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

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

Issue: 6

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. 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 Services and Service Link behaviour in an ECOA® system.
- Section 9 describes the System Management mechanisms that are provided by the Infrastructure.
- Section 10 describes the support for scheduling within an ECOA® system.
- Section 11 describes Module Operation behaviour.
- Section 12 describes the utilities provided by the ECOA® Software Platform.
- Section 13 describes how inter-platform communication occurs within an ECOA® system.
- Section 14 describes the concept of a composite (collection of Application Software Components).
- Section 15 describes the usage of composites in Platform Integration.

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. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development should rely on the DefStan or BNAE publications of the ECOA standard.

3 Normative References

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

Architecture Specification Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 6 Architecture Specification Part 10 – Ada Language Binding
Architecture Specification Part 11	IAWG-ECOA-TR-031 / DGT 154934 Issue 6 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 (with or without access control)

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 8.2 and 11 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.

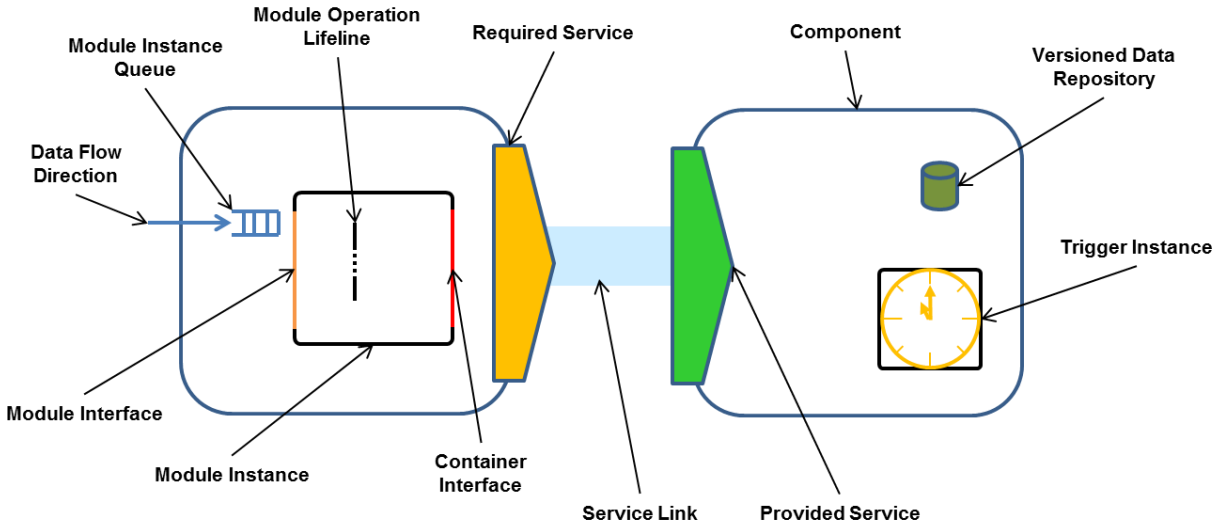


Figure 1 ECOA Interactions - Key

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.).

7.2 Module Instance Queues

Module run-time behaviour is dependent upon the Module Runtime Lifecycle state (see 9.1). 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 10.2 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.

Whenever a request is discarded by the ECOA Infrastructure, the ECOA Infrastructure returns with « no response » error code to the Client Module as far as is possible.

Note: it is recommended that a timeout is specified on the Client side. This is due to the fact that some failures cannot be detected and a 'no response' error may never be received. For example a network failure may prevent a remote ECOA Infrastructure on the server side from informing the client that it has discarded the request, or the request itself may never reach the server.

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 Module Instance that is not in the **RUNNING** state, the Event is discarded silently.

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.

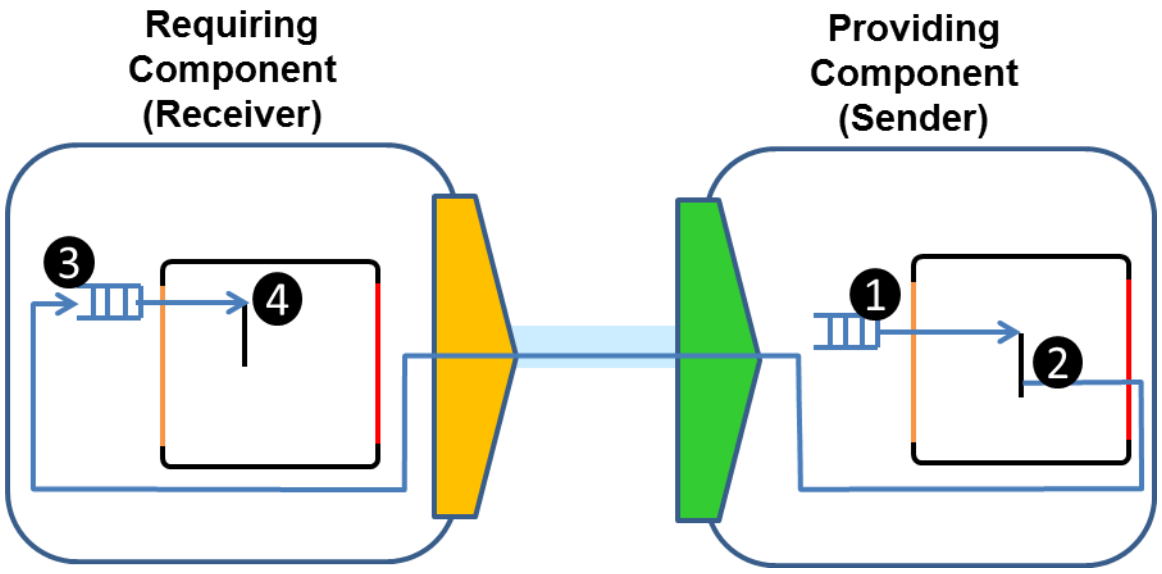


Figure 2 Event Sent by Provider

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.

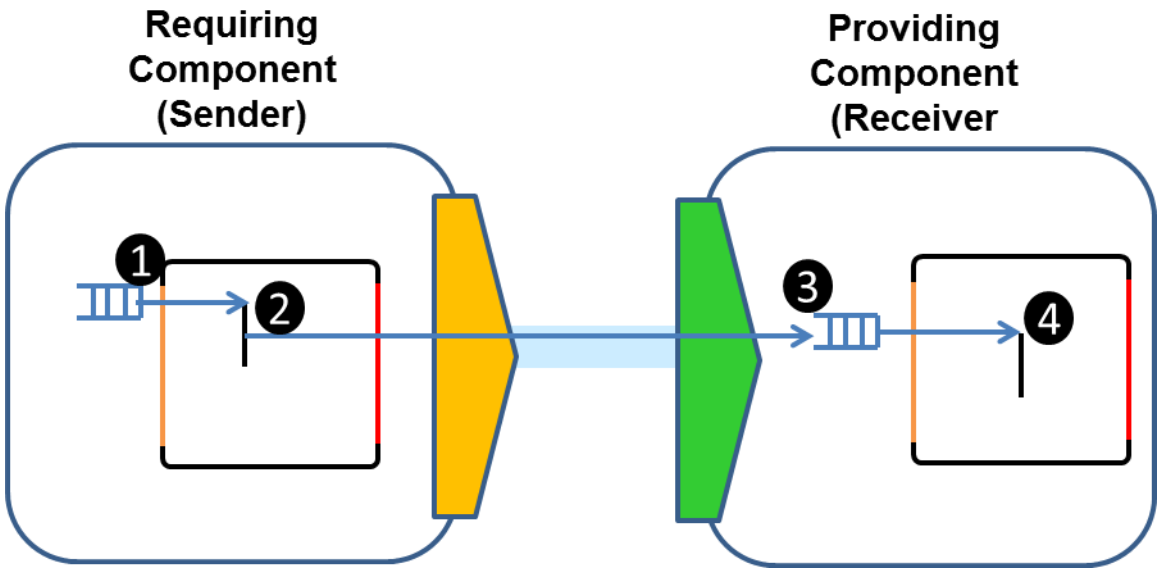


Figure 3 Event Received by Provider

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.

For each Request-Response, the set of possible Clients and Server is identified at design time, as is whether the Client uses the synchronous or asynchronous mechanism. For a given Client, there is only one Server.

A Response from a Server is only sent to the particular Client that has issued the Request.

The client Container instance will implement a timeout (if defined at component implementation level) 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 timeout. The value of the timeout is defined

at component implementation level; this can be related to QoS maximum response time in the case of service operations. 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 less than zero, it is considered as infinite; the Client of a Synchronous Request-Response remains blocked indefinitely if it never receives a Response.

A Request may fail if the Server is not available (e.g. it is not RUNNING, the remote platform is DOWN, network error). When such a failure can be detected by the Infrastructure, it returns with « no response » error code to the client, and reports the fault to the fault-management Infrastructure. Where failures cannot be detected by the infrastructure, the client may use the timeout to avoid being blocked indefinitely.

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.

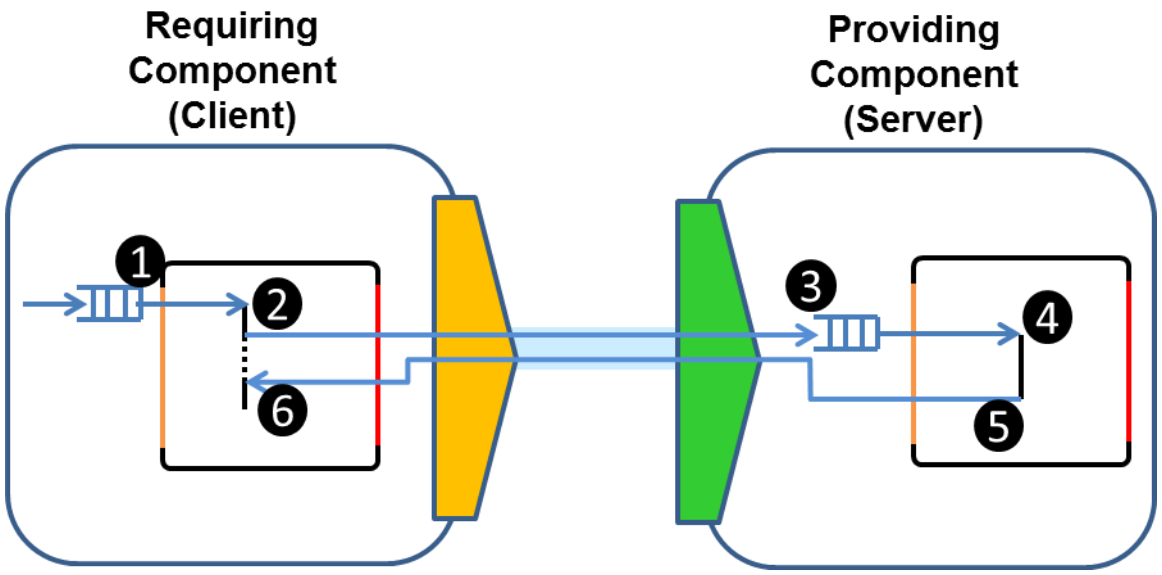


Figure 4 Synchronous Client Request-Response

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 9.1).

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 and it is in the **RUNNING** state. If not, the Request is discarded and a « no response » error code is returned to the client.

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.

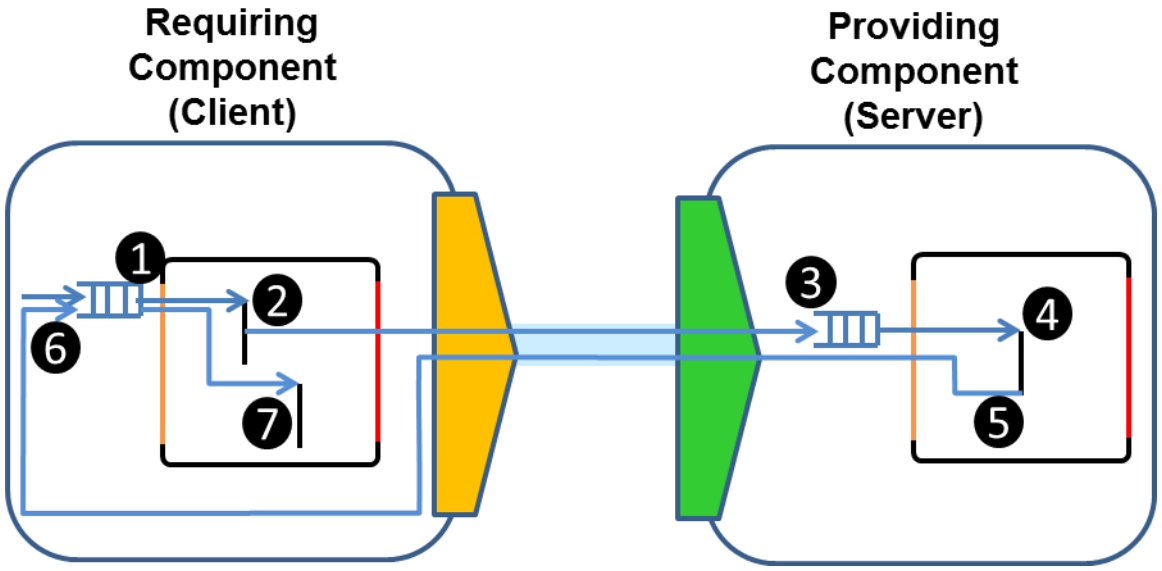


Figure 5 Asynchronous Client Request-Response

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 and it is in the **RUNNING** state. If not, the Request is discarded and a « no response » error code is returned to the client.

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** as long as it is in the **RUNNING** state, 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.

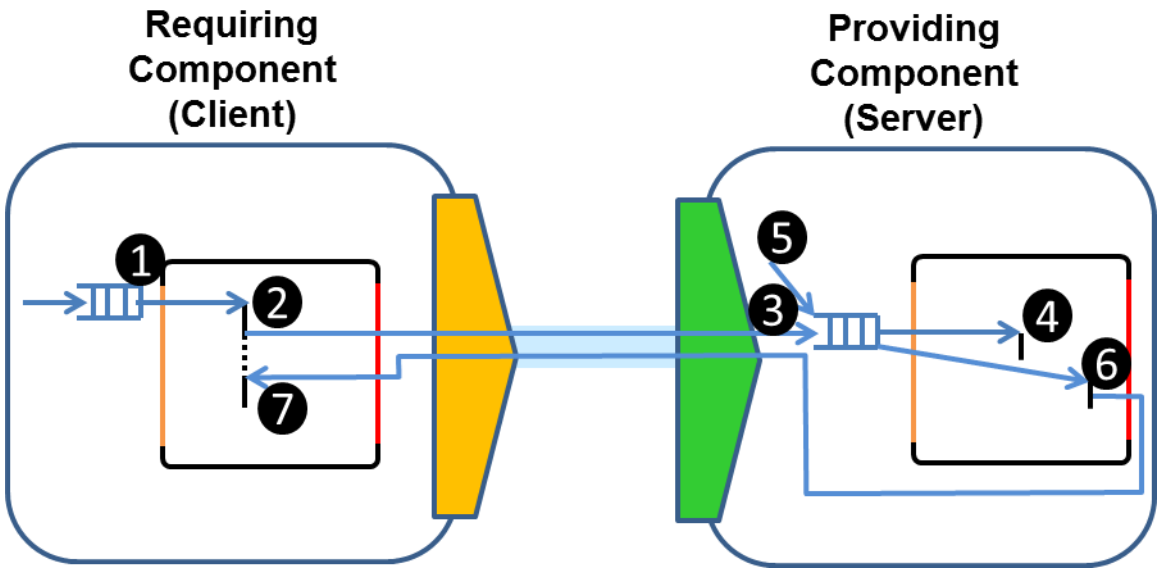


Figure 6 Deferred Response Server Request-Response

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

The Versioned Data mechanism allows Module Instances to share typed data. A reading Module Instance is named a “Reader”, and a writing Module Instance is named the “Writer”.

For each Module Operation Link that uses this type of interaction, the Versioned Data mechanism can be configured to operate with or without access control.

