_HANDLE API called with an invalid versioned data handle (e.g. null pointer)
  - No Infrastructure error

The Figure 27 and Figure 28 illustrate these behaviours.

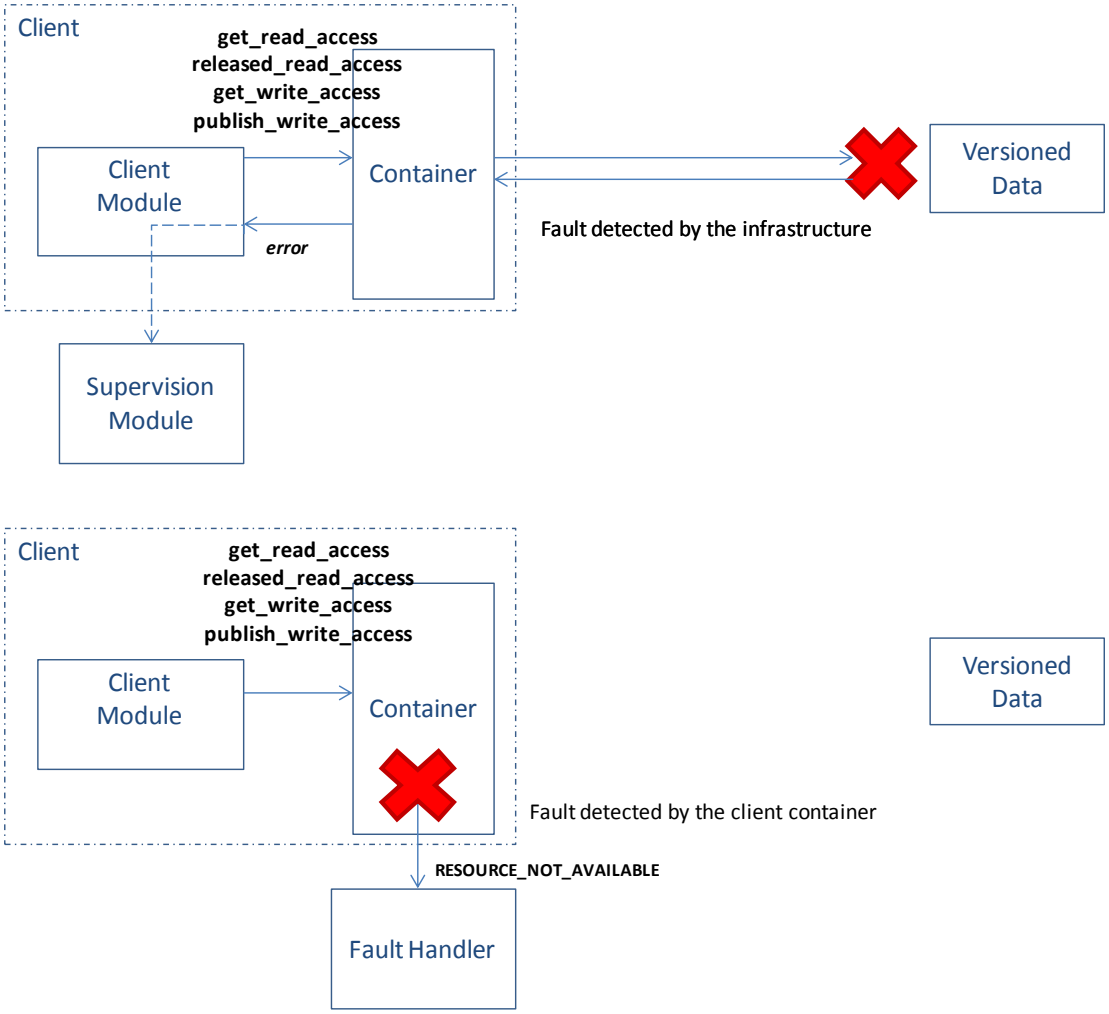

Figure 27 Versioned Data faults propagation behaviour part 1

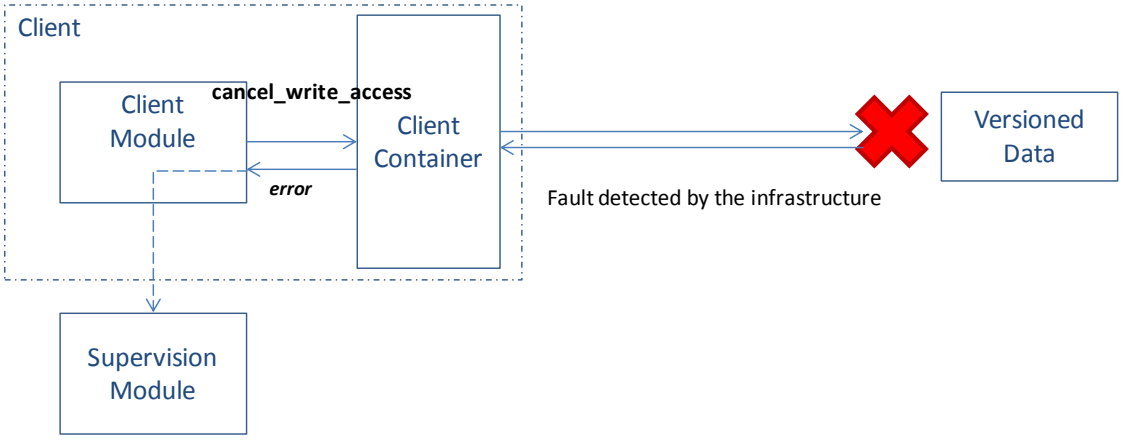

Figure 28 Versioned Data faults propagation behaviour part 2

8.4 Module Context

In order for a single module implementation to be instantiated multiple times, the module should not make use of "global" state; it should access instance specific data contained in the Module Context..

