INTERNATIONAL STANDARD

ISO/IEC 23004-6

First edition
2008-02-15

Information technology — Multimedia Middleware —

Part 6: Fault management

Technologies de l'information — Intergiciel multimédia —

Partie 6: Gestion des anomalies

Reference number
ISO/IEC 23004-6:2008(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword iv

Introduction v

1 **Scope** 1

2 **Normative references** 1

3 **Terms and definitions** 1
3.1 **Specification terms and definitions** 1
3.2 **Realization terms and definitions** 6

4 **Abbreviated terms** 6

5 **Overview of interface suites** 6

6 **Fault Management interface suites** 8
6.1 **Interfaces for controlling Fault Management** 8
6.2 **Interfaces for Fault Management coordination** 15

7 **Realization overview** 23
7.1 **Introduction** 23
7.2 **Concepts and terminology** 24
7.3 **Assumptions and capabilities** 25
7.4 **Entities and responsibilities** 26

8 **Fault Management realization** 26
8.1 **Initial Situation** 26
8.2 **The basic principle** 27
8.3 **Roles** 28
8.4 **Interaction with the System Integrity Management Framework** 35

Annex A (informative) **Fault tolerance techniques** 36

Annex B (informative) **Example scenarios** 42

Bibliography 44

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 23004-6 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 23004 consists of the following parts, under the general title *Information technology — Multimedia Middleware*:

— *Part 1: Architecture*

— *Part 2: Multimedia application programming interface (API)*

— *Part 3: Component model*

— *Part 4: Resource and quality management*

— *Part 5: Component download*

— *Part 6: Fault management*

— *Part 7: System integrity management*

Introduction

The Multimedia Middleware (M3W) Fault Management Framework provides mechanisms for adding fault tolerance techniques (e.g. error detection and recovery) to an existing system. These techniques deal with errors occurring in Service Instances created from Components that are not trusted, without modifying these Components and without relying on reflection. The Fault Management Framework is restricted to the definition of these mechanisms; the actual fault tolerance techniques are system dependent and outside its scope.

Fault management is done by means of intercepting and redirecting Service instantiation and interface method invocations by inserting a new entity in-between, called the "Middleman". The Fault Management Framework also defines a Fault Manager for coordinating several Middlemans[1]. Collectively, the Middlemans and Fault Managers provide the fault tolerance techniques.

Finally, the Fault Management Framework may interact with the (remote) System Integrity Management Framework in order to support fault removal.

The purpose of the Fault Management Framework is to enable component-based systems to operate failure-free despite the presence of faults in their Components (fault-tolerance). For a running system, fault-tolerance is based on error detection, confinement, and recovery. The Fault Management Framework aims to (enable) stopping error propagation by handling the detected errors and not letting them lead to a failure.

The Fault Management Framework provides mechanisms to add fault management to existing systems and Components. It does not address the design and implementation of new Components that may incorporate some fault management techniques such as those found in fault-tolerant systems.

The M3W Fault Management Framework should allow a large number of known or future fault-tolerance techniques to be incorporated.

1) The word “Middlemans” is used as the plural of Middleman, since a Middleman is not a human.

Information technology — Multimedia Middleware —

Part 6:
Fault management

1 Scope

This part of ISO/IEC 23004 defines the MPEG Multimedia Middleware (M3W) technology Fault Management Architecture. It contains the specification of the part of the M3W application programming interface (API) related to Fault Management as well as the realization. The M3W API specification provides a uniform view of the Fault Management functionality provided by M3W. The specification of the realization is relevant for those who are making an implementation of a Fault Management framework for M3W.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 23004-1, *Information technology — Multimedia Middleware — Part 1: Architecture*

ISO/IEC 23004-3, *Information technology — Multimedia Middleware — Part 3: Component model*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Specification terms and definitions

3.1.1
API specification
specification of a collection of software interfaces providing access to coherent streaming-related functionality

3.1.2
interface suite
collection of mutually related interfaces providing access to coherent functionality

3.1.3
logical component
coherent unit of functionality that interacts with its environment through explicit interfaces only

3.1.4
role
abstract class defining behavior only

3.1.5
role instance
object displaying the behavior defined by the role

3.1.6
attribute
instance variable associated with a role

NOTE Attributes are used to associate state information with roles.

3.1.7
signature
definition of the syntactic structure of a specification item such as a type, interface or function in IDL

NOTE For C functions, signature is equivalent to prototype.

3.1.8
specification item
entity defined in a specification

NOTE Data type, role, attribute, interface and function are examples of specification items.

3.1.9
qualifier
predefined keyword representing a property or constraint imposed on a specification item

3.1.10
constraint
restriction that applies to a specification item

3.1.11
execution constraint
constraint on multi-threaded behavior

3.1.12
model type
data type used for specification (modeling) purposes only

NOTE Set, map and entity are examples of model types.