With access control, the ECOA Infrastructure ensures a concurrency-safe read-write paradigm between Module Instances by making a local copy of the data when a reader or a writer accesses it.

Without access control, the local data repository is accessed directly by all Module Instances. Therefore, concurrency must be managed at application level by the Component Supplier.

Note: Versioned Data without access control may be used for improving the performance of ASCs in which large datasets are shared between several Module Instances.

Versioned Data without access control may be selected on a Module Operation Link only if all Readers and Writers connected over this link are Module Instances of the same ASC Instance and are deployed within the same Protection Domain. This is the condition for the ECOA Infrastructure to implement a single Versioned Data repository without local copies.

Any Versioned Data over Service Links must be implemented with access control.

Note: Platform Tooling may check that this condition is met when parsing ECOA XMLs prior to Platform Integration Code generation.

Access control is enabled by default for each Module Operation Link that uses the Versioned Data mechanism, unless explicitly disabled by the Component Supplier.

The details of the Versioned Data mechanism with and without access control are described in the following sections.

7.5.1 Versioned Data with access control

In the case of Versioned Data with access control, 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. When a Versioned Data has been written by a Module Instance, the ECOA Infrastructure makes it available across the ECOA System to any reader Module Instance, according to the Assembly Schema and according to Component Implementations.

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.

The ECOA Infrastructure provides reader Module Instances with a stamp associated with the reception of a versioned data. This stamp is available within the Versioned Data handle. It allows a given reader Module Instance to know if the data has been published (since an arbitrary previous access) or not. The ECOA Infrastructure hosting a reader Module Instance is responsible for updating the stamp of the Versioned Data.

The behavior of the versioned data stamp is the following and it is up to the platform supplier to choose how to manage the stamp value accordingly:

- The stamp value is “0” when the return status of a read operation is not OK.
- For any given reader, each time the data value is updated locally for that reader as a consequence of a publish action by a writer, the stamp value shall be different from the N previous values, N being at least equal to 2^{32} - 1. This allows being robust to N updates of the data since it was last read by the reader.
- The stamp may wrap around when reaching max value.

Note 1: the consequence is that, from the reader point of view, the stamp may not necessarily be perceived as increasing (it could go backwards), nor changed by increment of 1 or fixed increments. This behavior should be taken into account by the module application code.

Note 2: the fact that the stamp is incremented on reader side implies that, for a given versioned data, different reader Module Instances may get different stamp values when reading the data, because of asynchronous distribution of the data across the ECOA System (especially with regard to distribution over ELI). Thereby, stamp values should not be compared between different readers.

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.

The ECOA Infrastructure implements Versioned Data repositories with respect to the Assembly Schema and Component Implementations which define data links between Readers and Writers. It is possible for multiple instances of the same Versioned Data repository to exist in an ECOA system e.g. where the Readers are deployed on different Computing Nodes. As illustrated by Figure 7, Figure 8 and Figure 9 below, a local Versioned Data repository has to be implemented for an ASC as soon as there are both internal and external Writers of the Data, or as soon as there is more than one external Writer of the Data (from the viewpoint of that ASC).

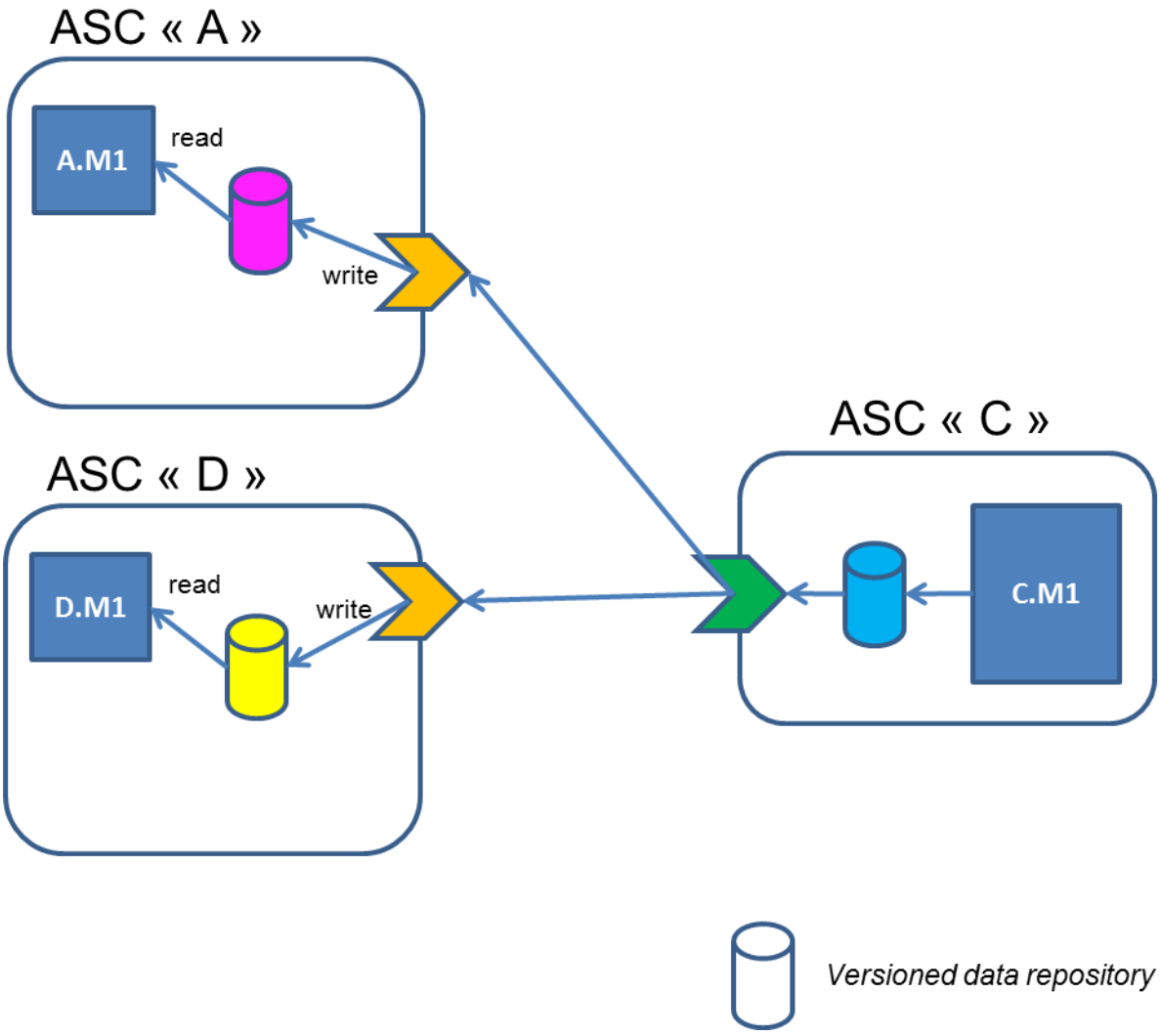


Figure 7 Versioned Data Repositories – Single Writer Use Case

Figure 7 illustrates a use case of a single external Writer of the Data and no internal Writer (from the viewpoint of ASCs “A” and “D”). In this case, the ECOA Infrastructure may implement a single repository for these three ASCs, depending on optimisations allowed by ASCs deployment onto Computing Nodes.

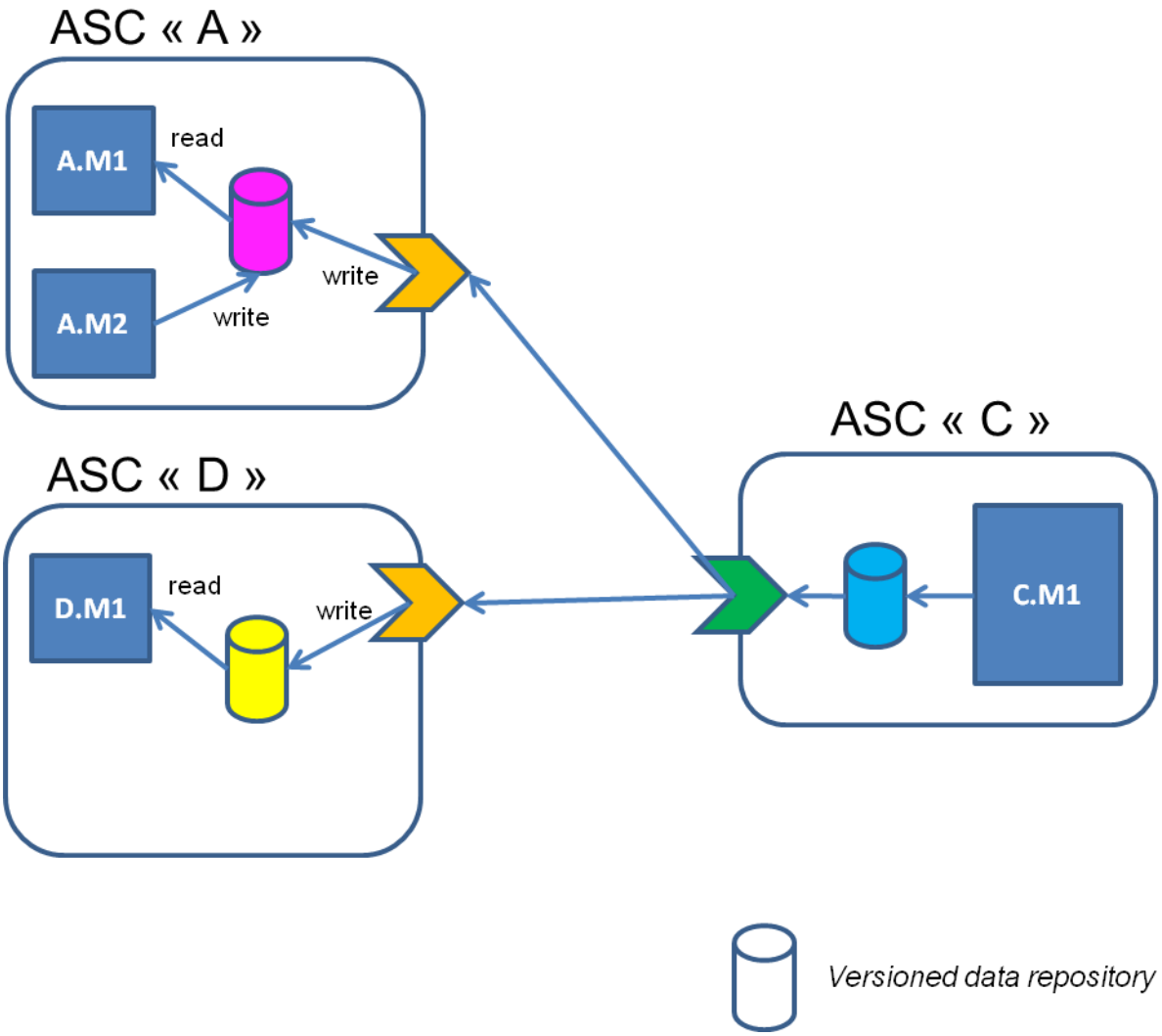


Figure 8 Versioned Data Repositories – Internal + External Writer Use Case

Figure 8 illustrates a use case of simultaneous external and internal Writers of the Data (from the viewpoint of ASC “A”). In this case, the ECOA Infrastructure should implement a separate Versioned Data repository for ASC “A”. It may implement a single repository for ASCs “C” and “D”, depending on optimisations allowed by ASCs deployment onto Computing Nodes. The reason for this is that Module Instance “D.M1” and “C.M1” must not have visibility of any data written by Module Instance “A.M2” due to the limitations imposed by the data flows according to the Wires and Data Links. Module Instance “A.M1” would still have visibility of data written by Module Instance “C.M1”.

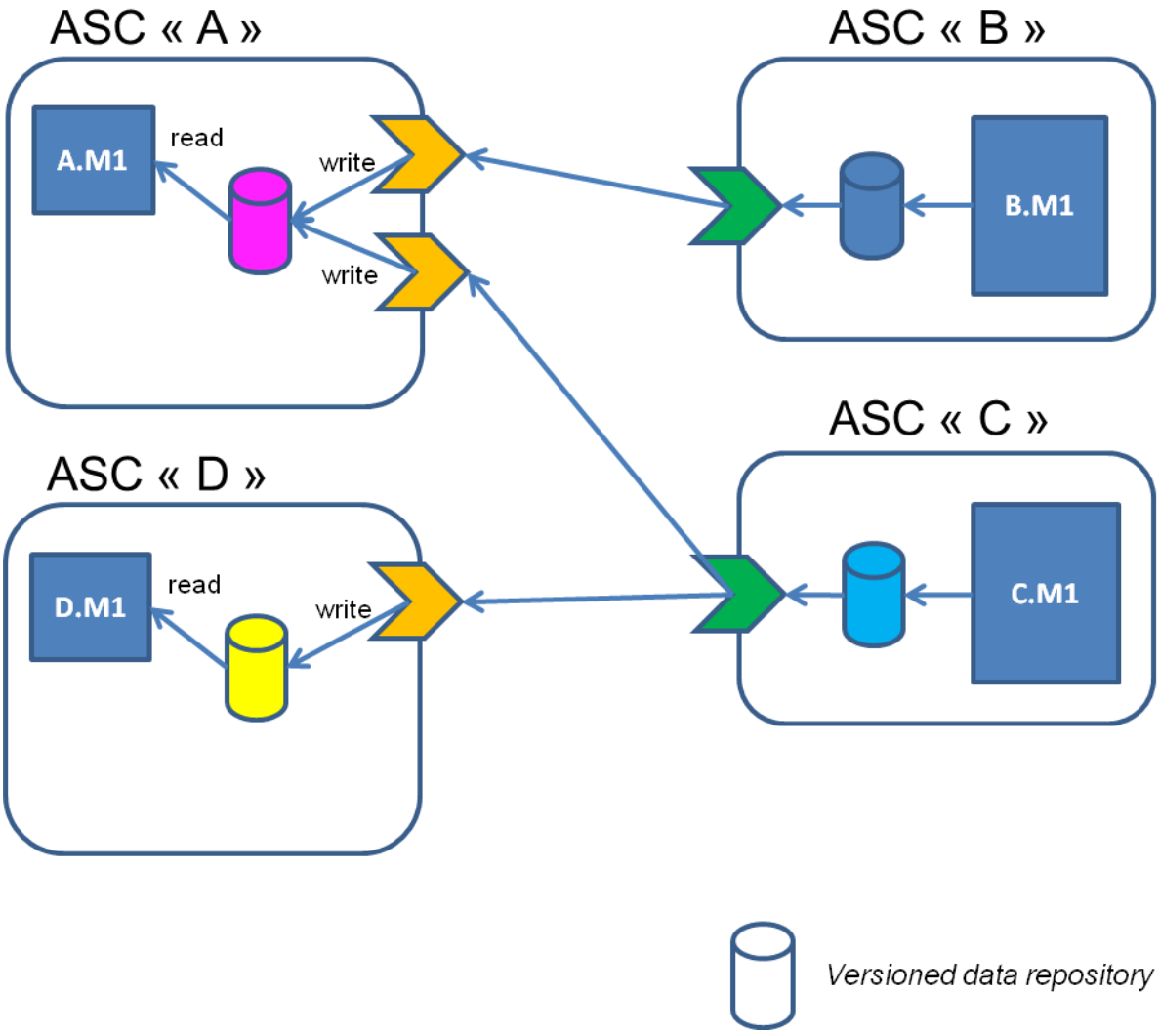


Figure 9 Versioned Data Repositories – Several External Writers Use Case

Figure 9 illustrates a use case of multiple external Writers of the Data (from the viewpoint of ASC "D"). In this case, the ECOA Infrastructure implements different Versioned Data repositories for ASCs "A", "B" and "C". It may implement a single repository for ASC "C" and ASC "D", depending on optimisations allowed by ASCs deployment onto Computing Nodes. Module Instance "D.M1" must not have visibility of any data written by Module Instance "B.M1" due to the limitations imposed by the data flows according to the Wires and Data Links. Similarly, Module Instances "B.M1" and "C.M1" must not have visibility of any data written by each other.