The Module Context contains two parts:

- User Context
- Warm Start Context

User Context is instance specific data defined and managed by the module implementation to allow it to maintain private state between each module operation invocation. This is analogous to the context of thread local storage in traditional operating system implementations.

User Context is zeroed by the infrastructure prior to an Initialize/Reinitialize Module Instance entry point is activated.

Warm Start Context is instance specific data defined and managed by the module implementation to allow it to maintain private state between transitions of Module Lifecycle states where the transition is an Initialize. The Module Implementation has access to a Container API so that it may request the Container to save the Warm Start Context in non-volatile storage such that it may be restored during a future Warm Restart. Warm Restart can be initiated by the Fault Handler, or when the Supervision Module restarts a Module Instance. The Infrastructure shall perform the save operation atomically. If the save operation fails no error is returned to the calling Module; however an error may be raised to the Fault Handler.

The calling module is responsible for determining the validity of the Warm Start Context. This may be achieved by incorporating additional validity entries in the Warm Start Context itself.

For a Cold Start from power up or a Cold Restart initiated by a Fault Handler the ‘saved’ value of the Module Warm Start Context is zeroed by the infrastructure.

In this way for a Warm Start initiated by a Fault Handler, the Module Warm Start Context is either restored from a previously zeroed version, or the most recently saved version.

Warm Start Context used by a Module Instance is zeroed/restored by the infrastructure prior to any activation of the Initialize/Reinitialize Module Instance entry point.

For restarts initiated by the physical platform, the Warm Start Module Context is non-volatile. Table 2 details the volatility of the User and Warm Start Module Context.

Table 2 User and Warm Start Module Context Volatility

User Context		Warm Start Context	
Warm	**Cold**	**Warm**	**Cold**
Volatile	Volatile	Non Volatile	Volatile

Figure 29 shows how the User Context and Warm Start Context are managed through these transitions.

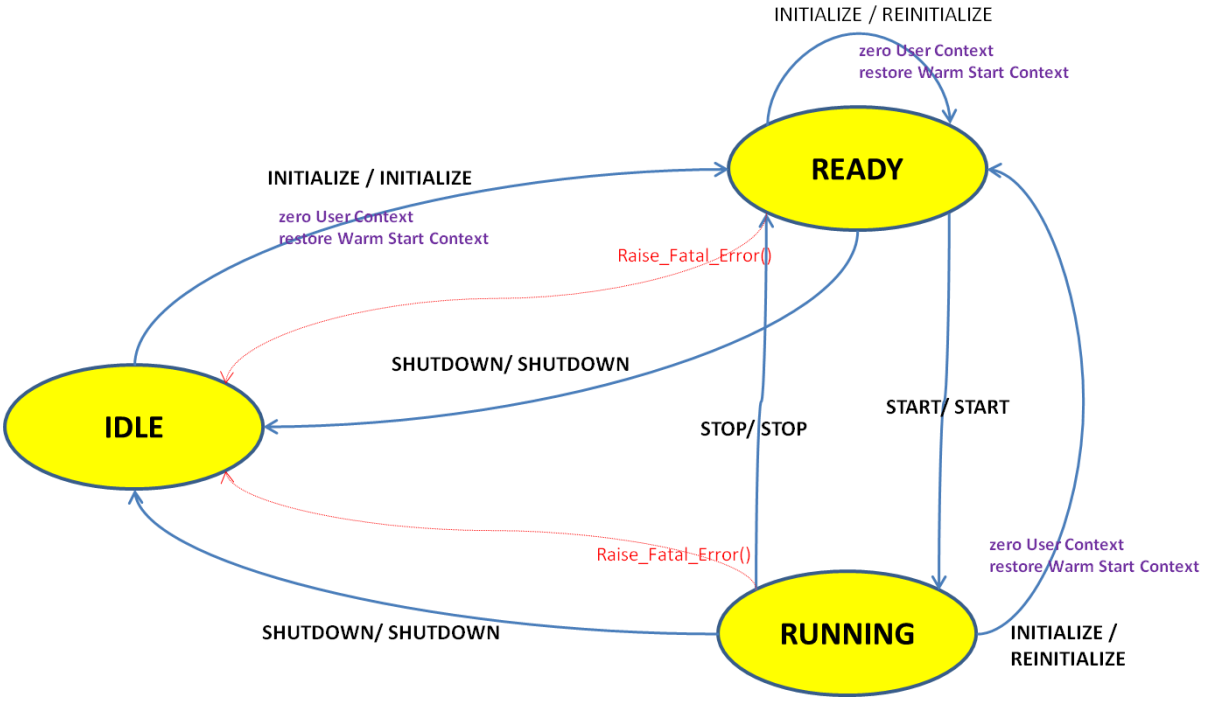

Figure 29 Management of Module Context

Architecture Specification Part 4 describes the detail of how the User Context and Warm Start Context are represented in various languages, and the language bindings specifies the detail of the implementation.