3.1.13
model constant
constant used for specification (modeling) purposes only

3.1.14
enum element type
enumerated type whose values can be used to construct sets (bit vectors) of at most 32 values by logical or-ing

3.1.15
enum set type
32-bit integer data type representing sets of enumerated values

3.1.16
set type
data type whose values are mathematical sets of values of a specific type

NOTE Unlike enum sets, these sets may be infinite.

3.1.17
map type
data type whose values are tables mapping values of one type (the domain type) to values of another type (the range type)

NOTE Maps are a kind of generalized array.

3.1.18
entity type
class of objects that may have attributes associated with them

3.1.19
interface-role model
extended Unified Modeling Language class diagram showing the roles and interfaces associated with a logical component, and their mutual relations

3.1.20
logical component instance
incarnation of a logical component: a configuration of objects displaying the behavior defined by the logical component

3.1.21
provides interface
interface that is provided by a role or role instance

3.1.22
requires interface
interface that is used by a role or role instance

3.1.23
specialization
behavioral inheritance
definition, by a role, of behavior which implies the behavior defined by another role

NOTE A role S specializes a role R if the behavior defined by S implies the behavior defined by R, i.e. if S has more specific behavior than R.

3.1.24
diversity
set of all parameters that can be set at instantiation time of a logical component, and that will not change during the lifetime of the logical component

3.1.25
mandatory interface
provides interface of a role that should be implemented by each instance of the role

3.1.26
optional interface
provides interface of a role that need not be implemented by each instance of the role

3.1.27
configurable item
parameter that can be set at instantiation time of a logical component, usually represented by a role attribute

3.1.28
diversity attribute
role attribute that represents a configurable item

3.1.29
instantiation
process of creating an instance (an incarnation) of a role or logical component

3.1.30
initial state
state of a role instance or logical component instance immediately after its instantiation

3.1.31
observable behavior
behavior that can be observed at the external software and streaming interfaces of a logical component

3.1.32
function behavior
behavior of the functions in the provides interfaces of a role

3.1.33
streaming behavior
input-output behavior of the streams associated with a role

3.1.34
active behavior
autonomous behavior that is visible at the provides and requires interfaces of a role

3.1.35
instantiation behavior
behavior of a role at instantiation time of a logical component

3.1.36
independent attribute
attribute whose value may be defined or changed independently of other attributes and entities

3.1.37
dependent attribute
attribute whose value is a function of the values of other attributes or entities

3.1.38
invariant
assertion about a role or logical component that is always true from an external observer's point of view

NOTE In reality, the assertion may temporarily be violated.

3.1.39
callback interface
interface provided by a client of a logical component whose functions are called by the logical component

NOTE A notification interface is an example of this, but there may be other call-back interfaces as well, e.g. associated with plug-ins.

3.1.40
callback-compliance
general constraint that the functions in a callback interface should not interfere with the behavior of the caller in an undesirable way, such as by blocking the caller, or by delaying it too long

3.1.41
event notification
act of reporting the occurrence of events to 'interested' objects

3.1.42
event subscription
act of recording the types of event that should be notified to objects

3.1.43
cookie
special integer value that is used to identify an event subscription

NOTE Clients pass cookies to a logical component when subscribing to events. Logical components pass cookies back to clients when notifying the occurrence of the events.

3.1.44
event-action table
table associating events that can occur to actions that will be performed in reaction to the events

NOTE This is used to specify event-driven behavior.

3.1.45
non-standard event notification
event notification that is accompanied by other actions (such as state changes of the notifying logical component

3.1.46
client role
role modeling the users of a logical component

3.1.47
actor role
role (usually a client role) whose active behavior consists of calling functions in interfaces without any a priori constraints on when these calls will occur

3.1.48
control interface
interface provided by a logical component that allows the logical component's functionality to be controlled by a client

3.1.49
notification interface
interface provided by a client of a logical component that is used by the logical component to report the occurrence of events to the client

3.1.50
specialized interface
interface of a role R that is inherited from another role and is further constrained by R

3.1.51
precondition
assertion that should be true immediately before a function is called

3.1.52
action clause
part of an extended precondition and postcondition specification defining the abstract action performed by a function

NOTE The abstract action usually defines which variables are modified and/or which out-calls are made by the function.

3.1.53
out-call
An out-going function call of an object on an interface of another object

3.1.54
postcondition
assertion that will be true immediately after a function has been called

3.1.55
asynchronous function
function with a delayed effect

NOTE The effect of the function will occur some time after returning from the function call.