When the access to the data begins:

- The Reader always gets the latest copy of data available locally,
- The Writer either gets the latest copy of data available locally ("Read+Write" mode) or gets an uninitialized copy ("Write only" mode), according to a Metamodel attribute. The default mode is "Read+Write".

NOTE: "Write Only" mode avoids copying the latest data value for Writers which do not need to read it, thereby improving performance within the ECOA Infrastructure.

As stated previously, the stamp enables the caller to determine if the data has been refreshed since the last read operation. 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.

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 read access ends when it is “released”

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 number of concurrent read or write accesses is incremented each time the ECOA Infrastructure grants access to a Module Instance after that Module Instance made a read or write access request. The number of concurrent read or write accesses is decremented each time a Module Instance releases a read access or publishes/cancels a write access. The number of concurrent read or write accesses cannot exceed the maxVersions attribute value. 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.

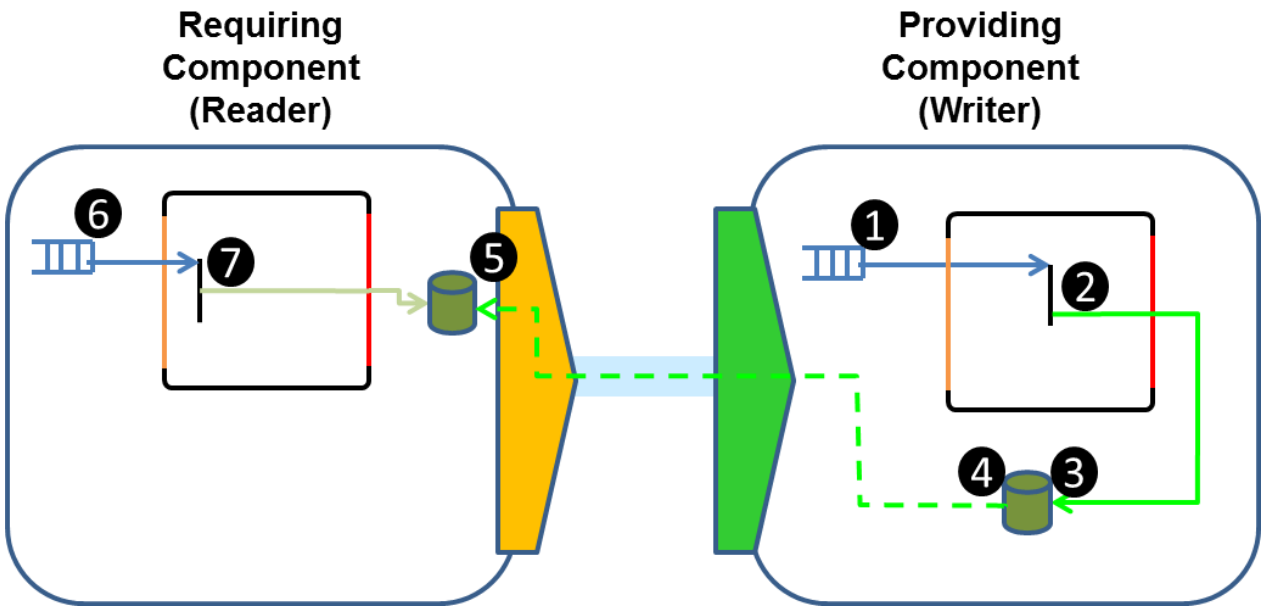


Figure 10 Versioned Data with access control Behaviour

Figure 10 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**.

7.5.2 Versioned Data without access control

In the case of Versioned Data without access control, the ECOA Infrastructure does not make any local copies of the data when a Reader or a Writer accesses it. Instead, the actual data repository address is passed by the ECOA Infrastructure to all Readers and Writers involved in the corresponding Module Operation Link. Therefore, it is the application responsibility to manage/synchronise/coordinate accesses to the repository.

The behaviour of the ECOA Infrastructure is the following when access control is disabled:

- The versioned data stamp is updated as a consequence of a publish action by a writer,
- The number of concurrent read or write accesses is incremented or decremented depending on read or write access requests and according to read or write access releases by Module Instances, although there are no local copies being created or deleted by the ECOA Infrastructure upon these actions,
- The number of concurrent read or write accesses cannot exceed the maxVersions attribute value,
- A publish or cancel action by a Writer has no effect on the repository data as any changes made to the data would have already been done directly, however the number of concurrent write accesses will be decremented/
- The ECOA Infrastructure cannot prevent a Reader from being able to write in the repository once the actual data repository address has been provided.

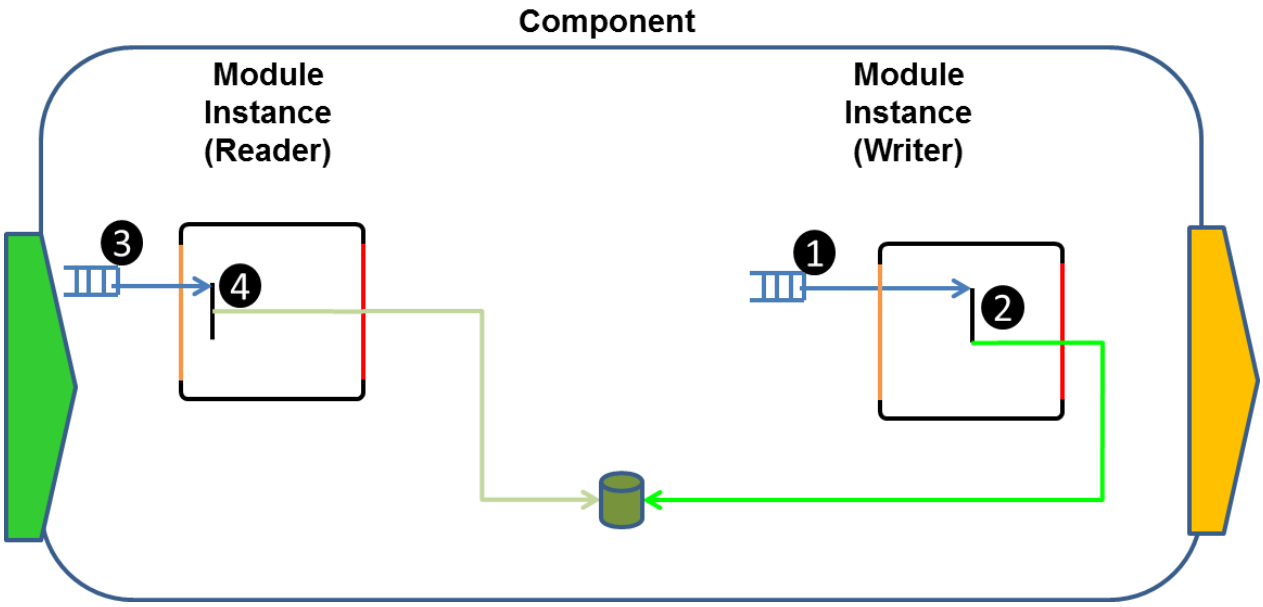


Figure 11 Versioned Data without access control Behaviour

Figure 11 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 updates the data repository at **point 2**. No local copy is made by the ECOA Infrastructure for this write operation.

Point 3 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 4** through which it accesses the data repository directly without a local copy being made by the ECOA Infrastructure.

Figure 11 illustrates that functional coordination between Module Instances must be implemented by the Component Supplier in order to avoid conflicting accesses to the repository.

7.5.3 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. This behaviour applies to Versioned Data with or without access control.

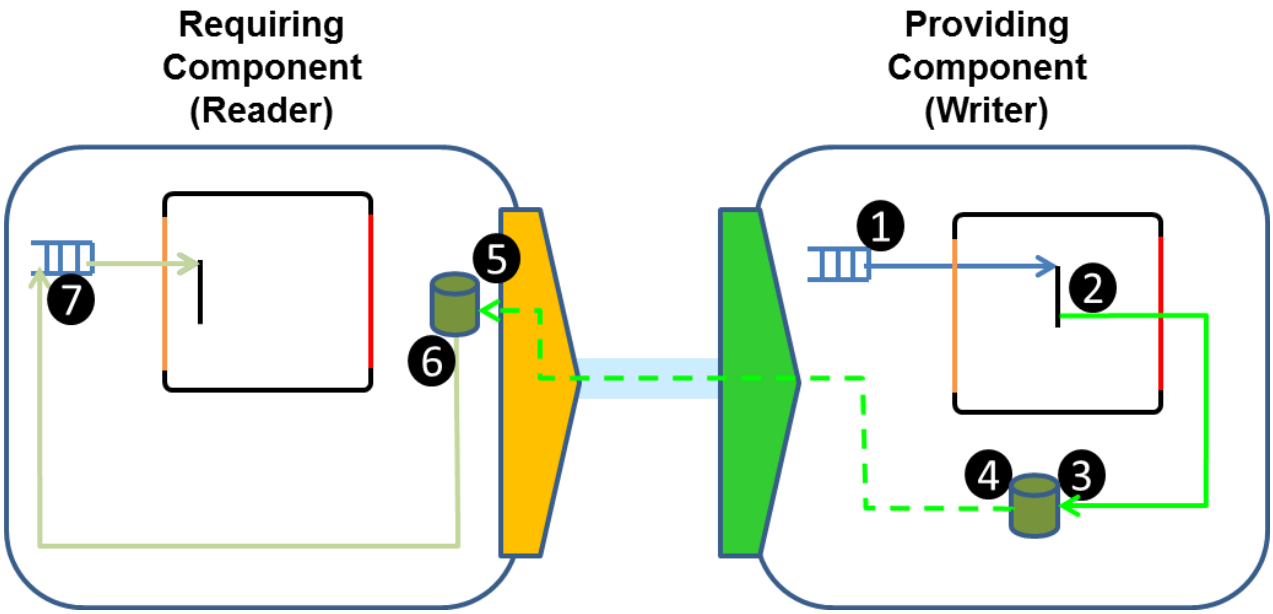


Figure 12 Notifying Versioned Data Behaviour – illustrated with access control

Figure 12 shows an example of Notifying Versioned Data with access control. At **point 1,** a Module Operation is 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** as long as it is in the **RUNNING** state. This invokes the notification Module Operation.

The Module Instance may then use standard Container Operations for getting a read-only copy of the latest data value, as detailed in section 7.5.1.

Note: the latest data value at the time of the read request may be different from the data value at the time of the notification.

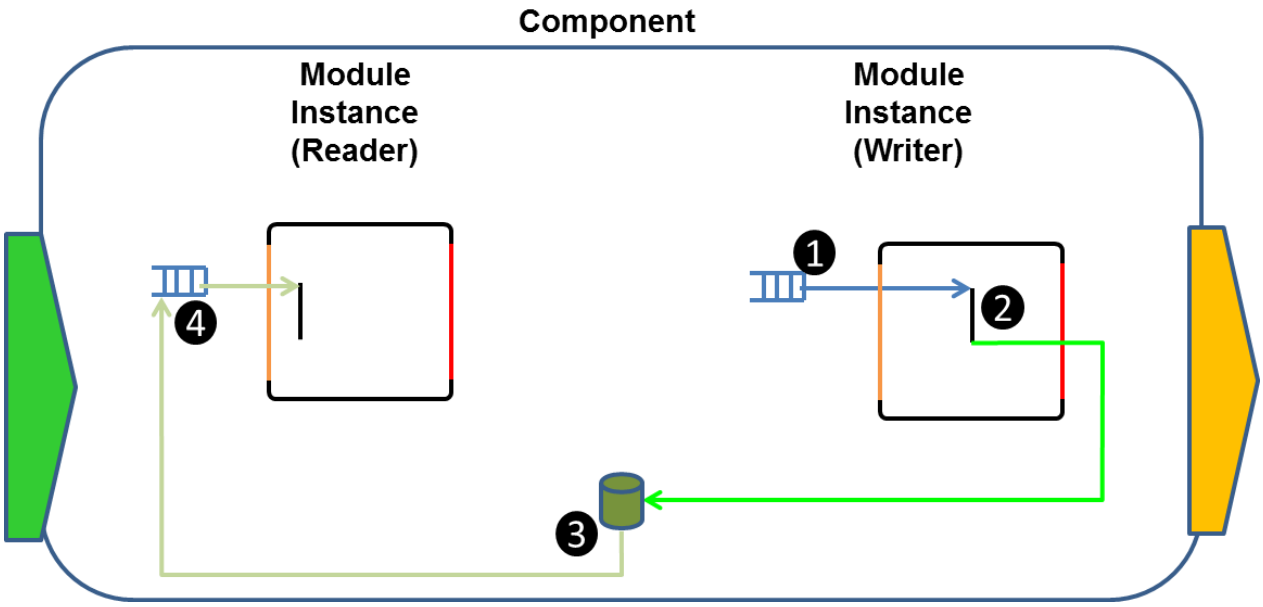


Figure 13 Notifying Versioned Data Behaviour – illustrated without access control

Figure 15 shows an example of Notifying Versioned Data without access control. At **point 1,** a Module Operation is invoked on the Writer Module Instance as a result of some other activity. During this execution, the Module Instance updates the data repository and performs a publish Container Operation at **point 2**. No local copy is made by the ECOA Infrastructure for this write operation.

As a result of the publish Container Operation performed by the Writer Module Instance, a notification Event is generated at **point 3** which is queued on the Reader Module Instance Queue at **point 4** as long as it is in the **RUNNING** state. This invokes the notification Module Operation.

The Module Instance may then use standard Container Operations for accessing the data repository directly without a local copy being made by the ECOA Infrastructure, as detailed in section 7.5.2.

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 14.

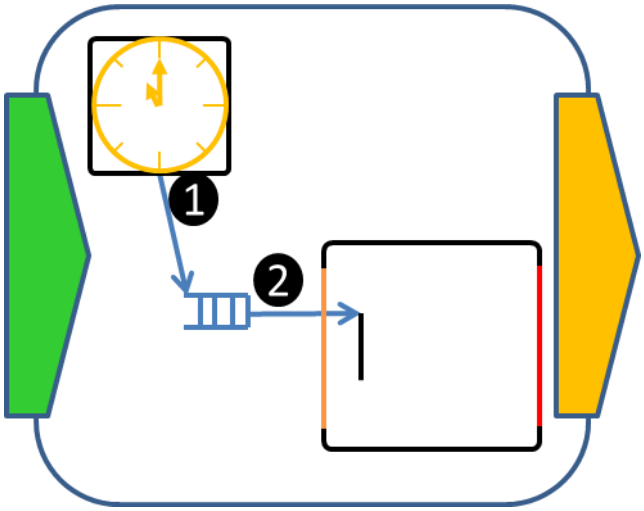


Figure 14 Trigger Behaviour

Figure 14 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 ECOA Infrastructure, i.e. one period after its lifecycle state has become RUNNING.

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 ECOA Infrastructure 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 in the form of the expiry time at which the `out` Event shall be sent. 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 expiry time. The remaining parameters can be of any ECOA type or type derived from ECOA types.

The Module Instance calculates the expiry time parameter of the `in` Event by summing the intended delay and the current **absolute system time** obtained by invoking the **get_absolute_system_time** API.

The `out` Event is generated with exactly the same parameters as the `in` Event, except that the expiry time parameter is omitted. The `out` Event is sent at the specified expiry time, except if the expiry time is already in the past when the Dynamic Trigger processes the `in` Event. In such failure case, the `out` Event is sent immediately and the ECOA Infrastructure raises an Error to the ECOA Fault Handler.