9 Scheduling

9.1 Scheduling Policy

Support for scheduling of deployed Module/Trigger Instances is provided by the underlying operating system. A deployed Module/Trigger Instance is single-threaded and will be assigned a priority by the System Integrator. The System Integrator shall take into account the relative priorities set by each Component Supplier on the Module/Trigger Instances declared in their Component Implementation.

Any ECOA platform shall respect the priorities defined by the System Integrator on deployed Module/Trigger Instances. This implies:

- Deploying Module/Trigger Instances onto OS tasks according to their priorities:
  - Module/Trigger Instances with different priorities shall not be deployed onto the same OS task,
  - Module/Trigger Instances with identical priorities may be deployed onto the same OS task,
  - OS task priorities shall be set with respect to the priority of the Module/Trigger Instances deployed onto them.
- Module pre-emption capability in ECOA platforms:
  - Any deployed Module/Trigger Instance becomes eligible for execution on the computing node resource whenever it receives an activating operation in its queue (including reception of the response of a synchronous request-response),
  - A Module/Trigger Instance currently executing on the computing node resource may be pre-empted by another eligible Module/Trigger Instance of a higher priority,

- A Module/Trigger Instance becomes ineligible for execution on the computing node resource when its queue does not contain any activating operation.

The rules defined above allow early verification of the system behaviour to take place at ECOA Module/Trigger Instance level, and provide assumptions to facilitate Component integration and reuse.

Scheduling of OS tasks (which Module/Trigger Instances are being deployed onto) is the responsibility of the Infrastructure and any scheduling policy supported by the OS/Middleware may be used as required by the System Integrator, provided that it respects the rules defined above. Scheduling analysis is outside the scope of ECOA; although it is anticipated that it would be carried out following existing, established methods. The type of scheduling analysis required will be dependent upon the chosen scheduling policy.

9.2 Activating and non-Activating Module Operations

By default, Module Operations are activating; the arrival of a new operation implies the execution of the associated entry-point as soon as the Module Instance is able to execute. This schema is an Event-driven programming model.

To disable this default behaviour, attributes are defined within the Component Implementation at `EventLink`, `RequestLink` and `DataLink` level:

- `activating` which is a boolean specifying the policy used by the Container to handle the operation:
  - when True (default value), the Container activates the associated entry-point as soon as possible.
  - when False, the operation is queued and remains pending. When an activating Module Operation arrives to the same Module Instance through another Module Operation Link, all pending Module Operations arrived before the activating one are then processed in FIFO order and executed as any other Module Operation in accordance with the priority of the Module Instance. If non activating Module Operations are queued while this processing is done, their processing is postponed until the arrival of a new activating Module Operation. It is envisaged that this type of mechanism could be used to implement a time-driven programming model, which may allow for easier schedule feasibility analysis.
- `fifoSize` which is an integer specifying the maximum number of pending operations of a single type at each receiving Container level. The `fifoSize` attribute is defined against an operation Link; therefore different operations can be specified to have different maximum queue sizes. When the maximum number is reached for a given operation, the receiving Container discards the new incoming Module Operations associated to the Link. For an incoming Request Response, the receiver Container sends back an error message to the sending Container in order to notify the Client of the failure.

However, this choice does not apply to Module Lifecycle Operations, which are necessarily activating operations (i.e.-e it is not possible to define them as being non-activating).

NOTE Activating on DataLink is only useful when associated to a notifying Versioned Data (attribute `notifying` set to `true`) (see §7.5.1).

10 Service Availability

The availability of Services provided by a Component instance can be set, on a per Service basis, by the Supervision Module Instance via the Container Interface. The availability of a provided Service is totally left under the responsibility of the Provider component: it is likely to be dependent upon a combination of component internal logic and the availability of any required Services necessary in order to successfully provide its Service.

The availability of Services may be affected by run-time errors that could cause Module Instances to be shutdown.

10.1 Initialisation

During initialisation the ECOA Software Platform sets all Services as unavailable, and will propagate the availability of Services as Component instances are initialized and started. The set of Services in the system are defined by the Assembly Schema (see section 10.2), which may or may not be available at any given time.

10.2 Assembly Schema

The possible connections between the provided and required Services of Application Software Components in an ECOA system are determined by the Assembly Schema. This provides details of the Service Links (called Wires in the Assembly Schema) between Services. Service discovery may be:

- Static: where there is a single Provider of a Service. The links between providers and requirers of such Services are statically pre-determined by the Assembly Schema and are established at system start up.
- Dynamic: where there is more than one Provider of the Service and which is the active Provider is determined at run-time, when the Service Request is made. All of the possible connections are statically defined in the Assembly Schema.

An ECOA system may contain a mix of static and dynamic connections between its Services.

10.2.1 Service Links and Ranks

The links between Services are described by the Wires in the Final Assembly Schema. Each Wire has a single Requirer of a Service (identified as a source in the schema), a single Provider of the Service (identified as a target in the schema), and a Rank.

There may be multiple Providers and Requirers of the same Service; the connections between them are determined by the Wires in the Assembly Schema. Where multiple Providers (targets) are connected to the same Requirer (source) in the case of Versioned Data or Request Response, each Wire that is connected must have a unique Rank, which is used to choose a single Provider. The Provider with an available service which is connected by a Wire with the lowest Rank value is chosen in preference to the others (i.e. Active Provider). The behaviour of Events is dependent upon whether the Service Link is specified to multicast Events. If multicast is not enabled, the Active Provider concept is used.