3.2 Realization terms and definitions

3.2.1
error
unwanted software state that is liable to lead to a failure

3.2.2
failure
event that occurs when the delivered service of a system deviates from correct service

3.2.3
fault
adjudged or hypothesized cause of an error

3.2.4
unit of failure
entity of the system on which a failure can be observed

3.2.5
unit of fault management
entity of the system that will be subject to Fault Management as a whole

3.2.6
middleman
Fault Management entity used for wrapping untrusted services

NOTE This wrapper can be used to insert fault tolerance mechanisms.

3.2.7
fault manager
Fault Management responsible for coordinating fault management between different middlemans in the system

3.2.8
fault management instantiation policy
extension to the runtime environment that enables instantiation of a middleman when an untrusted service is requested

4 Abbreviated terms

API Application Programming Interface
IDL Interface Definition Language
M3W Multimedia Middleware
OS Operating System
UML Unified Modeling Language
UoF Unit of Failure
UoFM Unit of Fault Management
USI Untrusted Service Instance
UUID Universally Unique Identifier

5 Overview of interface suites

This clause is informative. The M3W Fault Management Framework enables the addition of fault tolerance techniques to realization elements (service instances or compositions of service instances). This addition of fault tolerance techniques can be fully transparent for the applications and other middleware. In this case the

fault management interfaces would not be visible as part of the M3W API (see Figure 1 — Fault Management is fully transparent for applications and other middleware).

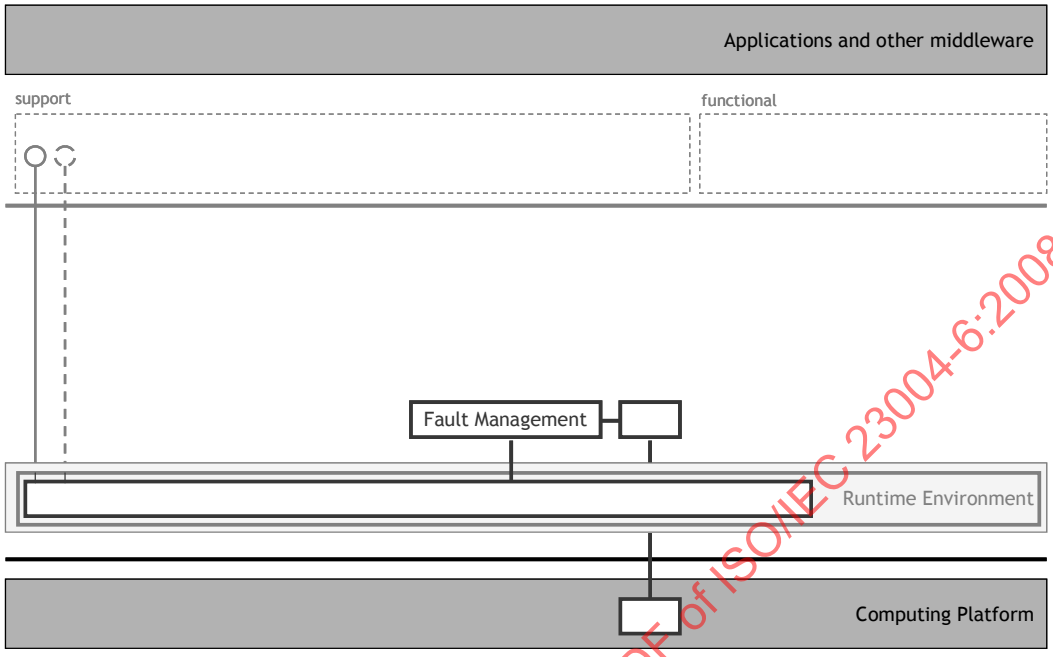

Figure 1 — Fault Management is fully transparent for applications and other middleware

M3W exposes the fault management interfaces as (an optional) part of the M3W API. These interfaces enable the development of a Fault Manager (responsible for coordination of fault management) or a Middleman (wrapper with fault tolerance mechanisms) by an application or other middleware developer. Finally, it also enables control of the Fault Management framework (which realization entities need to be managed and how). Integration with the overall architecture shall be done according to ISO/IEC 23004-1.

The Fault Management API consists of the following parts:

— Fault Management Instantiation Policy

— Fault Manager / Middleman (Coordination)