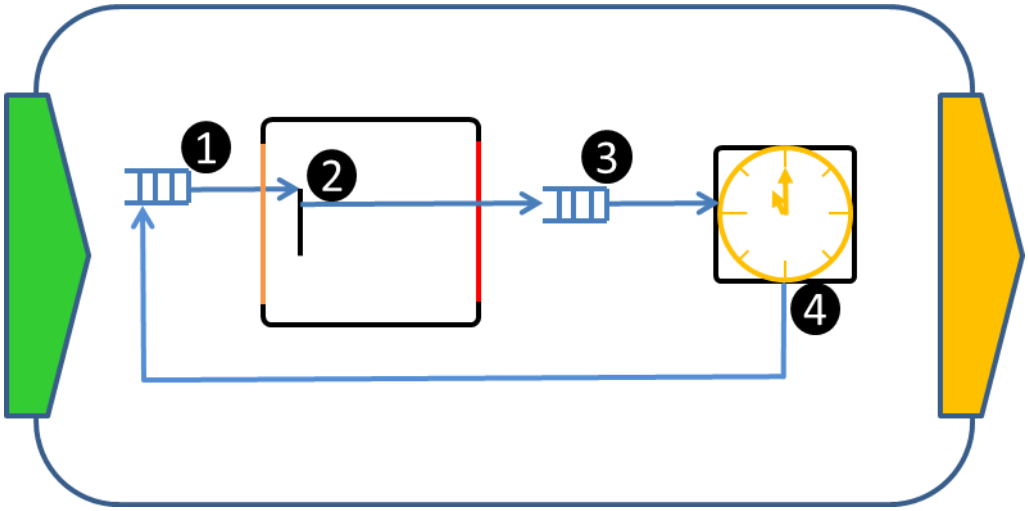


Figure 15 Dynamic Trigger Behaviour

Figure 15 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) specified via the expiryTime parameter. 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 (i.e. when the current time becomes equal to or greater than the expiry time), 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 ECOA Infrastructure 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 expiryTime received in the “in” Event
  - Parameters:
    - expiryTime : expressed as **Absolute System Time** and typed by ECOA:global_time (seconds, nanoseconds)
    - p1, p2, etc: user defined parameter based on ECOA types
- EventSent out

- The Dynamic Trigger sends an “out” Event at the expiry time received in the “in” Event.
- 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).

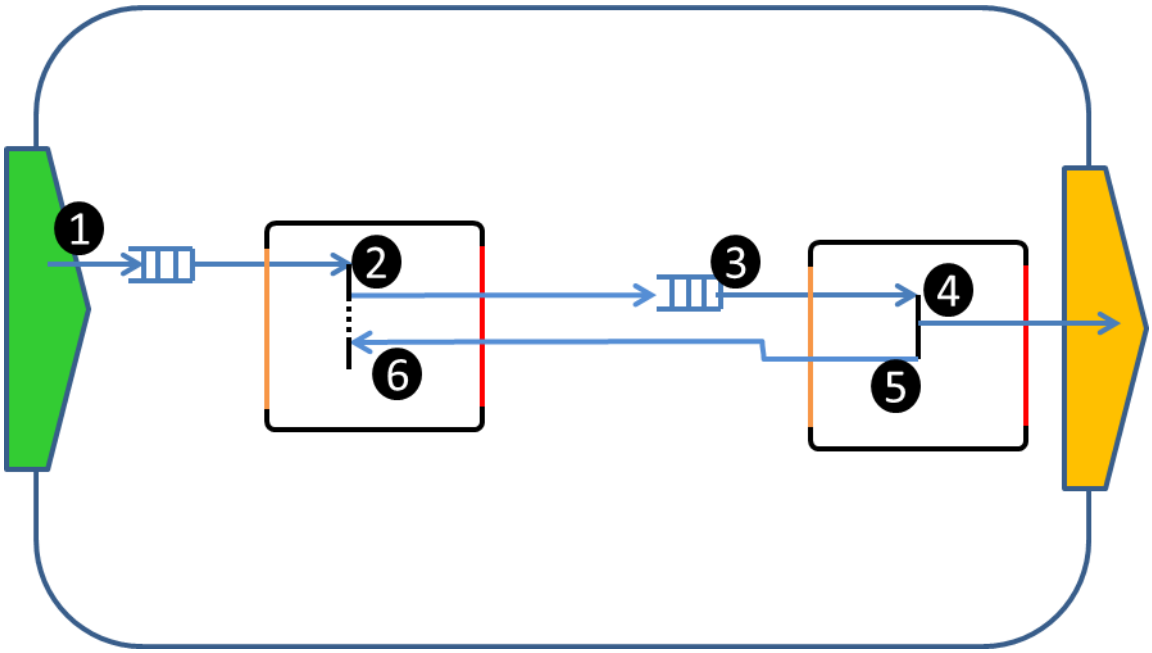


Figure 16 Interactions within Components – Synchronous Request-Response

Figure 16 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 to 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.
- Public PINFO – is data accessible by any Module Instance in the Assembly Schema.

PINFO is Read-Only.

PINFO can be referenced (i.e. accessed) by multiple Module Instances.

Figure 17 illustrates the aspects of PINFO and how they are being declared in an ECOA system.

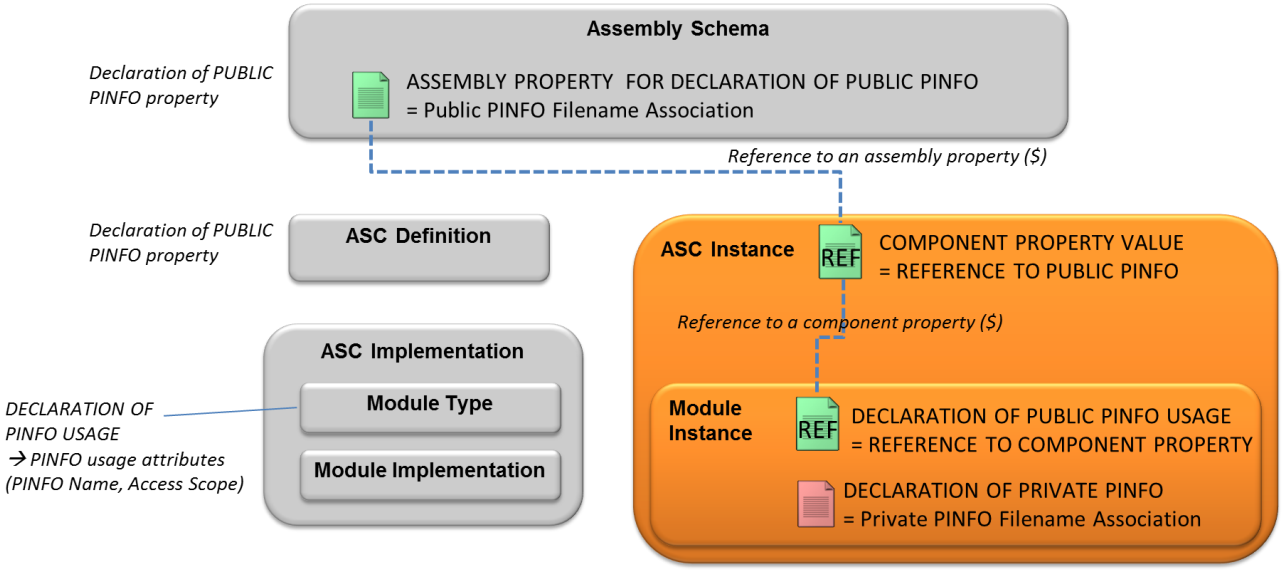


Figure 17 Aspects of PINFO

7.10.1 PINFO Attributes

Private and Public PINFO have the following attributes associated with them:

- 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 PINFO is bounded by the following rules:

- PINFO Filename Association is declared as a string:
  - The maximum length of that string is 256 characters,
  - The PINFO path and filename declaration can use alphanumeric characters in addition to the following special characters: "_", ".", "/"
  - The PINFO name declaration can use alphanumeric characters in addition to the following special character: "_"
- PINFO Filename Association is not case sensitive.

7.10.2 Private PINFO

- A Filename Association can be associated with more than one Private PINFO, 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.

7.10.3 Public PINFO

- 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.

7.10.4 PINFO organization

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

- PINFO are stored in directories specified in §7.10.2 and §7.10.3.
- 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 18.

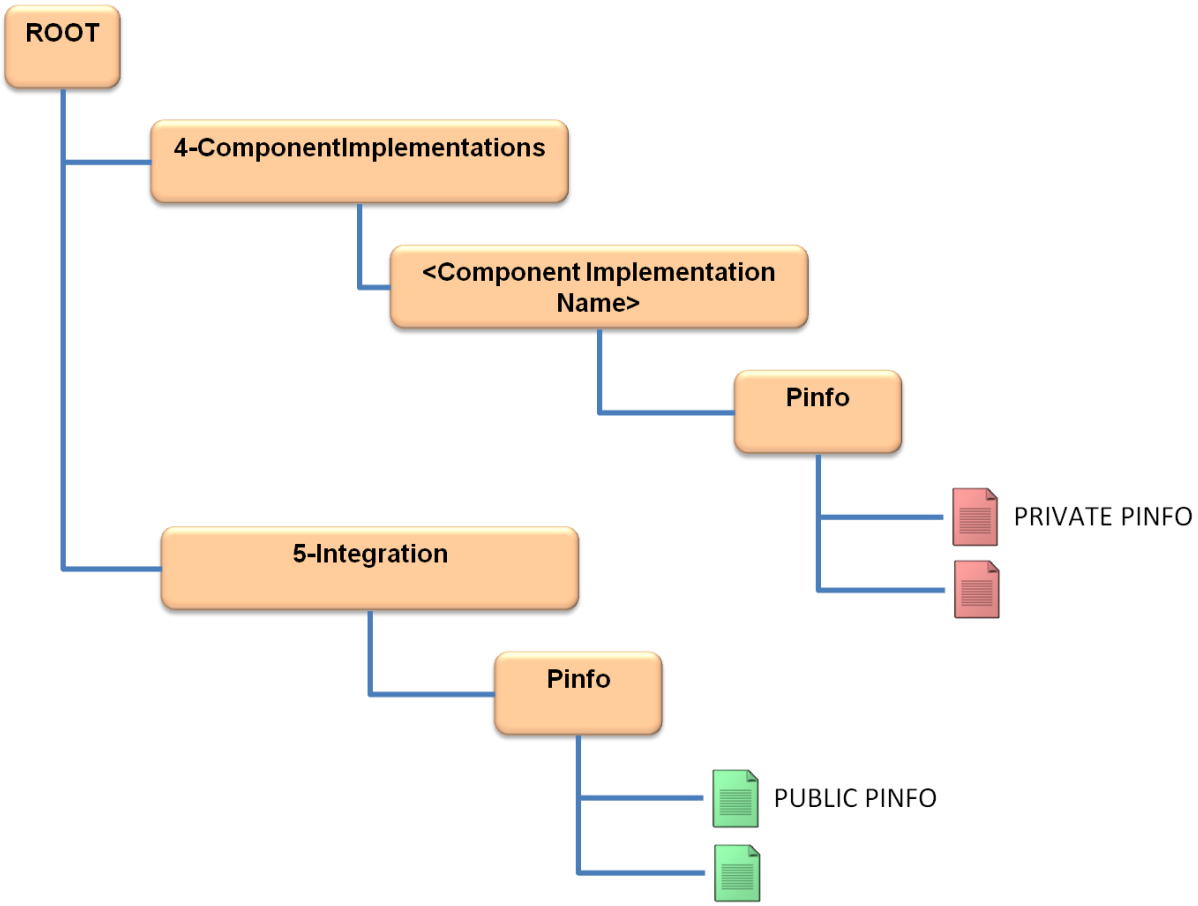


Figure 18 PINFO organization prior to deployment

- 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.
  - The ECOA Infrastructure shall ensure non-conflicting access to PINFO by Module Instances.

7.10.5 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 19:

- PINFO is an array of bytes,
  - Note: the use of PINFO may limit ASC portability if the binary PINFO data and the functional code that accesses it are not designed to explicitly handle endianness in a platform independent way (i.e. byte order when accessing PINFO). It is up to ASC suppliers to ensure that binary PINFO data is managed in a platform independent way.
- 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 a size denoted by PINFO'size.

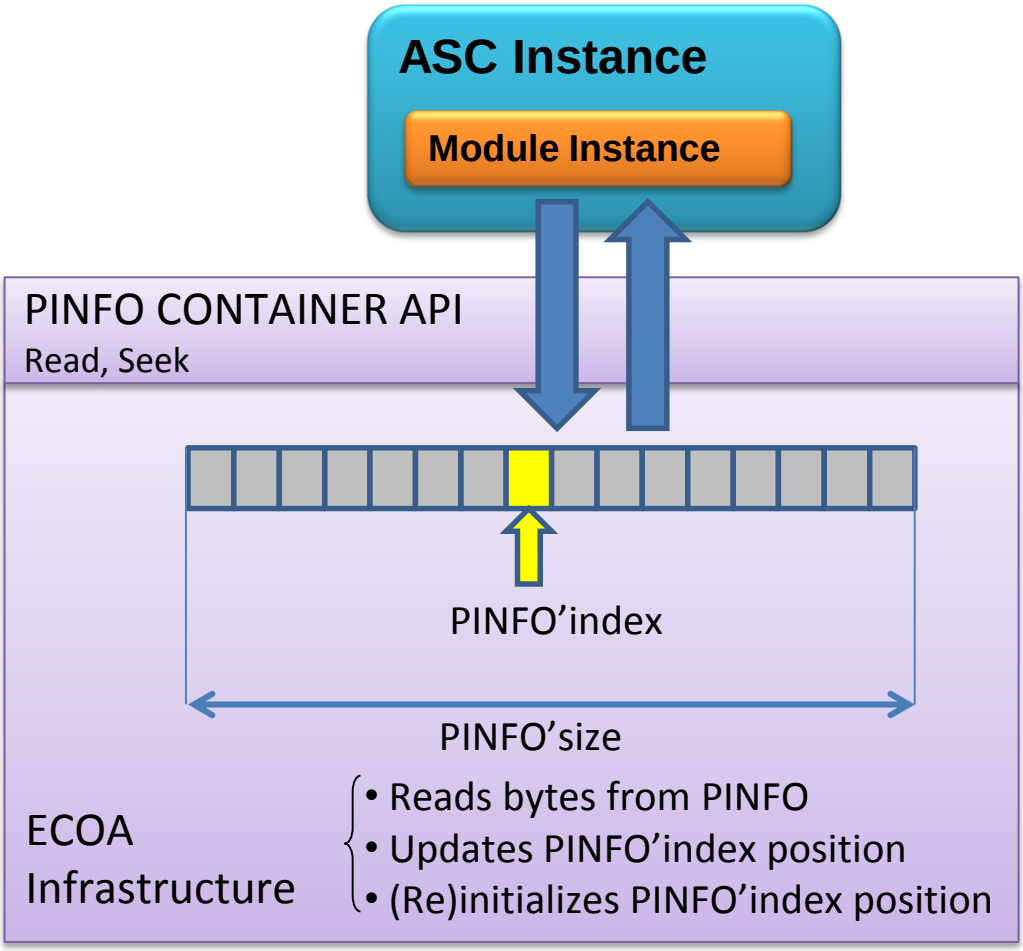


Figure 19 PINFO API

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 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.