Further detail on Rank and Service Link behaviour can be found in section 11.

10.3 Dynamic Service Availability

The Active Provider for Events, Versioned Data and Request-Response may be decided dynamically at run-time from the possible connections defined in the Final Assembly Schema.

The current Provider may become unavailable due to the loss of the Supervision Module or due to internal decision which sets the provided service to unavailable. In the case of a Supervision Module loss, i.e. as soon the Supervision Module leaves the RUNNING state, all provided services are set automatically by the platform to unavailable and associated modules are shutdown.

Where the current Provider of a Service (the available Provider with the lowest Rank value) becomes unavailable, the ECOA Software Platform will arrange for the Provider connected by a Wire with the next lowest Rank value (if the Service is available) to be chosen as:

- the sender in the case of an Event Sent By Provider (unless multicast is enabled),
- the receiver in the case of an Event Received By Provider (unless multicast is enabled),
- the responder in the case of a Request-Response,
- the writer in the case of Versioned Data

Any change to the Provider of a Service is notified to the Requirer. If there is no available Provider the Requirer will be notified that the Service is no longer available.

Optionally, the Quality-of-Service provided by any Provider may be monitored at run-time to ensure it is within the required QoS of the Requirer. If this is not the case, then a fault may be generated and reported to the Fault Management. In addition an alternative Provider may be used if one is available.

11 Service Link Behaviour

11.1 Introduction

A Service Link connects Application Software Components together[1]. Each Service Link connects one Provided Service to one Required Service which refers to the same Service Definition (which consists of operations i.e. Events, Request Response and Versioned Data). This is shown in Figure 30.

Figure 30 Service Links

It is necessary to specify the behaviour of each operation across the Service Links. This is because it is different for each type of operation.

11.2 Active Provider Component

The term *Active Provider* is introduced to describe the Application Software Component selected by the Infrastructure according to the following policy: it's provided Service is set as available and its Service Link has the lowest value of Wire Rank. (The value of the Rank attribute can be computed according to the deployment, for example to reflect a notion of "bonding" between Requirer and Provider). Rank is expressed as a positive integer value, whereby a low integer value represents a high ranking Service Link.

[1] The wiring of Application Software Components through Service Links may lead to cyclic dependencies between them: the correctness and the consistency of the resulted Assembly Schema is the responsibility of the System Designer.

11.3 Summary of Behaviour

When a Service Definition includes an Event Sent By Provider operation, the Event (and its associated typed data) sent by any Provider is received by all Requirers linked to the Provider (the Event data is distributed from the Provider to many Requirers). If there are multiple providers, the requirer only receives Events from the Active Provider, unless the Service Link is specified to multicast Events.

Similarly, when a Service Definition includes an Event Received By Provider operation, the associated typed data sent by any Requirer is received by the Active Provider, unless the Service Link is specified to multicast Events.

When a Service Definition includes a Versioned Data operation, each providing Application Software Component that is linked to the Provided Service may supply that data. Application Software Components that are linked to the Required Service read the most recent value of the data provided by the Active Provider. When the Active Service Provider is changed to another one, Readers read the instance of the new Provider. The Containers hide the switch between instances.

For a Request-Response operation referenced in a Service Link, the Request from the Client is addressed (directed) to the Active Provider (Server). In other words, in the case of multiple eligible Providers, the Infrastructure will select one of the Providers to provide a Response.

These behaviours are summarized in Table 3.

Table 3 Behaviour across a Service Link

Operation		Provider	Requirer
Event Operations	*sent_by_provider*	The Event and its associated typed data is sent to all Requirers	Receives the Event and its associated typed data from the Active Provider selected from set of eligible Providers (Servers) as determined by the Infrastructure according to the Rank attribute[2].
	received_by_provider	Receives the Event and its associated typed data from all Requirers if the Provider is the Active Provider[3].	The Event and its associated typed data is sent to all Providers
Request-Response Operations		Receives Requests from all Requirers (Clients)	Active Provider selected from set of eligible Providers (Servers) as determined by the Infrastructure according to the Rank attribute.
Versioned Data Operations		Updates data for all Requirers (Readers)	Active Provider (Writer) selected by Infrastructure according to the Rank attribute[4].

The behaviour above is true when the associated service is defined as available. When the service is defined as unavailable, the following behaviour is then applied:

- Event
  - The event is discarded on server side in case of local communication
  - The event is discarded on server and client sides in case of ELI-based communications
- Request-Response
  - NO_RESPONSE returned on the client side as soon as possible

2 This behaviour is the default one when the flag AllEventsMulticasted is set to False (default value). When this flag is set to True, Requirers receive all events sent by the Providers for which the flag is set to True.

3 This behaviour is the default one when the flag AllEventsMulticasted is set to False (default value). When this flag is set to True, all Providers receive all events sent by the Requirers for which the flag is set to True.

4 Providers maintain their own instance of the data: when a Provider accesses the data, it gets the value it wrote previously.

- The request is discarded on server side
- The response is always returned when the request has been handled before the setting of the service to unavailable

- Versioned data
  - The versioned data is always accessible by the writer but the publish action does not actually push the data.
  - The platform is required to republish data when the service becomes available again
  - The versioned data is always accessible by the reader, but the versioned data may become stale.