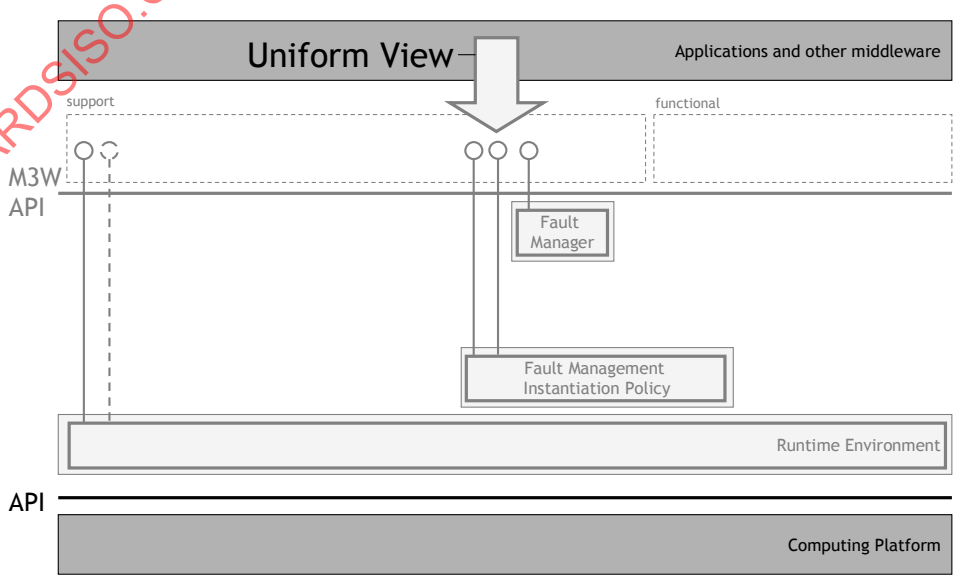

Figure 2 — Fault Management API

6 Fault Management interface suites

6.1 Interfaces for controlling Fault Management

6.1.1 Fault Management

6.1.1.1 Concepts

The Fault Management Framework enables wrapping "un-trusted" services with a middleman. The middleman contains fault tolerance mechanism such as error detection and error recovery. The Fault Management Framework can be controlled in the sense that we can specify which services need to be wrapped by which middlemans.

In a system that supports Fault Management the Runtime Environment must be extended with a Fault Management Instantiation Policy. This logical extension to the Runtime Environment provides an interface to register a Middleman Service for a registered service that needs to be managed.

6.1.1.2 Types and Constants

6.1.1.2.1 Public Types and Constants

6.1.1.2.1.1 Error Codes

Signature

```
const rcResult RC_ERR_FAULTMANAGEMENT_NOT_ALLOWED = 0x00000001;
```

Qualifiers

— Error-codes

Description

The non-standard error codes that can be returned by functions of this logical component

Constants

Table 1

Name	Description
`RC_ERR_FAULTMANAGEMENT_NOT_ALLOWED`	The operation that is invoked is not allowed. This error is returned by the `setMiddleMan` operation, for example when it is not allowed to wrap the service due to security reasons.

6.1.1.2.1.2 rcFaultManagement_MiddlemanRecord_t

Signature

```
struct _rcFaultManagement_MiddlemanRecord_t {
           pUUID middlemanId;
           pUUID svcId;
           pUUID dummyId;
} rcFaultManagement_MiddlemanRecord_t, *prcFaultManagement_MiddlemanRecord_t;
```

Qualifiers

— struct-element

Description

A data structure that is used to describe a middleman in the Fault Management framework. This structure contains the id of the middleman, the id of the service that is managed by the middleman and an optional dummy Id that can be used by the middleman. This dummy id is needed in order for the middleman to be able to instantiate the service that is managed (avoiding an infinite loop of instantiating middlemans).

6.1.1.2.2 Model Types and Constants

None

6.1.1.3 Logical Component

6.1.1.3.1 Interface Role Model

Figure 3 — Interface Role Model depicts the interface-role model of the FaultManagement logical component (grey box). It shows the interfaces and the roles involved with this component.

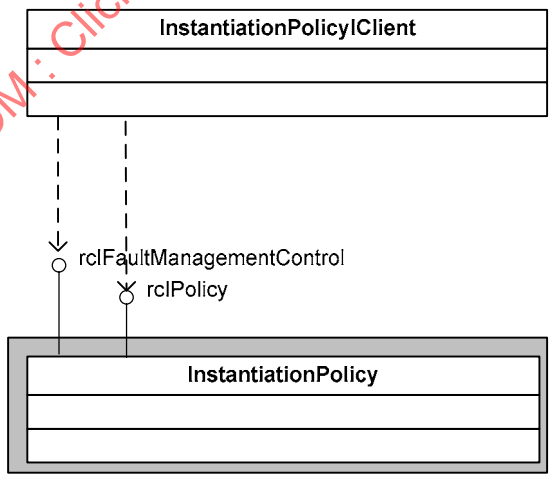

Figure 3 — Interface Role Model

A Fault Management logical component contains the `InstantiationPolicy` role that provides the `rcIFaultManagementControl` interface. This interface can be used by a client to control which services are managed by which middleman. The `InstantiationPolicy` role also provides the `rcIPolicy` interface. This interface is used by a client (usually the Runtime Environment) to ask the `InstantiationPolicy` which service needs to be instantiated (middleman or the original service)

6.1.1.3.2 Diversity

6.1.1.3.2.1 Provided Interfaces

Table 2

Role	Interface	Presence
InstantiationPolicy	rcIFaultManagementControl	Mandatory
InstantiationPolicy	rcIPolicy	Mandatory

6.1.1.3.2.2 Configurable Items

The behavior of the InstantiationPolicy depends on the contents of the Fault Management registry. This registry contains information about which services are managed by which middleman and information that enables the middleman to instantiate the managed service while all other clients will get the middleman on request of the service that is managed. This registry is specified using the following attribute.