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

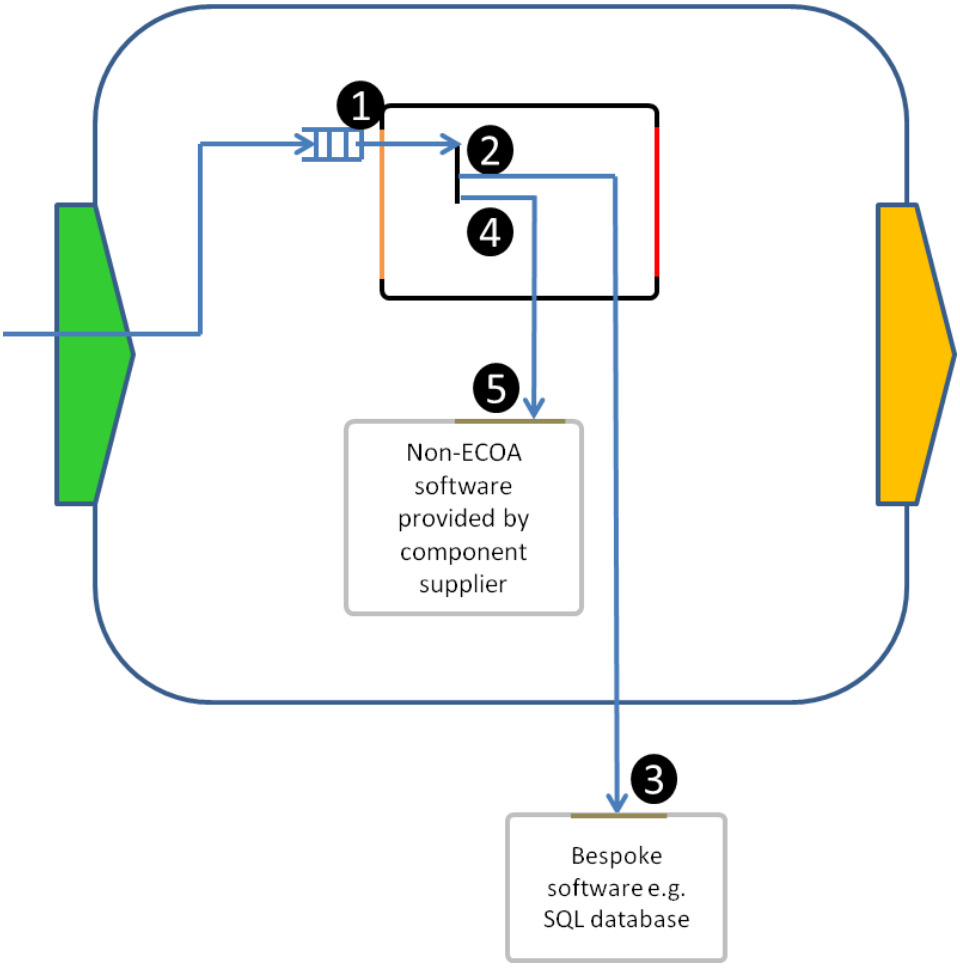


Figure 20 Driver Component Example – Interaction with Bespoke APIs

Figure 20 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.

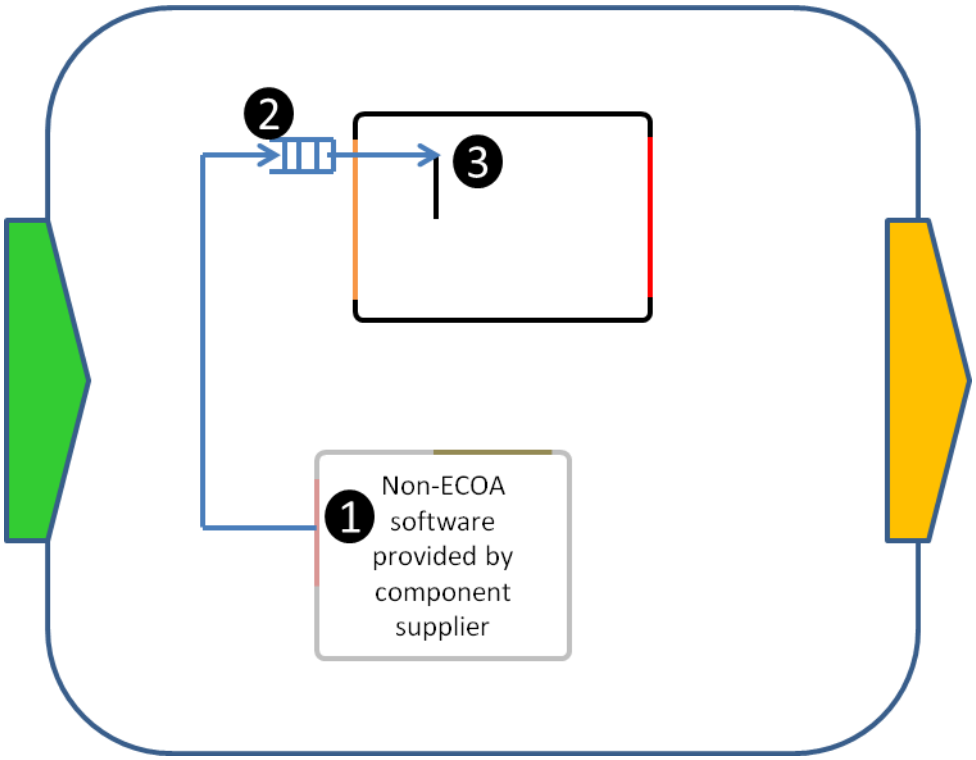


Figure 21 Driver Component Example – External Asynchronous Interaction

Figure 21 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 Services

8.1 Overview

Services provided by a Component instance are considered technically available at any time. However, as specified in section 9.1, the ECOA Infrastructure discards any operation arriving to a Module Instance when that Module Instance is not in the **RUNNING** state.

The functional availability of a provided Service is left under the responsibility of the Provider component: it is likely to be dependent upon a combination of component internal logic and the functional availability of any required Services necessary in order to successfully provide its Service.

Note: the functional availability of a provided Service may be conveyed to client ASCs by expressing it as a Service Operation (e.g. a Versioned Data).

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 is static, and a Client has only 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.

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) and a single Provider of the Service (identified as a target in the schema).

There may be multiple Requirers of the same Service; the connections between them are determined by the Wires in the Assembly Schema

8.2 Service Link Behaviour

8.2.1 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 22.

[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.

Figure 22 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.

8.2.2 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).

Similarly, when a Service Definition includes an Event Received By Provider operation, the associated typed data sent by any Requirer is received by the Provider.

When a Service Definition includes a Versioned Data operation, the 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 Provider.

For a Request-Response operation referenced in a Service Link, the Request from the Client is addressed (directed) to the Provider (Server).

These behaviours are summarized in Table 1.

Table 1 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 Provider defined in the Assembly Schema.
	received_by_provider	Receives the Event and its associated typed data from all Requirers.	The Event and its associated typed data is sent to the Provider
Request-Response Operations		Receives Requests from all Requirers (Clients)	Provider as defined in the Assembly Schema.
Versioned Data Operations		Updates data for all Requirers (Readers)	Provider (Writer) as defined in the Assembly Schema.

The behaviour above is true when the server is technically accessible. Examples of situations where the server is not technically accessible are: loss of connection between Platforms, remote Platform down, server Module Instance not in RUNNING state. When the server is not technically accessible the following behaviour is applied:

- Event
  - The event is discarded by the ECOA Infrastructure
- Request-Response
  - NO_RESPONSE returned by the ECOA Infrastructure to the client as soon as possible or after the Request Response timeout set by the ASC supplier has expired.
- Versioned data
  - The versioned data is always accessible by the writer but any publication may not actually be distributed to the clients.
  - The latest value of the versioned data is always accessible by the reader, but the versioned data may become stale.

8.2.3 Examples

Note: this section illustrates queues at ASC level for conceptual purpose only. Actual queues are implemented at Module level within ASCs.

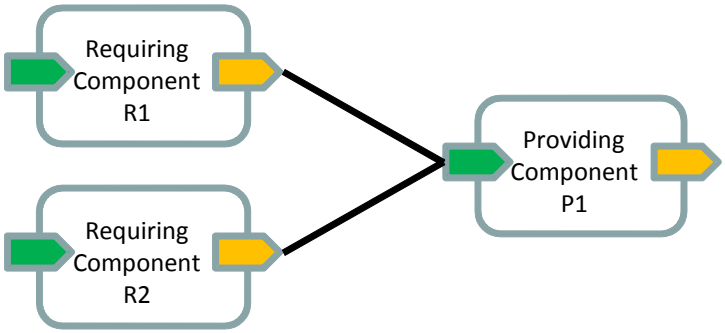


Figure 23 Example Assembly Schema

In the above Assembly Schema, two Application Software Components R1 and R2 are connected, via Service Links, to one Application Software Component P1. P1 is the Service Provider for R1 and R2.

"Sent by Provider" Events

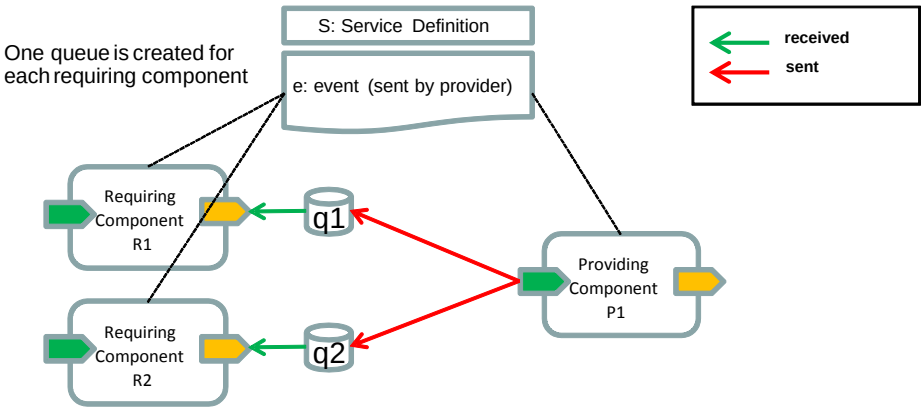


Figure 24 Generation of an Event

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

- An Event queue (q1,q2) is created for each Requiring Component (R1, R2);
- An Event sent by the Service Provider (P1) is received by all its Requirers (R1 and R2).

"Received by Provider" Events

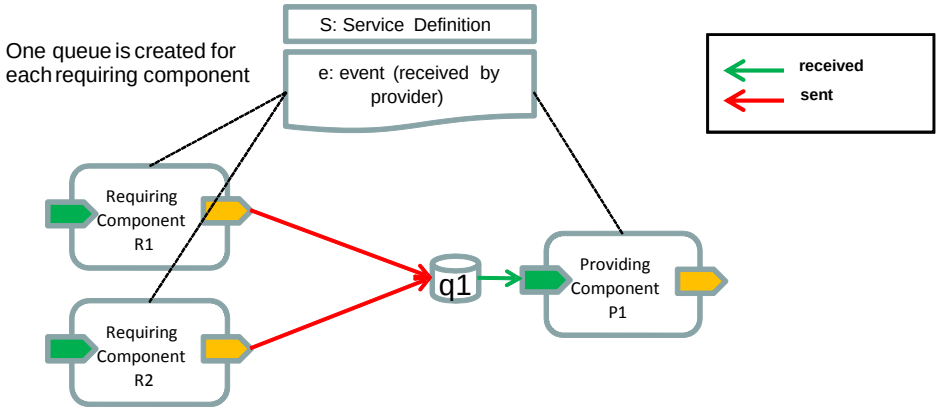


Figure 25 Consumption of an Event

An Event "received by Provider" in a Service translates to:

- An Event queue (q1,q2) is created for the providing Component (P1);
- An Event sent by a Requirer (R1, R2) is received by the Service Provider (P1)
- All Requirers are allowed to send the Event.

Request-Response

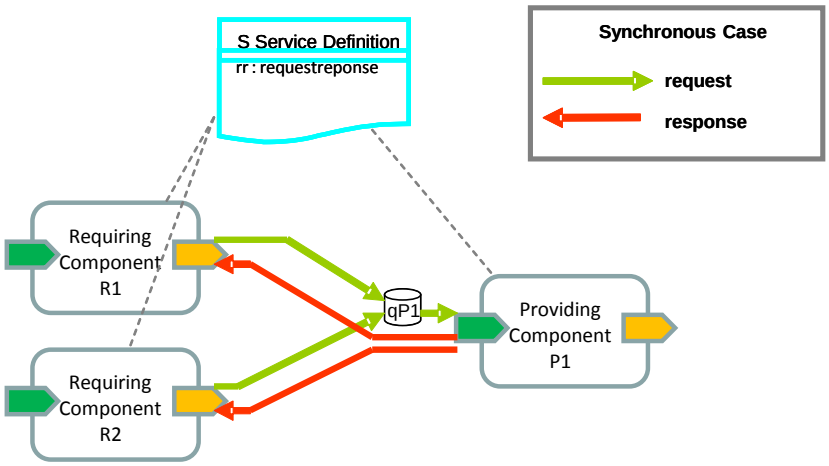


Figure 26 Synchronous Request-Response Operation

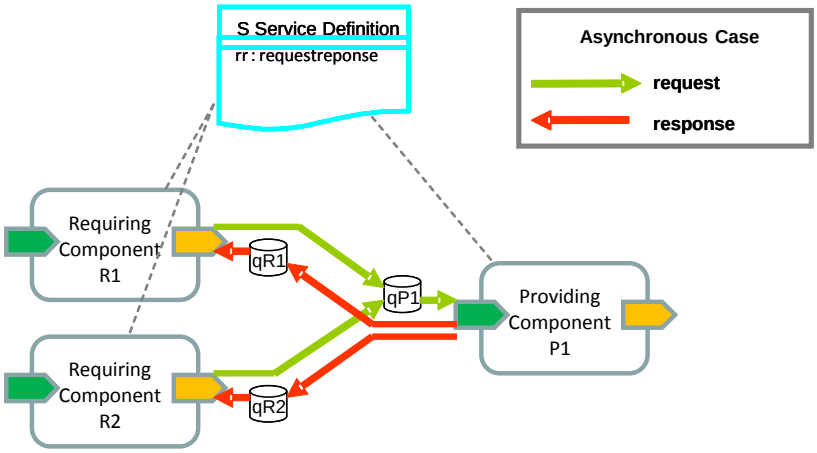


Figure 27 Asynchronous Request-Response Operation

In Figure 26 and Figure 27, Service definition S defines a single Request-Response operation (rr). This translates to:

- Service Requirers (R1 and R2) issue a Request to their Provider (P1).
- The Service Provider (P1) responds 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 the Provider are queued (qP1) 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) 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

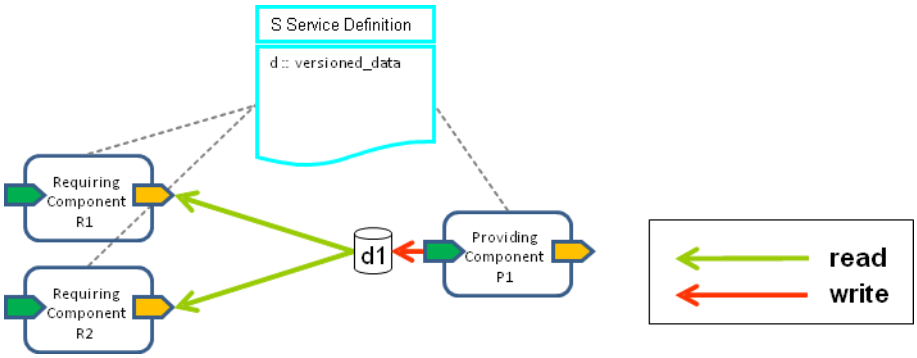


Figure 28 Versioned Data

In Figure 28 Service definition S defines one Versioned Data operation (d). This translates to:

- One instance of data[2] (d1) created
- The Versioned Data instance is written by only one Application Software Component and can be read by many Application Software Components.

Note: as specified in §7.5, Versioned Data across service links can only be done with access control.