11.4 Examples

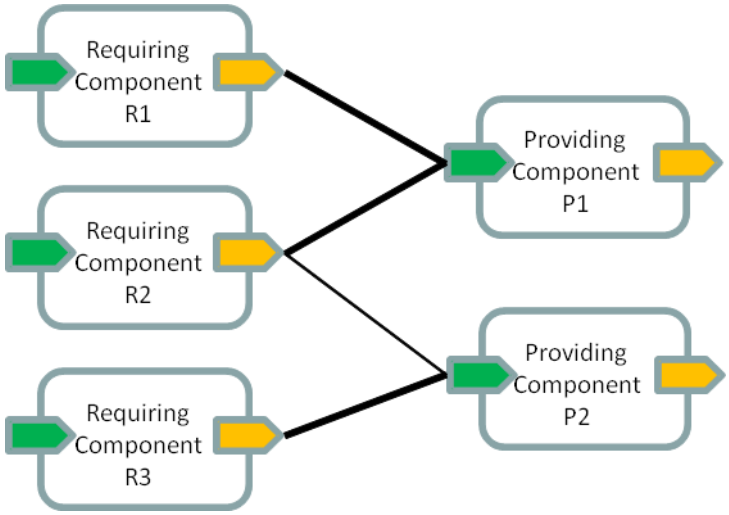

Figure 31 Example Assembly Schema

In the above Assembly Schema, three Application Software Components R1, R2 and R3 are connected, via Service Links, to two Application Software Components P1 and P2. P1 is considered as the Active Service Provider for R1 and R2, while P2 is considered as the Active Service Provider for R3.

"Sent by Provider" Events

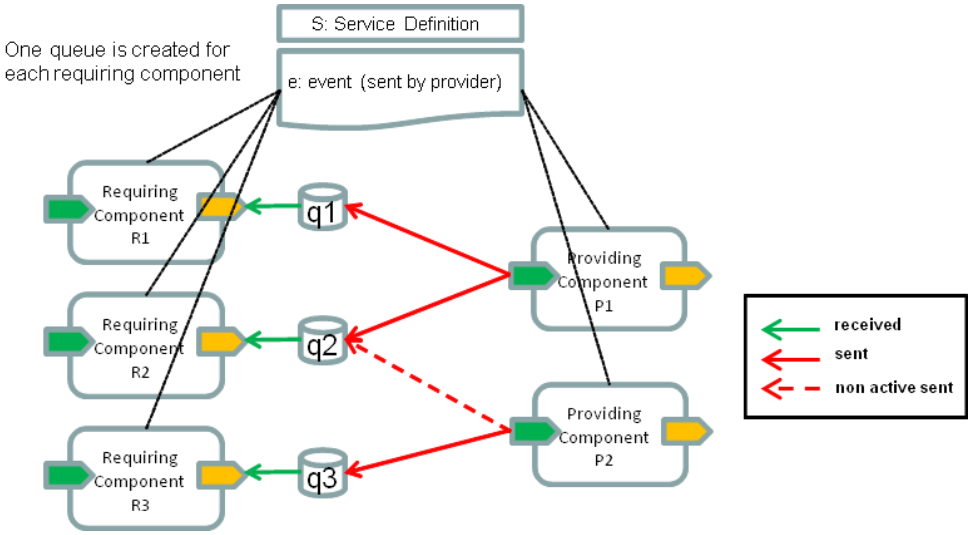

Figure 32 Generation of an Event

An Event e “Sent by Provider” in Service S translates to:

- An Event queue (q1,q2,q3) is created for each Requiring Component (R1, R2, R3);
- An Event sent by an Active Service Provider (P1 or P2) is received by all its Requirers (R1 and R2 for P1, R3 for P2).
- An Event sent by a non-active Provider is discarded by the Infrastructure except if the *allEventsMulticasted* flag is set to true. Therefore, an Event sent by P2 is only received by R2 if the *allEventsMulticasted* flag is set to true, otherwise, it does not reach R2.

"Received by Provider" Events

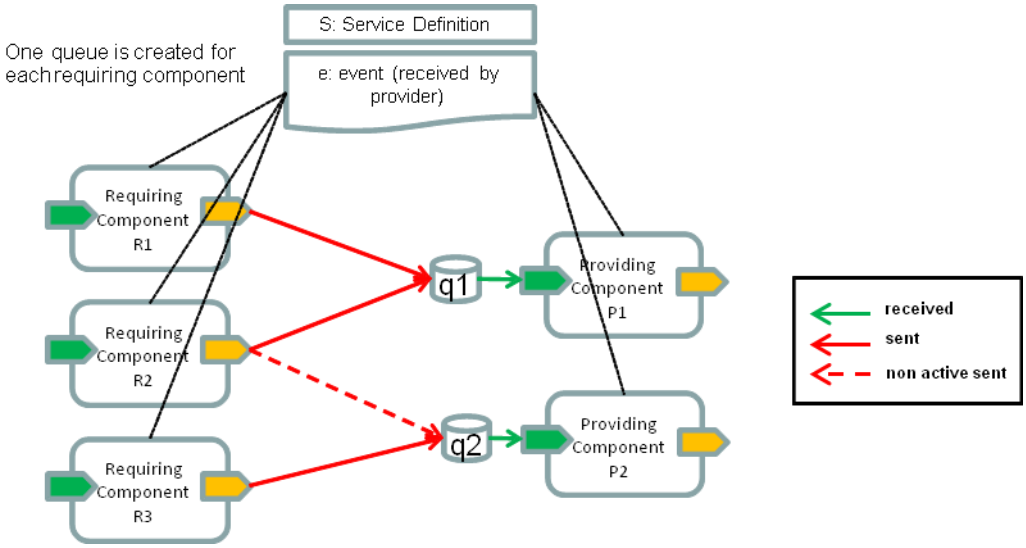

Figure 33 Consumption of an Event

An Event “received by Provider” in a Service translates to:

- An Event queue (q1,q2) is created for each providing Component (P1 and P2);
- An Event sent by a Requirer (R1, R2, R3) is received by the Active Service Provider (P1 for R1, P1 for R2, P2 for R3)
- An Event sent by a Requirer is received by its non-active Providers if the *allEventsMulticasted* flag is set to true (P2 for R2). Otherwise, the Event does not reach non-active Providers.
- All Requirers are allowed to send the Event.

Request-Response

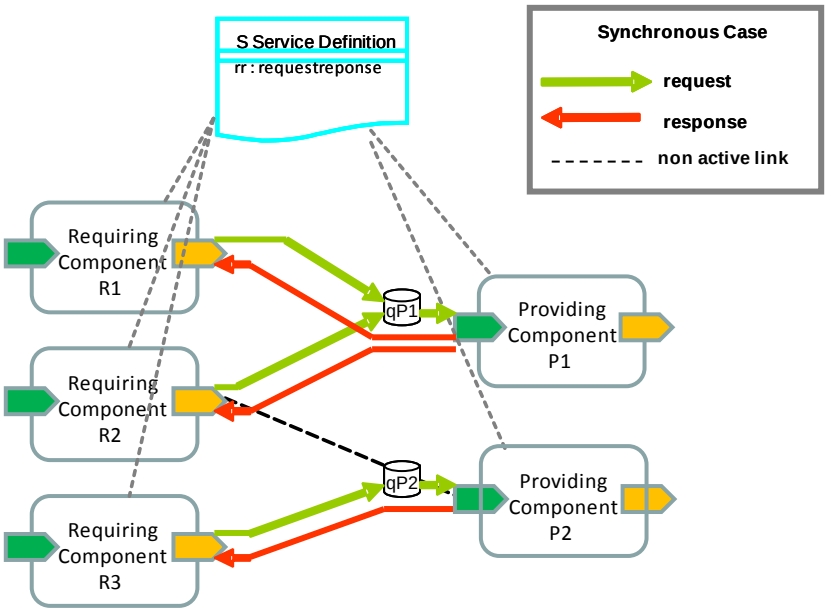

Figure 34 Synchronous Request-Response Operation

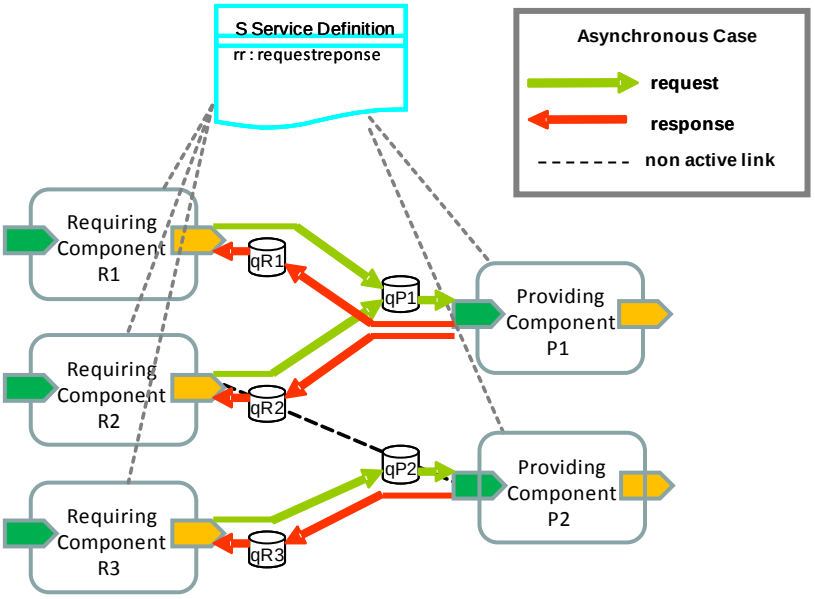

Figure 35 Asynchronous Request-Response Operation

In Figure 34 and Figure 35, Service definition S defines a single Request-Response operation (rr). This translates to:

- Service Requirers (R1, R2 and R3) issue a Request to their current Active Provider (P1 for R1 and R2, P2 for R3).

- Based on the Rank attribute of the Service Links, the Infrastructure will determine the Active Provider for Component R2. In this example the Active Provider for R2 is P1.
- The Service Providers (P1, P2) respond to the received Requests.
- For *Synchronous* Request-Response operations, the Requiring Component is blocked until the Response is received.
- For *Asynchronous* Request-Response operations, the Requiring Component is not blocked. The Response is received at some later time and processed by a callback function.
- Requests into a Provider are queued (qP1 and qP2) until the relevant Request_Received API callback function is called by the Provider Component's Container. This will cause additional blocking delays to Requirers of *Synchronous* Request-Response operations.
- Responses to *Asynchronous* Request-Response operations are queued in the Requiring Component's queue (qR1, qR2, qR3) until the relevant Response_Received API callback function is called by the Requirer Component's Container.
- Responses to *Synchronous* Request-Response operations are not queued in the Requiring Component which will be blocked waiting in the Request_Sync API function for the Response.