Table 3

Role	Attribute
InstantiationPolicy	middlemanRecords

6.1.1.3.2.3 Constraints

None.

6.1.1.3.3 Instantiation

6.1.1.3.3.1 Objects Created

The logical component FaultManagement and the InstantiationPolicy role are always available. They are not created by the clients of M3W.

Table 4

Type	Object	Multiplicity
InstantiationPolicy	instantiationPolicy	1

6.1.1.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.1.1.3.4 Execution Constraints

The `FaultManagement` logical component is thread-safe.

6.1.1.4 Roles

6.1.1.4.1 InstantiationPolicy

Signature

```
role InstantiationPolicy {
    rcFaultManagement_MiddlemanRecord_t     middlemanRecords[];
}
```

Qualifiers

— Root

Description

This role is the single role of the `FaultManagement` logical component. This role enables controlling Fault Management by controlling which services are wrapped by which middleman. The middleman determines which fault tolerance mechanisms are used.

Independent Attributes

Table 5

Attribute	Description
`middlemanRecords`	This attribute is used to store which services are managed by which middleman. It also contains information that enables the middleman to instantiate the managed service without getting into an infinite loop of instantiation of middlemans.

Invariants

— None

Instantiation

The `InstantiationPolicy` role is always created as part of the `FaultManagement` logical component.

Active Behavior

The `InstantiationPolicy` role has no active behavior.

6.1.1.5 Interfaces

6.1.1.5.1 rclFaultManagementControl

Qualifiers

— None

Description

This interface enables control of Fault Management by identifying which services need to be wrapped by which middleman. The interface provides an operation for registration of a new middleman for a service (from that moment on all new instances of the services that are instantiated will be wrapped by an instance of the middleman) and an operation for un-registration of a middleman for a service.

Interface ID

```
{ 6bf7c2b7-ed1b-4f5d-941d-8c9d7c5bb6a5 }
```

6.1.1.5.1.1 setMiddleman

Signature

```
rcResult setMiddleman (
    [in] pUUID middleman,
    [in] pUUID untrusted,
    [in] pUUID dummy
);
```

Qualifiers

— synchronous

— thread-safe

Description

Add a middleman record to the Fault Management registry.

Parameters

Table 6

Name	Description
`middleman`	Reference to the UUID of the middleman that will be wrapping (adding fault tolerance mechanisms) the un-trusted service.
`untrusted`	Reference to the UUID of the un-trusted service that will be wrapped by the middleman
`dummy`	Reference to a dummy UUID that will be used by the middleman in order to instantiate the un-trusted service.

Return Values

Only the non-standard error codes are listed.

Table 7

Name	Description
RC_ERR_FAULTMANAGEMENT_NOT_ALLOWED	This operation is not allowed. Reason is not specified.

Precondition

middlemanRecords = M

Action

None

Postcondition

middlemanRecords = M ∪ <middleman,untrusted,dummy>

6.1.1.5.1.2 clearMiddleman

Signature

```
rcResult clearMiddleman (
    [in] pUUID middleman,
    [in] pUUID untrusted
);
```

Qualifiers

— synchronous

— thread-safe

Description

Remove a middleman record from the Fault Management registry.

Parameters

Table 8

Name	Description
middleman	Reference to the UUID of the middleman that is wrapping (adding fault tolerance mechanisms) the un-trusted service.
untrusted	Reference to the UUID of the un-trusted service that is wrapped by the middleman

Return Values

Standard.

Precondition

`middlemanRecords = M`

Action

None.

Postcondition

`middlemanRecords ∩ <middleman,untrusted,X> = ∅ for all X`

6.1.1.5.2 rclPolicy

Qualifiers

— None

Description

This interface is used by the client (usually Runtime Environment) to ask the `InstantiationPolicy` which service should be instantiated based on a requested service or a service proposed by another policy.

Interface ID

`{ 8751b464-f875-4339-8195-85ded910c82d }`

6.1.1.5.2.1 getServiceUUID

Signature

```
rcResult getServiceUUID (
    [in]  pUUID svcId,
    [out] pUUID *retValue
);
```

Qualifiers

— Synchronous

Description

Return the UUID of the service that should be created according to this `InstantiationPolicy` based on the UUID of the proposed UUID.