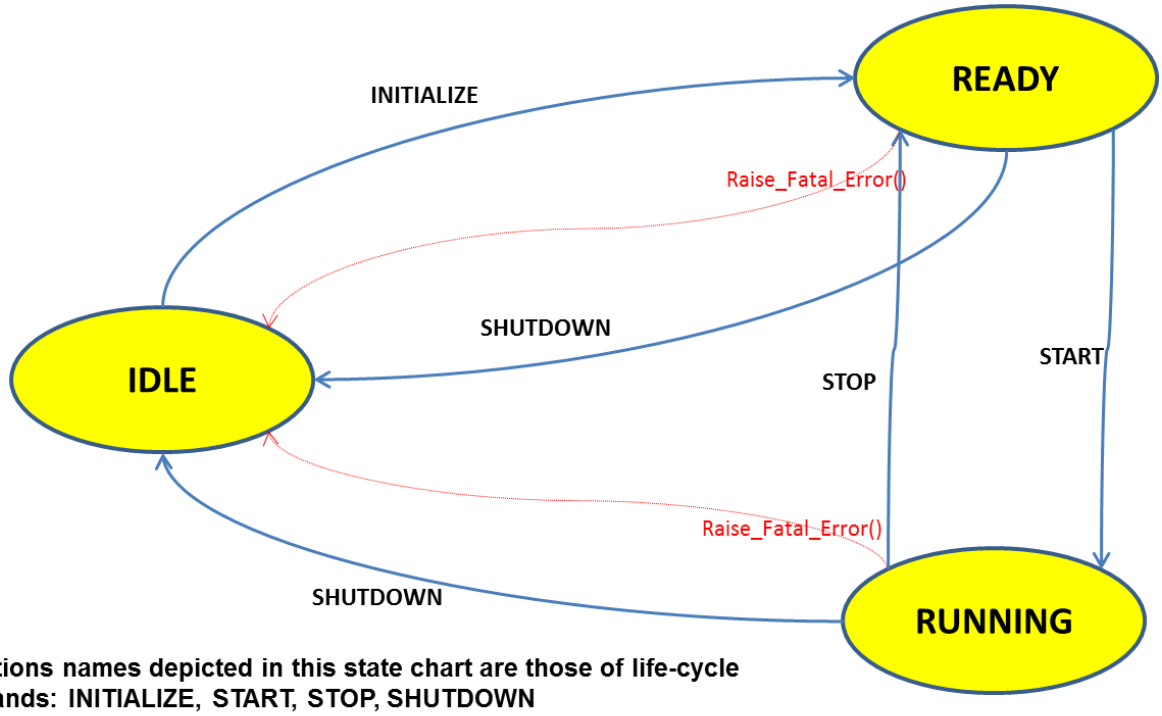
9 ECOA System Management

9.1 Lifecycle

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

Figure 29 illustrates the Module Runtime Lifecycle.

[2] 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.



Transitions names depicted in this state chart are those of life-cycle commands: INITIALIZE, START, STOP, SHUTDOWN
Corresponding Module entry-points names are the following: INITIALIZE__received, START_received, STOP_received, SHUTDOWN_received

A call from the Module Instance to the Raise_Fatal_Error() container function will set it into the IDLE state, as well as all other Module Instances in the same ASC.

Figure 29 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 29. 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 necessarily 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 (such as upon ECOA System start up / shutdown, or upon a recovery action triggered by the ECOA Fault Handler). They are:

- INITIALIZE
- START
- STOP

- SHUTDOWN

The lifecycle of Module Instances within a Component instance are managed by the ECOA Infrastructure.

9.1.1 Module Startup

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

Component Instances may be started in any order or even in parallel; however the following describes the startup order of the Module Instances within each Component Instance:

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

Then, with the help of Lifecycle Events, the ECOA Infrastructure requests all Modules Instances of the Component Instance to **INITIALIZE** in the following sequential order:

- All Dynamic Triggers,
- Then all periodic Triggers,
- Finally all other Module Instances. Each Module Instance performs any actions required to initialize such as allocating further resources, setting its internal variables and/or reading its Properties to reach a technically coherent and initialized internal state.

When all Module Instances of the Component Instance are in **READY** state, the ECOA Infrastructure requests all Module Instances of the Component Instance to **START** in the following sequential order:

- All Dynamic Triggers,
- Then all periodic Triggers,
- Finally all other Module Instances. At this point, each 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.

Note: Periodic Triggers and dynamic Triggers should be started first because the overhead of starting triggers is more controllable than normal modules.

At this stage, all types of Module Instance within the Component Instance are in RUNNING state.

Note: further functional initialization of the ECOA System may take place once Module Instances have reached the RUNNING state (however from the module lifecycle point of view, the technical initialization of Module Instances is considered to be complete at this stage).

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

- the ECOA Infrastructure requests a Module Instance lifecycle state change
- the Container invokes the appropriate entry point in the Module Interface
- when the entry point returns the Container changes the state of the Module Instance

9.1.2 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 29, are discarded, and the fault management Infrastructure will be notified (see section 9.3).

Lifecycle Events are activating operations.

Lifecycle Events have no priority over other operations:

- Operations arriving before the **START** entry-point has returned are discarded, as illustrated in Figure 30.

- The ECOA Infrastructure discards any operation (except legal lifecycle events) arriving to a Module Instance when that Module Instance is not in the **RUNNING** state (i.e the operation is not queued), as illustrated on Figure 30 and Figure 31.

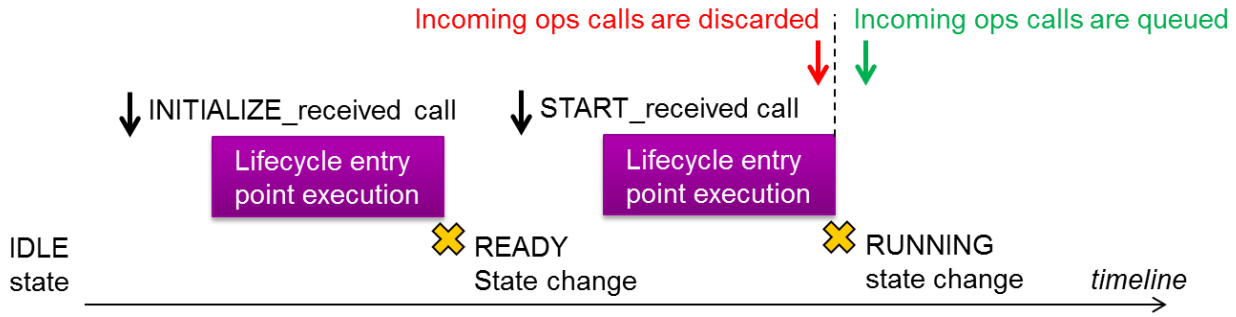


Figure 30 Module Run-time Behaviour – Startup

- On receipt of **STOP** Events, operations already executing are allowed to complete and operation calls may continue to be queued but are finally discarded as soon as the **STOP** or **SHUTDOWN** entry point has been executed (i.e. once the Module is not in **RUNNING** state anymore, the Module queue is cleared).

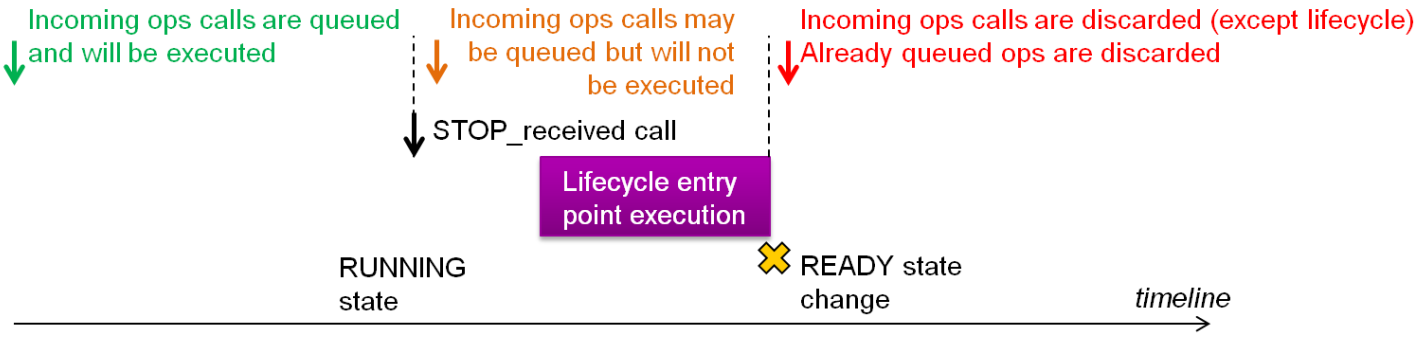


Figure 31 Module Run-time Behaviour – Stop

- Operations arriving after receipt of **SHUTDOWN** Events (whilst in RUNNING) may be queued, but are finally discarded as soon as the **SHUTDOWN** entry point has been executed (i.e. once the Module is not in **RUNNING** state anymore, the Module queue is cleared), as illustrated in Figure 32.

- Operations arriving whilst in READY continue to be discarded and the SHUTDOWN Event does not change this, as illustrated in Figure 32.

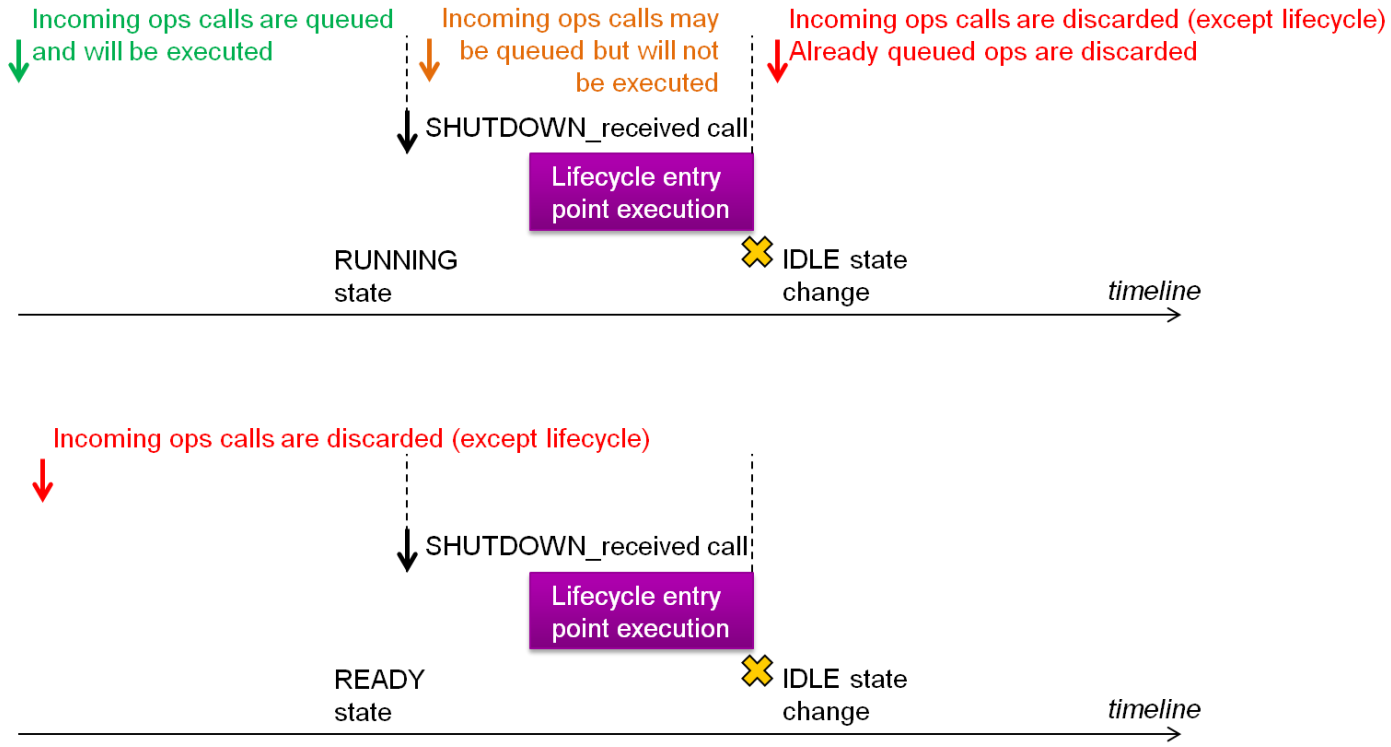


Figure 32 Module Run-time Behaviour – Shutdown

A Module should not 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.

9.1.3 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.

9.1.4 Graceful vs fast shutdown

9.1.4.1 Graceful shutdown procedure

The ECOA Infrastructure may call the **STOP** and **SHUTDOWN** entry points of Module Instances for the purpose of performing a “graceful shutdown” procedure.

A graceful shutdown may be useful for simulation purposes in order to be able to stop, shutdown and restart selected ASCs and let them perform user-specified code as part of these entry points. It may also be

useful for driver components so that these can free any specific underlying resource before becoming **IDLE**.

ECOA itself does not mandate ECOA Platforms to be capable of graceful shutdown. In order for a graceful shutdown procedure to be possible, the capability for it needs to be procured within the ECOA Platform. In addition, the functional specifications of targeted ASCs need to tell the ASC supplier what application code to implement in the **STOP** and **SHUTDOWN** entry points.

9.1.4.2 Fast shutdown procedure

As part of some recovery actions (see section 9.3.2.2), the ECOA Infrastructure may perform a "fast shutdown" procedure of Module Instances, by putting them into **IDLE** state without invoking their **STOP** or **SHUTDOWN** entry points.

9.2 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.

9.3 Fault Handling

9.3.1 Error Categorization

Error is a global term that encompasses:

- Application errors: consequences of faults occurring at application 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. Their management is the responsibility of the system architect and system integrator.

9.3.2 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).

The ECOA Infrastructure associates a timestamp to any error that it raises to ECOA Fault Handler(s).

9.3.2.1 Application Errors Propagation and Recovery Actions

Application errors may be detected by an application Module. The Component provider determines the recovery actions (if any) to be applied by an application 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 the Fault Handler through *raise_error*.
  - The Fault Handler will determine the Recovery Action.
- In case of fatal application error, the application Module uses the *raise_fatal_error* API.
  - The error is automatically sent to the Fault Handler
  - All application Modules within the faulty component are immediately placed into the IDLE state by the ECOA Infrastructure.
  - The Fault Handler will determine the Recovery Action to be applied, if any.

Figure 33 illustrates the propagation of Non-fatal application errors.

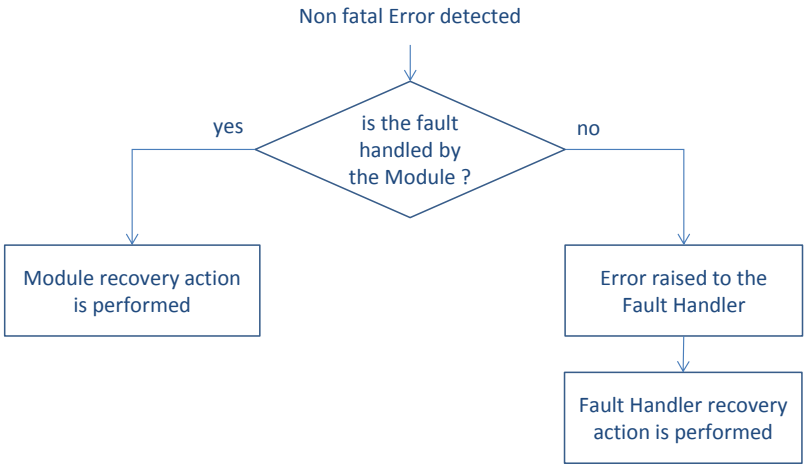


Figure 33 Propagation path of non-fatal application error

Figure 34 illustrates the propagation of fatal application errors.

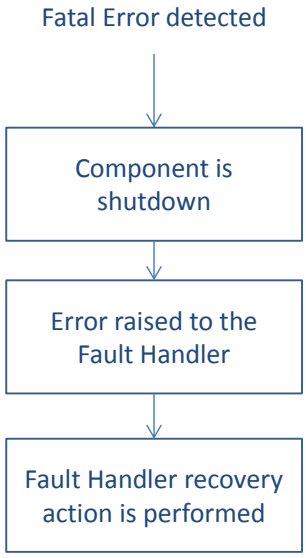


Figure 34 Propagation path of fatal application errors

9.3.2.2 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 or Computing Platform level. Figure 35 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.