Versioned Data

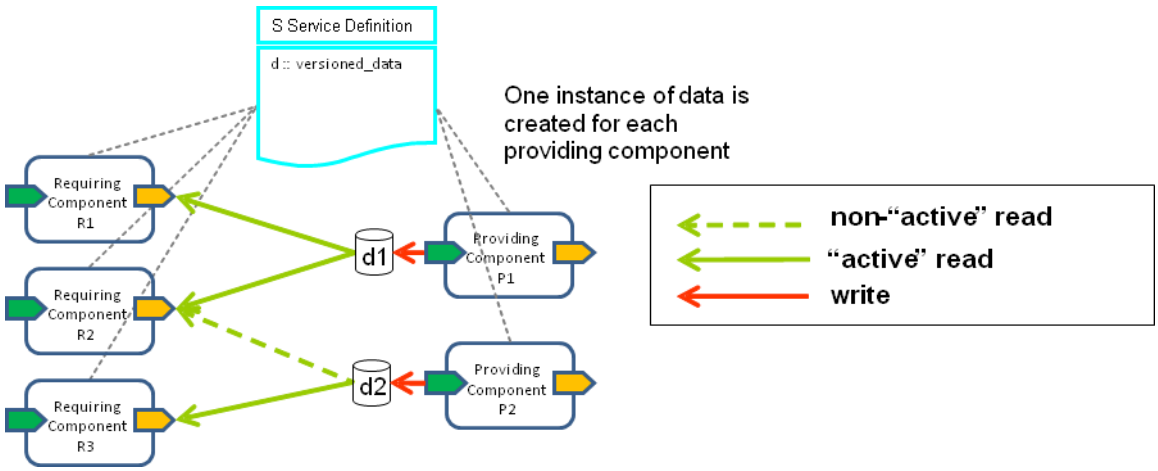

Figure 36 Selection of Versioned Data

In Figure 36 Service definition S defines one Versioned Data operation (d). This translates to:

- Two instances of data[5] (d1 and d2) created – one per providing Components (P1 and P2)
- Based on the Rank attribute of the Service Links, the Infrastructure will determine the Active Provider for the data Reader Component R2. In this example the Active Provider for R2 is P1 (hence R2 accesses data instance d1).
- Each Versioned Data instance is written by only one Application Software Component and can be read by many Application Software Components. In this example, even if P1 is the active Provider for R2,

[5] This is a logical view from the point-of-view of the Component. In an actual platform implementation, the data may be physically distributed and synchronized across the processing nodes in different ways.

P2 only accesses data instance d2. That means there is no underlying synchronisation between d1 and d2.

12 Module Operation Link Behaviour

This section shows the relationships between Service Operations and Module Operations.

Figure 37 shows possible interactions between Service Operations and Module Operations.