Parameters

Table 9

Name	Description
`svcId`	Reference to the UUID of the service that is proposed for instantiation.
`retValue`	Reference to the UUID of the service that is the proposal by this policy

Return Values

Default.

Precondition

```
middlemanRecords = M
```

Action

None

Postcondition

```
(∃ m,d : <m,svcId,d> ∈ M)  ⇒ retValue = m

(∃ m,s : <m,s,svcId> ∈ M)  ⇒ retValue = s

otherwise retValue = svcId
```

6.2 Interfaces for Fault Management coordination

6.2.1 Fault Manager

6.2.1.1 Concepts

The scope of a Middleman is restricted to the Service it wraps. In case where fault management techniques must be coordinated over multiple Middlemans, a coordinating entity, called Fault Manager may be used. This interface suite specifies the interaction pattern between the Middlemans and the Fault Manager but leaves the handled information and the internal logic of the Fault Manager and Middlemans out of scope, as this is system specific.

6.2.1.2 Types and Constants

6.2.1.2.1 Public Types and Constants

6.2.1.2.1.1 Error Codes

Signature

None

Qualifiers

— Error-codes

Description

The non-standard error codes that can be returned by functions of this logical component

Constants

None

6.2.1.2.1.2 rcFaultManager_MiddlemanInstanceRecord_t

Signature

```
struct _rcFaultManager_MiddlemanInstanceRecord_t {
           prcIMiddleman middlemanIntf;
           pUUID middlemanId;
           Int32 Id;
} rcFaultManager_MiddlemanInstanceRecord_t,
  *prcFaultManager_MiddlemanInstanceRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to store registration of a middleman instances in the FaultManager. This structure contains the interface reference of the middleman instance, the UUID of the middleman, and the id of the middleware instance.

6.2.1.2.2 Model Types and Constants

None

6.2.1.3 Logical Component

6.2.1.3.1 Interface Role Model

Figure 3 — Interface Role Model depicts the interface-role model of the Fault Manager logical component (grey box). It shows the interfaces and the roles involved with this component.

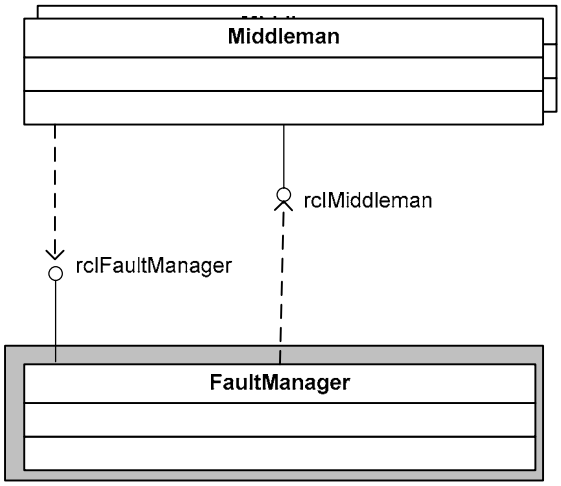

Figure 4 — Interface Role Model

A `FaultManager` Logical Component contains the `FaultManger` role that provides the `rcIFaultManager` interface. This interface can be used by a client to register a `Middleman`. The same interface can be used by the middleman to escalate problems to the `FaultManager`. The other role in this interface suite is the `Middleman` which provides the `rcIMiddleman` interface. This interface can be used by the `FaultManager` to indicate desired fault management operations to a `Middleman`.

6.2.1.3.2 Diversity

6.2.1.3.2.1 Provided Interfaces

Table 10

Role	Interface	Presence
`FaultManager`	`rcIFaultManager`	`Mandatory`
`Middleman`	`rcIMiddleman`	`Mandatory`

6.2.1.3.2.2 Configurable Items

The behavior of the FaultManager is dependent on the Middleman (instances) that are registered at the `FaultManager`. These references to these instances are stored internally and are modeled by the `middlemanInstanceRecords`.

Table 11

Role	Attribute
`FaultManager`	`middlemanInstanceRecords`

6.2.1.3.2.3 Constraints

None.

6.2.1.3.3 Instantiation

6.2.1.3.3.1 Objects Created

The logical component `FaultManager` and the `FaultManger` role are singletons. At most one instance of these entities is created. The `FaultManager` role is created on creation of the `FaultManager` logical component.

Table 12

Type	Object	Multiplicity
`FaultManager`	`faultManger`	`1`

6.2.1.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.2.1.3.4 Execution Constraints

The `FaultManager` Logical Component is thread-safe.

6.2.1.4 Roles

6.2.1.4.1 FaultManager

Signature