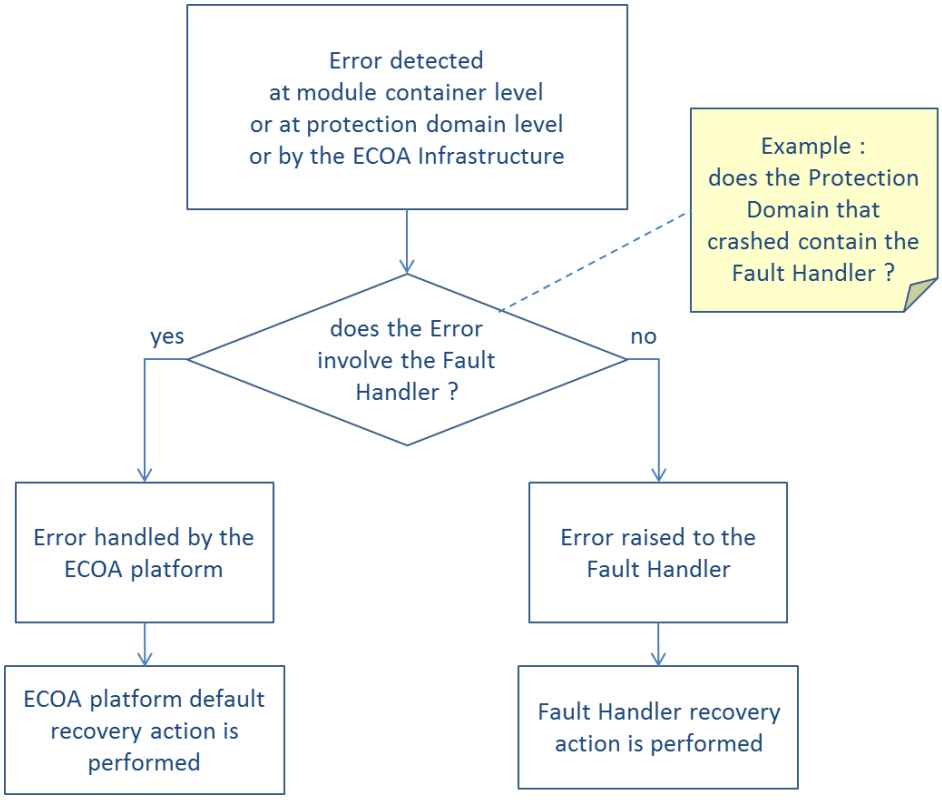


Figure 35 Propagation path of infrastructure errors

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, depending on the capabilities procured in the ECOA Platform:

- 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

Note: the warm restart recovery actions supported by the ECOA Platform should be consistent with that ECOA Platform ability to maintain the warm start context of Module Instances upon those recovery actions.

The component shutdown action consists of the Infrastructure putting all Module Instances of the targeted Component Instance in idle state (with the Trigger Instances being the first). The Infrastructure frees

associated resources, without calling the shutdown entry point of these potentially faulty Module Instances. Therefore, this is a fast shutdown and not a graceful functional shutdown and is not part of the Module Lifecycle.

The component restart action consists of the Infrastructure triggering a component shutdown action, followed by initialize and start lifecycle actions according to orders defined in 9.1.1.

In order to enable fault filtering, the Fault Handler has a context which can be used as memory to store the number of occurrences of a fault over a time frame. As an example, this could be used to enable a sporadic fault to be distinguished from a recurrent fault and applying a recovery action only if a given fault occurs more than X times in a given time frame.

Figure 36 summarizes the error propagation path and applicable recovery actions.

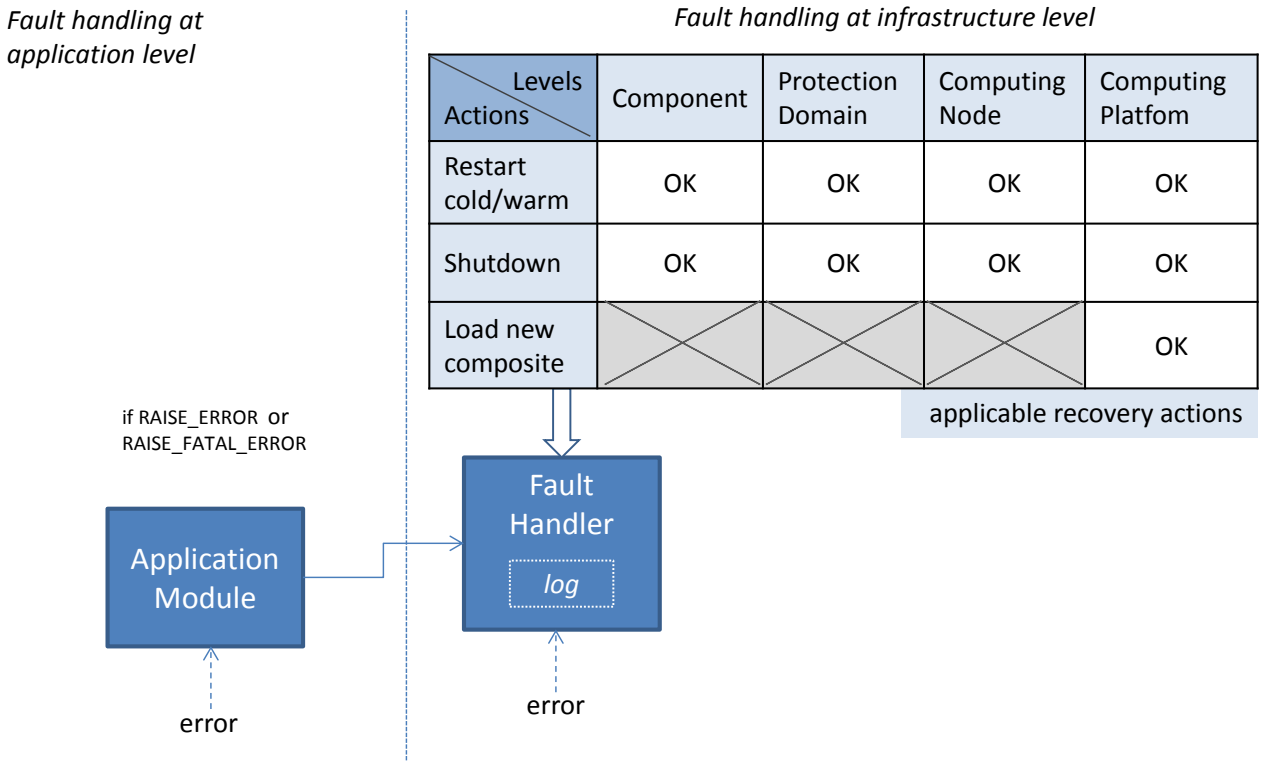


Figure 36 Error propagation path

9.3.3 Operations and faults

ECOA provides the component supplier with three types of operations for use in application modules:

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

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

The following section provides Infrastructure errors handled by the Fault Handler. Error codes returned to the Module are provided in Architecture Specification Part 4.

Note: Figures in this section provide examples of infrastructure error propagation paths to the Fault Handler. Actual paths may depend on Platform implementation.

Event:

- 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 37 illustrates these behaviours.

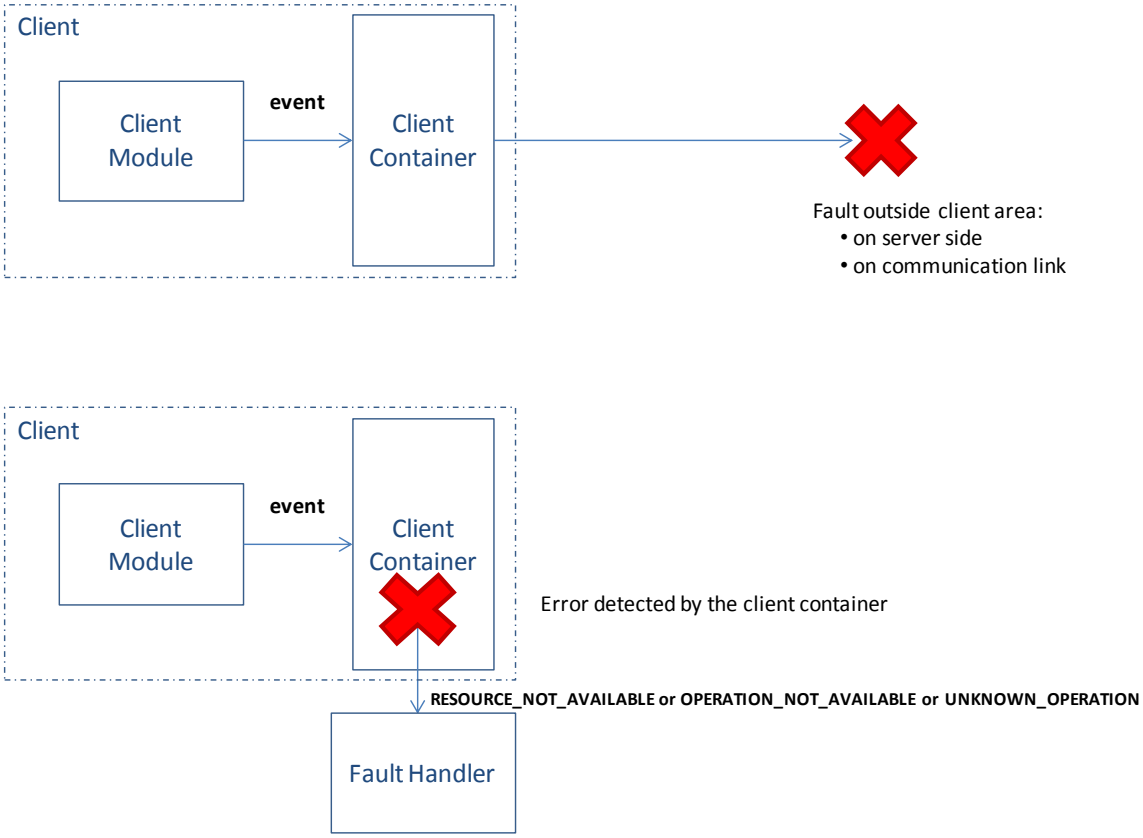


Figure 37 Event faults propagation behaviour

Request-Response:

- Request faults:
  - 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:
  - 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 38 and Figure 39 illustrate these behaviours.

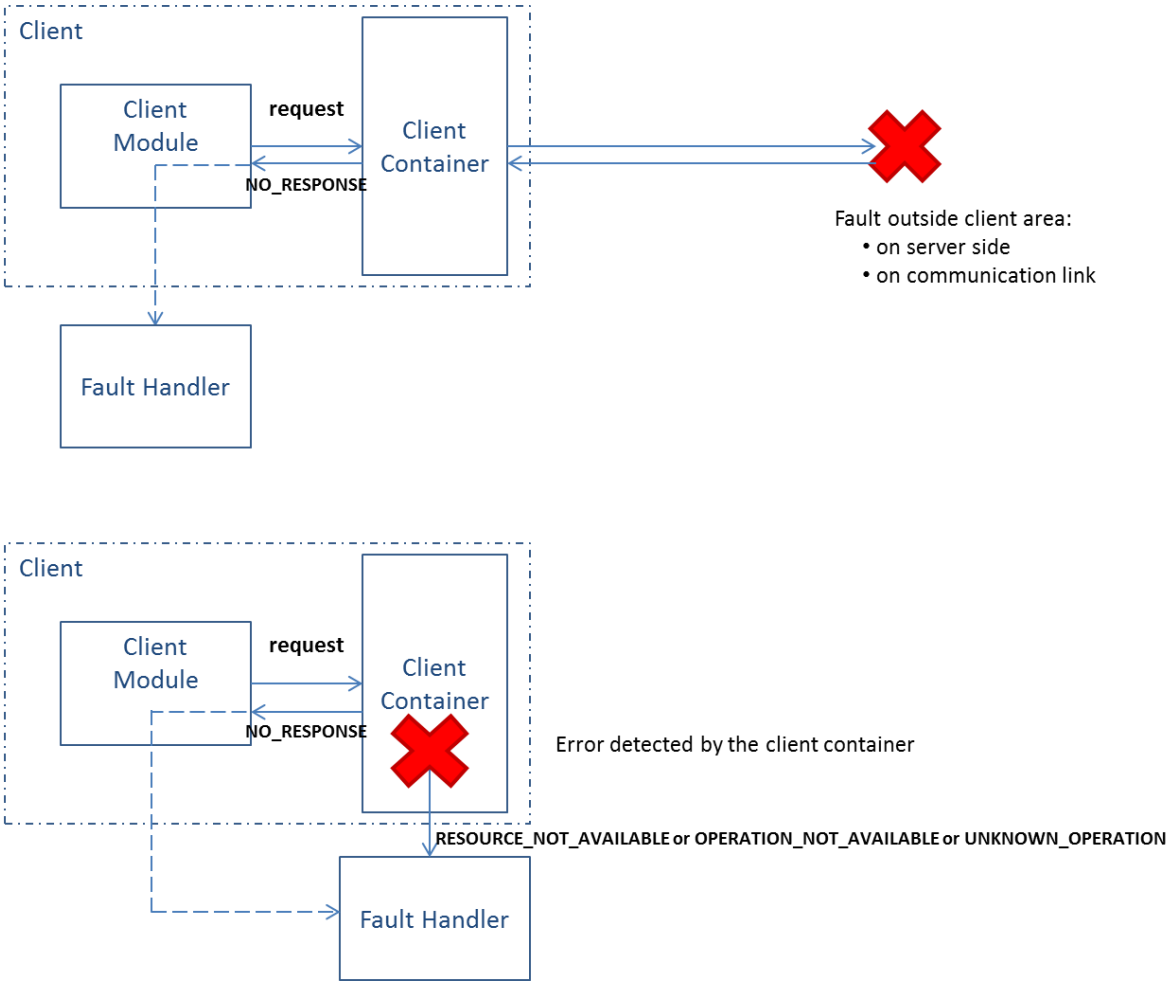


Figure 38 Request-Response faults propagation behaviour part 1

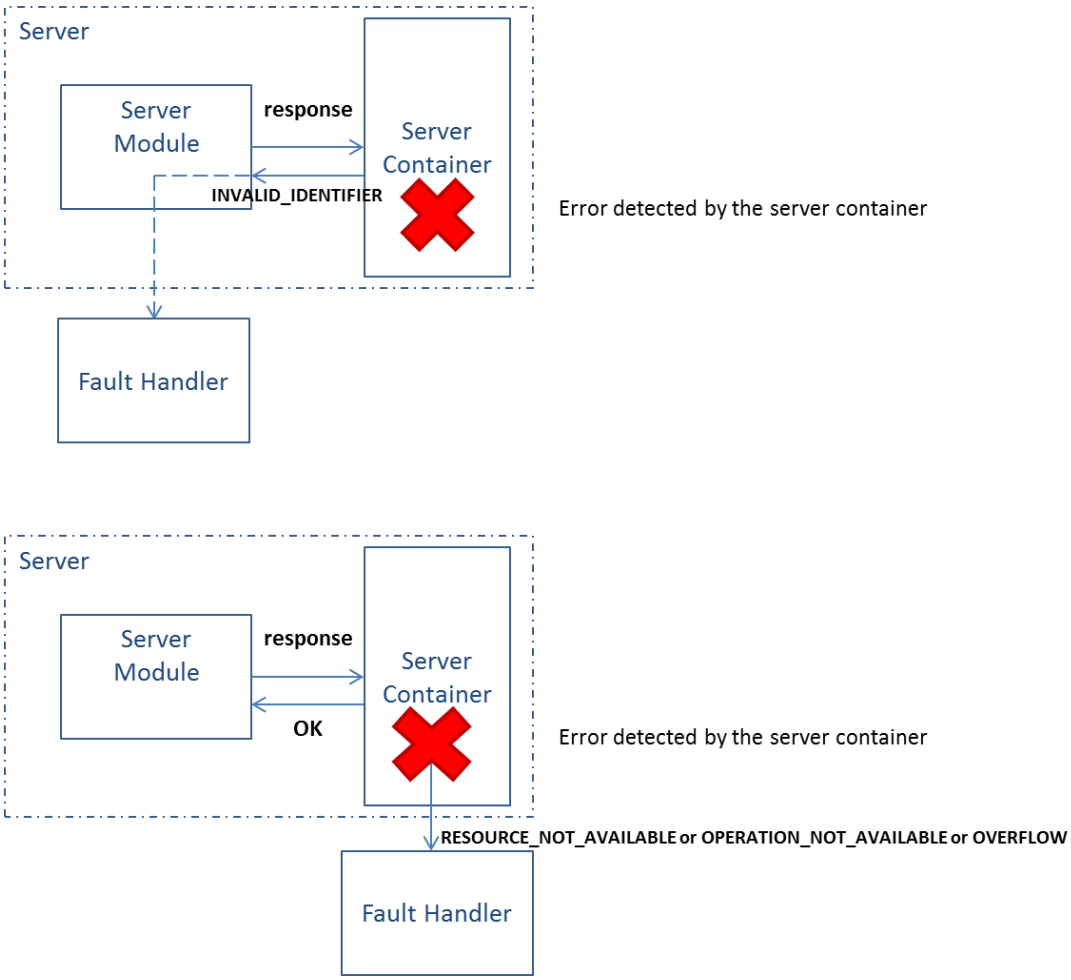


Figure 39 Request-Response faults propagation behaviour part 2

Note that in the case of a NO_RESPONSE error being detected by the client Container it is possible that two recovery actions are performed as a result of a single originating fault. This is due to the fact that the Infrastructure error is handled by the Fault Handler, but in addition, it is also possible that the Module raises an error to the Fault Handler.