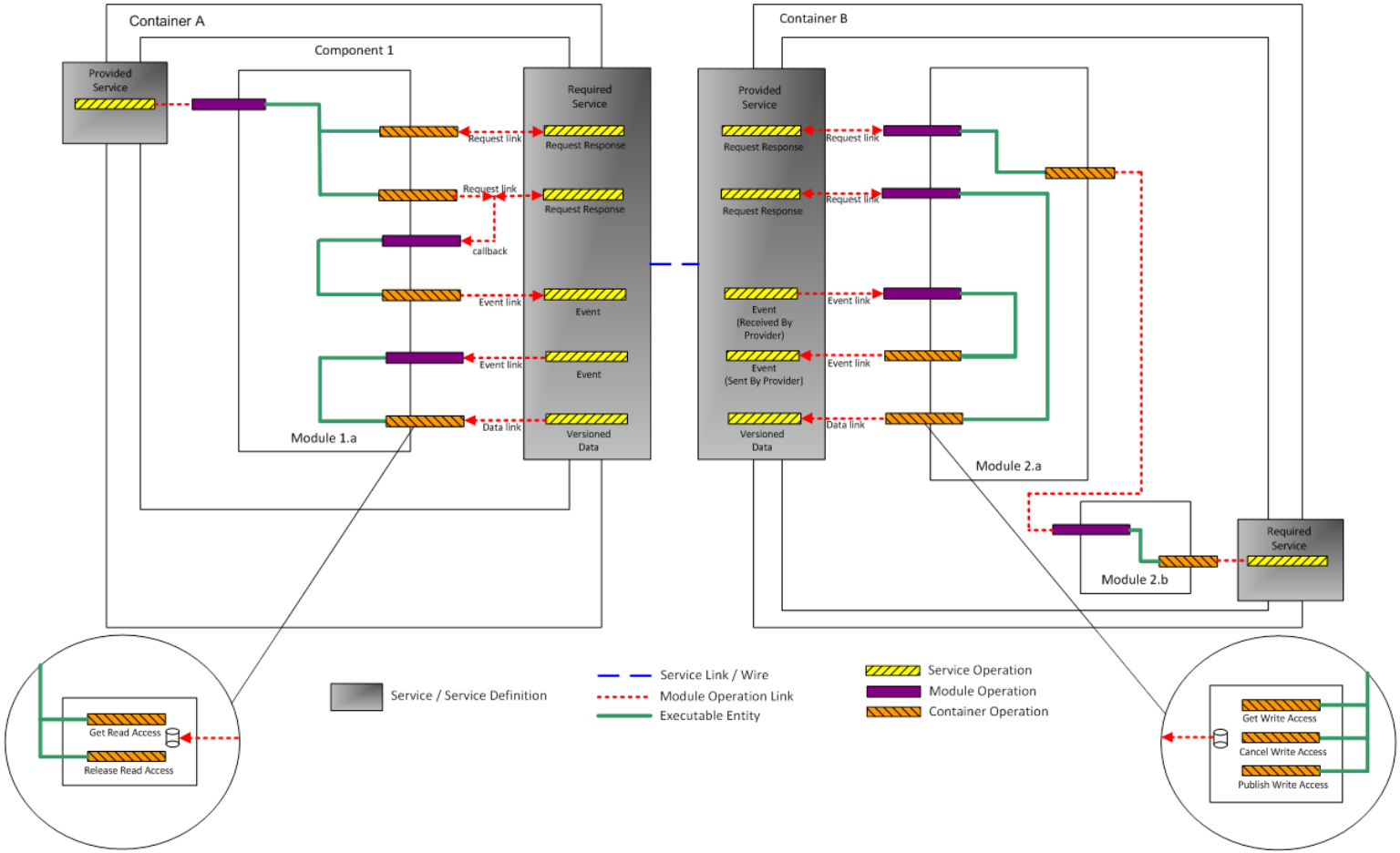

Figure 37 Interactions between Service Operations and Module Operations

The following give more details on each of these:

- The Grey boxes are the Services which are collections of Service Operations and are described by Service Definitions.
- Required and Provided Services are connected together using Service Links / Wires which are in Blue.
- The Yellow boxes are Service Operations.
- The Purple boxes are Module Operations which are entries onto Executable Entities which are in Green.
- The Orange boxes are Container Operations which are called from an entry point of a Module Instance.
- Incoming Service Operations are connected to Module Operations using Module Operation Links.
- Container Operations can be connected to outgoing Service Operations using Module Operation Links.
- Container Operations can be connected to Module Operations using Module Operation Links.
- Service Operations cannot be mapped to other Service Operations using Module Operation Links.
- All Module Operation Links require Container code.
- The names of Service Operations and Module Operations which are connected together don't have to match.
- A Request Response Service Operation can only be mapped to a single Module Operation.
- Multiple Event Service Operations can be mapped to the same Module Operation.
- One Container Operation can be mapped to multiple Event Service Operations.
- Versioned Data is always Read or Written by an entry point of a Module Instance using a Container Operation.

13 Utilities

An ECOA Software Platform provides utility functions for acquiring time and for generating logs. The Architecture Specification Part 4 contains more detail regarding these functions.

One of the ECOA Software Platform provided functions is a method for allowing access to global time. It is a system specific decision how this global time is synchronised, and at what precision, however ECOA assumes that time values acquired through these functions are synchronised across the system.

14 Inter Platform Interactions

In order to provide interoperability between ECOA Software Platforms, a message protocol, termed ECOA Logical Interface (ELI), has been defined. This message protocol requires an underlying transport protocol for its implementation. The choice of the underlying transport protocol is left to the system designer depending on system-level requirements (performance, security, etc.). As an example, a binding to the UDP transport layer has been defined. These can be found in the Architecture Specification Part 6.

15 Composites

ECOA is fully compliant with the SCA Composite concept.

A composite is described by its definition, the list of its Application Software Components and the associated Assembly Schema of these Application Software Components. The Composite will provide several Services, each one linked to one or several Services or provided by its Application Software Components. This kind of Service Link is called a Promotion Link. The Composite will require several Services required by internal Application Software Components. The link used here is also called a promotion link. An Application Software Component external to the Composite is only connected to

Services provided or required at Composite level and has no knowledge of the internal Application Software Components.

ECOA allows specifying an Initial Assembly Schema made of hierarchical composites containing ECOA components in order to help with design abstractions. However, the Final Assembly Schema is a flattened single composite containing all wired ECOA components.

Figure 38 shows an example Composite constructed with four Components.

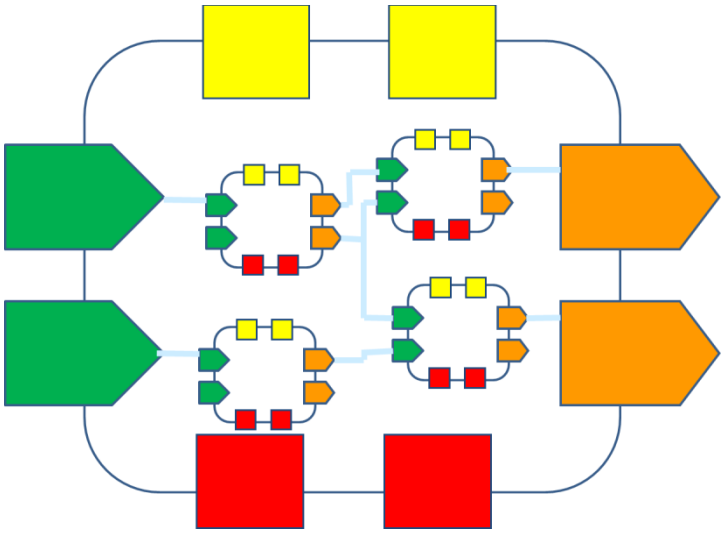

Figure 38 A Composite