```
role FaultManager {
    rcFaultManager_MiddlemanInstanceRecord_t       middlemanInstanceRecords[];
}
```

Qualifiers

— Root

Description

The `FaultManager` is a singleton (at most 1 instance will be created). This role is responsible for coordination of fault management activities. The scope of a `Middleman` is local. The scope of the FaultManager is system wide.

Independent Attributes

Table 13

Attribute	Description
`middlemanInstanceRecords`	This attribute is used to store the middleman instances that are registered at the `FaultManager`.

Invariants

— None

Instantiation

The `FaultManager` role is always created as part of the `FaultManager` logical component.

Active Behavior

None.

6.2.1.5 Interfaces

6.2.1.5.1 rclFaultManager

Qualifiers

— None

Description

The `rcIFaultManager` interface is used by `Middleman`s. It enables error notification to the `FaultManager`. The `FaultManager` can coordinate error handling at the system level. The `rcIFaultManager` interface also contains operations for registration and un-registration of middleman.

Interface ID

```
{ aace1505-e854-4a62-bb5a-d6673f6650a4 }
```

6.2.1.5.1.1 registerMiddleman

Signature

```
rcResult registerMiddleman (
    [in]  prcIMiddleman middlemanIntf,
    [in]  pUUID middlemanId,
    [out] Int32 *retValue
);
```

Qualifiers

— Synchronous

— Thread-safe

Description

Add a new middleman instance to the `middlemanInstanceRecords`.

Parameters

Table 14

Name	Description
`middlemanIntf`	Reference to the `rcIMiddleman` interface of the middleman that is registered
`middlemanId`	UUID of the middleman that is registered
`retValue`	Return value that contains the id of the middleman instance given by the `FaultManager`. This id can be used for example for un-registration.

Return Values

Default.

Precondition

```
middlemanInstanceRecords = M
```

Action

None

Postcondition

```
middlemanInstanceRecords = M ∪ <middlemanIntf, middlemanId, id>

(∀ mi, md : <mi,md, id> ∉ M)

retValue = id
```

6.2.1.5.1.2 unregisterMiddleman

Signature

```
rcResult unregisterMiddleman (
    [in] Int32 middlemanId
);
```

Qualifiers

— Synchronous

— Thread safe

Description

Remove middleman instance from `middlemanInstanceRecords`.

Parameters

Table 15

Name	Description
`middlemanId`	Id that was assigned by the `FaultManager` to the middleman instance.

Return Values

Default.

Precondition

[todo]

Action

None

Postcondition

[todo]

6.2.1.5.1.3 escalate

Signature

```
rcResult escalate (
    [in] Int32 middlemanId,
    [in] String notificationId
);
```

Qualifiers

— Synchronous

— Thread-safe

Description

Notification of errors to the FaultManger when for example the error cannot be handled locally.

Parameters

Table 16

Name	Description
`middlemanId`	Id of the middleman instance that is raising the error notification. This id is assigned upon registration of a middleman instance.
`notificationId`	The format and semantics of the `notificationId` are specific to the type of the Middleman and FaultManager; their specification is outside the scope of this interface suite.

Return Values

Standard.

Precondition

true

Action

None.

Postcondition

Platform specific.

6.2.1.5.2 rclMiddleman

Qualifiers

— None

Description

Interface used by the Fault Manager to control the Middleman's fault management logic.

Interface ID

```
{ cdd56af6-7d0c-4031-b1e3-57cc43342037 }
```

6.2.1.5.2.1 designate

Signature