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 Infrastructure, or do nothing and wait for the timeout (if specified) 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:
  - Infrastructure error handled by the Fault Handler:
    - RESOURCE_NOT_AVAILABLE client container unable to access the memory slot
- Release_read_access

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

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

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

- Cancel_write_access:
    - No Infrastructure error

The Figure 40 illustrates these behaviours.

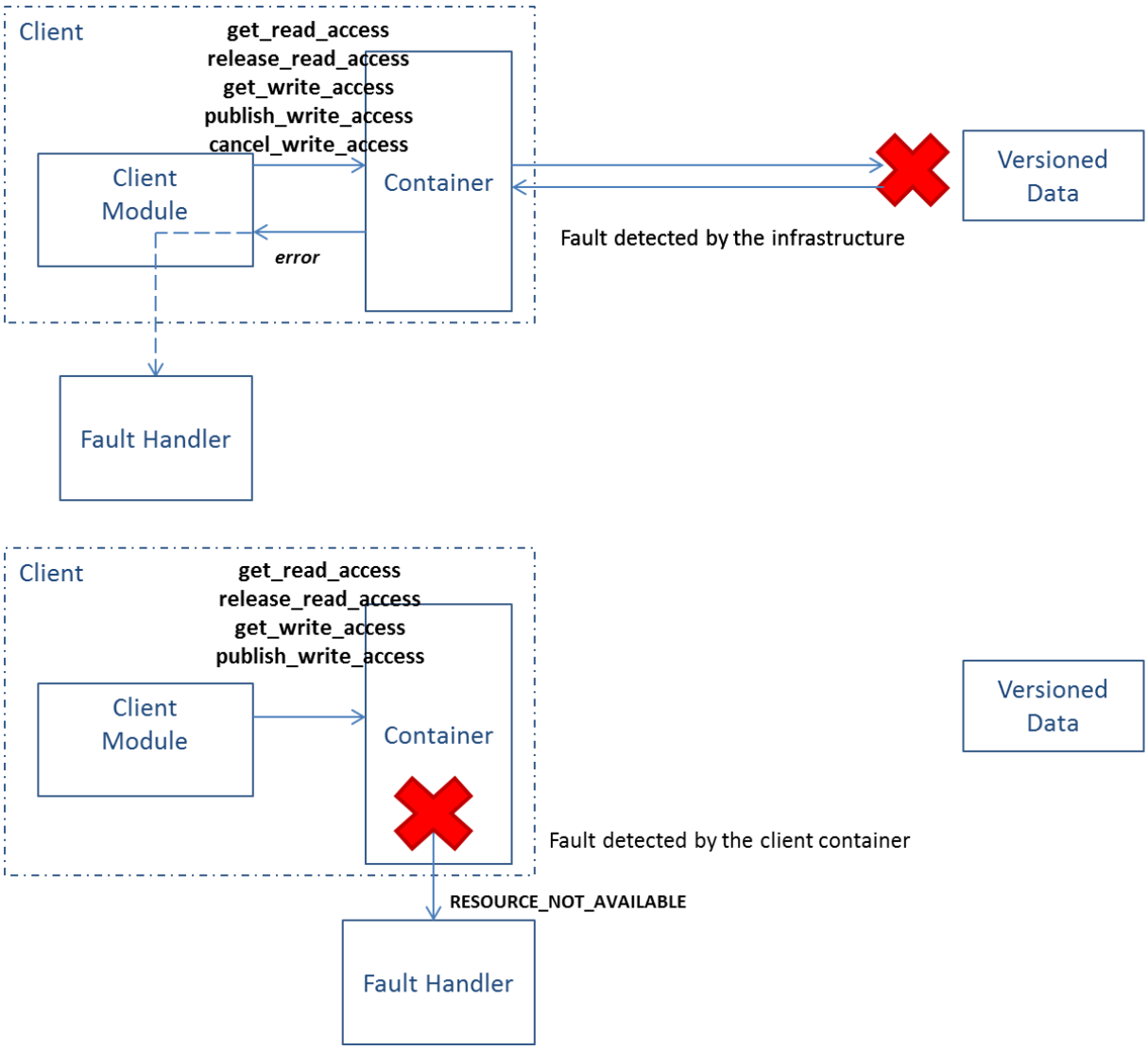


Figure 40 Versioned Data faults propagation behaviour

9.4 Module Context

When a module is not stateless, its source code should save states using the optional User Context or Warm Start Context within the Module Context, and should not use global variables. This enables a platform to instantiate a single module implementation multiple times and manage instance specific data contained in the Module Context.

User Context is instance specific optional 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.

Warm Start Context is instance specific optional 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. 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 ability to maintain the warm start context of Module Instances upon warm start recovery actions is a Platform procurement requirement. This is aimed at tailoring ECOA Platforms complexity to actual procurement needs. As previously stated, the warm restart recovery actions supported by an ECOA Platform should be consistent with that ECOA Platform ability to maintain the warm start context of Module Instances upon those recovery actions. In this section it is being assumed that this consistency is fulfilled.

Example: if a Platform does not support warm restart recovery actions at Computing Node level and Computing Platform level, the Platform does not need to save the warm start context in a way that would support these recovery actions.

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 Module Instance entry point.

Note: the ASC supplier shall initialize the User Context, if any, every time the Initialize Module Instance entry point has been called, and shall never assume the User Context has been maintained by the Infrastructure.

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 41 shows how the Warm Start Context is managed through these transitions.

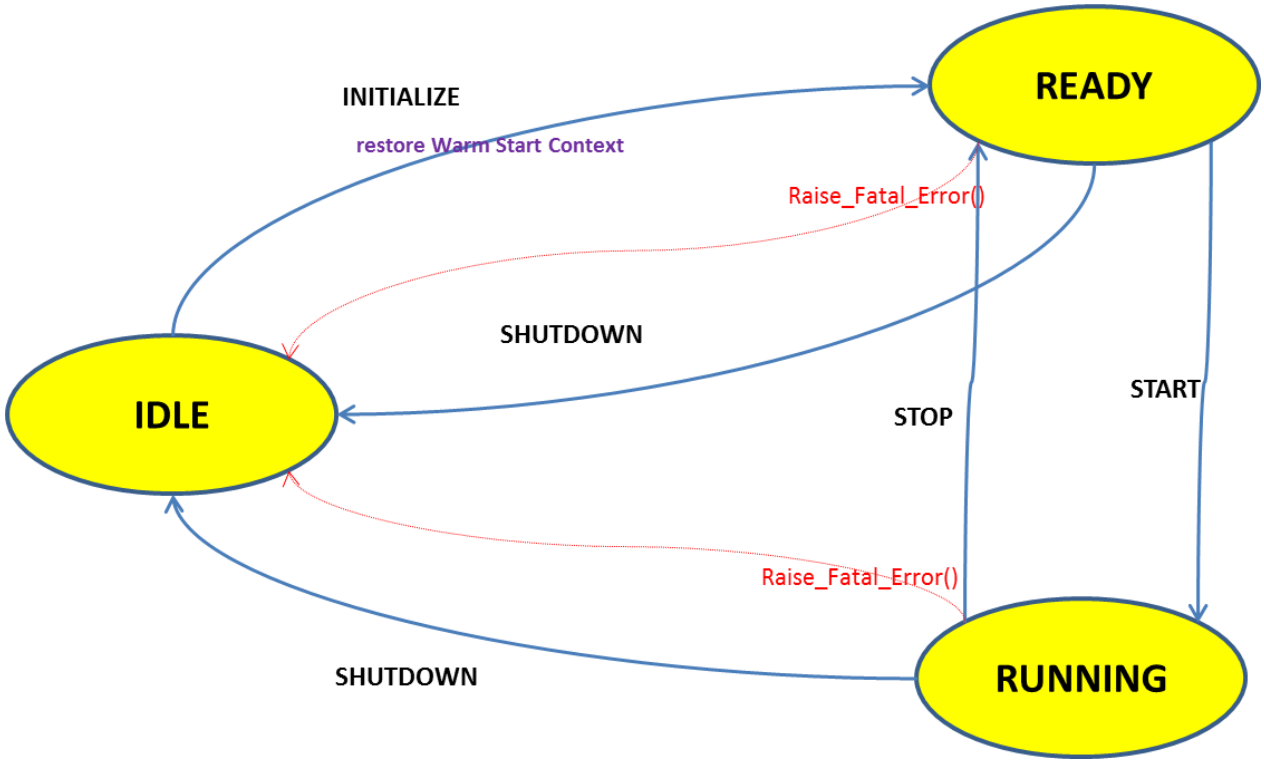


Figure 41 Management of Module Context

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.

User Context and Warm Start Context are optional. They are only necessary for Module Instances maintaining an internal state between operation calls. Metamodel attributes on a Module Type (in an ASC Implementation) are used to specify whether the User Context and/or Warm Start Context APIs should be made available for any Module Instances of that Module Type.

10 Scheduling

10.1 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.

10.2 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 that 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. 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 §7.5.3).

11 Module Operation Link Behaviour

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

Figure 42 shows possible interactions between Service Operations and Module Operations.

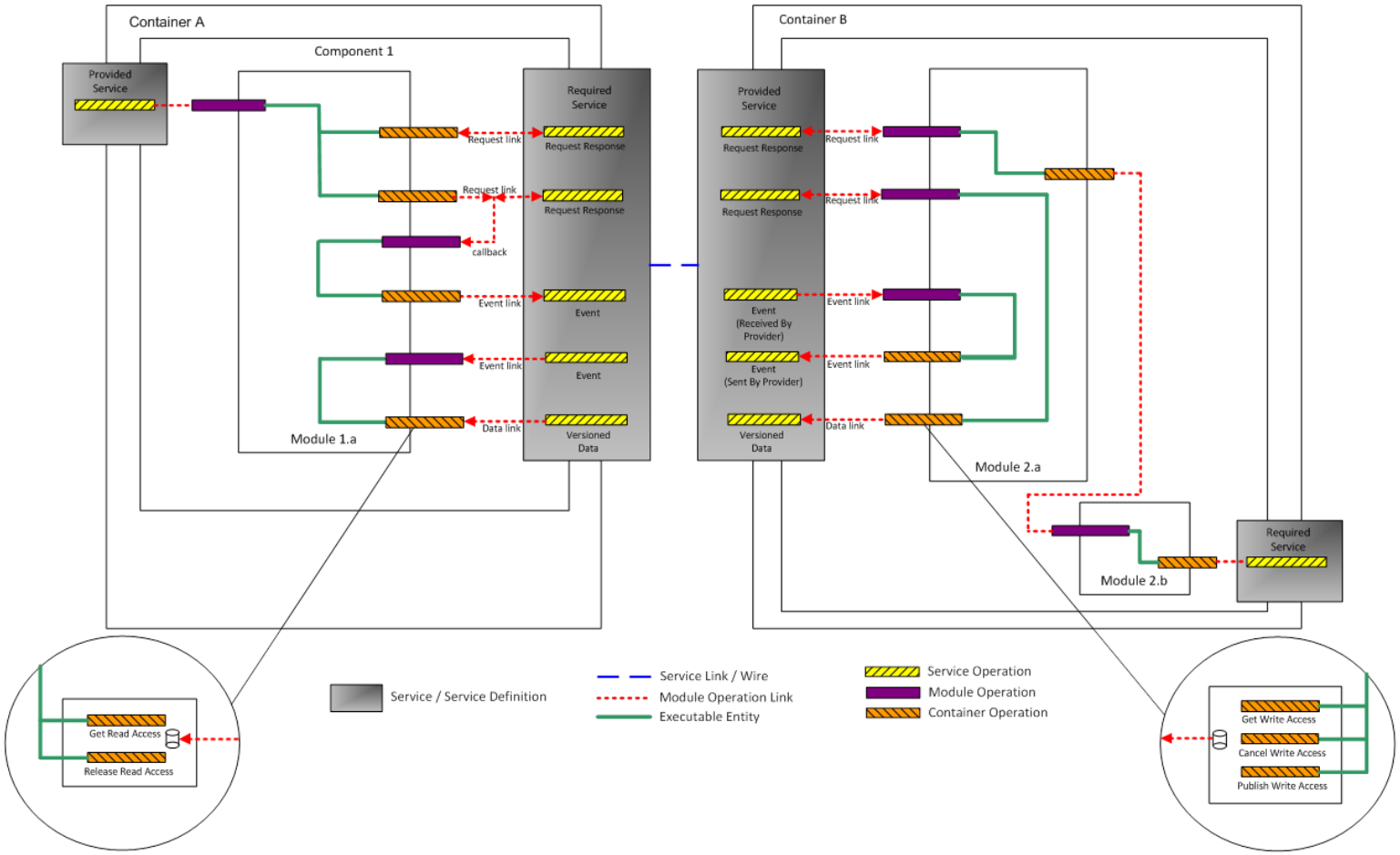


Figure 42 Interactions between Service Operations and Module Operations

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.

12 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.

13 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 and can be found in the Architecture Specification Part 6.

14 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 the specification of Initial Assembly Schemas made of hierarchical composites containing ECOA components in order to help with design abstractions.

Figure 43 shows an example Composite constructed with four Components.

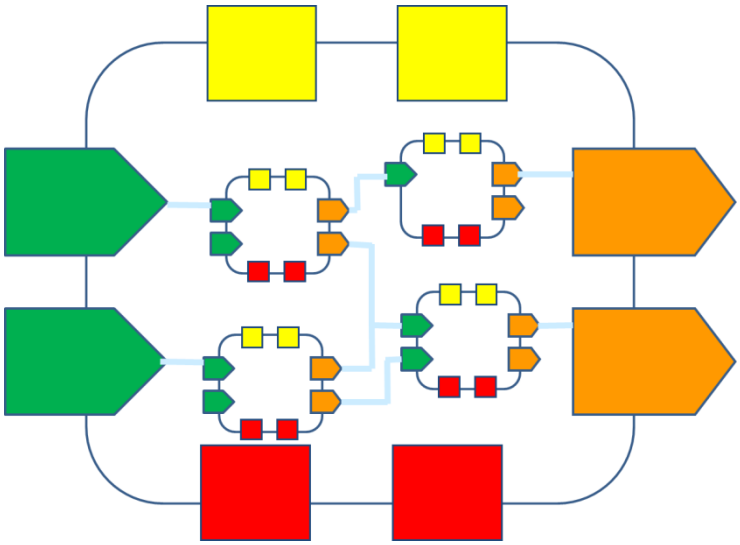


Figure 43 A Composite

15 Use of Composites in Platform Integration

The Final Assembly Schema used for integration on a given ECOA Platform is a flattened single Composite that contains as a minimum all ECOA Components that are deployed onto that platform. Interactions with parts of the ECOA System that are deployed onto other platforms are taken into account in the Final Assembly Schema by representing them as Components. This representation is a local view to the particular platform and may not correspond to the actual assembly on those remote platforms.

Parts of an ECOA System that are deployed onto multiple ECOA platforms can be modelled through Composites deployed onto those platforms. Interactions between these Composites that rely on inter-platform messages using the ELI can be modelled using the Cross Platform View [Architecture Specification Part 7]. This allows the capture of the functional and logical interfaces between platforms in a way which is independent of the final assembly on each platform.

.