```
rcResult designate (
    [in]  String notificationId,
);
```

Qualifiers

— Synchronous

Description

Issues fault management notifications to the Middleman. The notification consists of a `notificationId` which identifies the desired fault management operation.

Parameters

Table 17

Name	Description
`notificationId`	Platform specific identification of the desired fault management operation. The format and semantics are specific to the Middleman and Fault Manager. Specification of format and semantics is outside the scope of this interface suite.

Return Values

Default.

Precondition

True

Action

None

Postcondition

Platform specific

7 Realization overview

7.1 Introduction

This is an informative clause that gives an overview of the Fault Management framework realization. The realization is based on the core realization technology specified in ISO/IEC 23004-3.

7.2 Concepts and terminology

7.2.1 Failure, Error, Fault

We adopt the basic terminology of dependable systems introduced in [13]. For the intuitive definition of wrong-doing being something incorrect, illegal, mistaken, improper, unsuitable or out of order, we have the following basic definitions that relate to fault management.

A *failure* is an event that occurs when the delivered service of a system deviates from correct service [2)]. Examples of failures are unexpected behavior of a system with respect to its specification such as, for example, delivery of wrong data to the user or loss of communication over a network that needed to drop packets due to a buffer overflow.

An *error* is an unwanted software state that is liable to lead to a failure. Examples of errors are invalid memory states, e.g. detectable by checksums, or the drop of packets from a network due to a buffer overflow.

A *fault* is the adjudged or hypothesized cause of an error. Often faults are mistakes made by a human, but they can also be caused by external factors such as defective hardware. Examples of faults include design/programming errors by a system developer, the toggling of memory bits due to electromagnetic disturbances in the environment where a given hardware operates, and a network congestion beyond its buffer capacities.

Fault tolerance is the technique of designing a system that is failure-free despite the existence of faults. Fault tolerance mechanisms are based on detecting, resolving and hiding errors. Note that the faults in a fault-tolerant system remain but the resulting errors are resolved to prevent failures.

7.2.2 Unit of Failure

The term *Unit of Failure (UoF)* is used to denote an entity of the system on which a failure can be observed, i.e. it is not possible to decompose this system entity to constituent entities and justifiably associate the observed failure with one of them. Then a UoF experiences a failure atomically, as a single whole.

One failure is bound to a single UoF. This property does not prohibit a failure on a given UoF to cause a fault, an error and another failure on a different UoF. Rather, it prohibits two distinct UoFs to experience the same failure (failure as an instance of some event). Hence, if two distinct UoFs experience simultaneously the same type of failure, these two failures are not causally related to each other, although they may have a common fault in the *fault* → *error* → *failure* → *fault* → … chains that led to each of them.

In the case of internal faults, it follows from the above definition that the fault and the error that are immediate predecessors of a failure in a *fault* → *error* → *failure* chain respectively reside in and are detected on the same UoF that exposes the given failure. As a consequence, a fault (respectively error) resides in a single UoF. The same type of fault (respectively error) may reside in different UoFs but that corresponds to a number of different instances of the fault type (respectively error type).

In the context of M3W, a Method off an Interface is the Unit of Failure.

7.2.3 Unit of Fault Management

The term *Unit of Fault Management (UoFM)* is used to denote an entity of the system that will be subject to Fault Management as a whole. For the M3W Fault Management Framework, the UoFM is a Service Instance. This means that fault management operates independently for each instance of each un-trusted Service.

Note, however, that the Fault Manager role provides options for coordinating the operations between various Fault Managed Service Instances.

---

2) For the purpose of the Fault Management Framework, correct service is service that meets the specification (i.e. the framework does not consider faults in the specification).

7.3 Assumptions and capabilities

7.3.1 Assumptions

The characteristics of this framework are that the unit of Fault Management is a Service Instance and that such a Service Instance has no reflection or inspection capability.

The following assumptions are made:

— The Components exist and cannot be inspected or changed, no inside knowledge of the Services is used or assumed, but they do comply with the M3W specification, in particular:

— the creator of the fault managed Service Instance cannot be modified,

— the clients of the fault managed Service Instance cannot be modified,

— the fault managed Service cannot be modified,

— the servers, i.e. Services bound to requires ports, of the fault managed Service Instance cannot be modified.

— The IDL definitions of the fault managed Service and of all its interfaces are known.

— The fault-managed Components and Services do not need to provide any reflection or inspection capability to any other entities. The fault managed Service Instances are therefore black-boxes (i.e. there exists no possibility of observation or intervention on the internals of the Service Instance).

— The fault-managed Services do not have to provide any specific Interface(s) related to fault management, nor do their Components.

7.3.2 Capabilities

Given a set of Services that cannot be inspected or changed and are possibly linked by uses/requires relationships, the Fault Management Framework provides the capability to add fault tolerance without either side of the relationship being modified. The Fault Management Framework enables:

— Intercepting and stopping method invocations from other Service Instances on a fault managed Service Instance.

— Intercepting and stopping a fault managed Service Instance method invocations through its requires ports.

— Replacing a fault managed Service Instance X by another Service Instance X' by redirecting to X' all bindings to X.

— Isolating a fault managed Service Instance from the running system.

— Coordinating fault management operations between multiple fault managed Service Instances.

— Adding new detection mechanisms,

— Adding new recovery mechanisms.

7.4 Entities and responsibilities

The Fault Management framework enables adding fault tolerance mechanisms to Service Instances or a composition of Service instances (Services according to the specification in ISO/IEC 23004-3). There are three important entities in the Fault Management Framework:

— Middleman: Interception and redirection, which is called wrapping, is done by a "Middleman". This Middleman intercepts calls to its interface methods which it may redirect to the wrapped Service Instance or to an alternate implementation (which may contain the fault tolerance mechanisms).

— Fault Manager: The scope of a Middleman is restricted to the Service it wraps. Where fault management techniques must be coordinated over multiple Middlemans, a coordinating entity called the Fault Manager may be used. The M3W Fault Management Framework specifies the interaction pattern between the Middlemans and the Fault Manager but leaves the internal logic of the Fault Manager and Middlemans out of the scope, as this is system-specific.

— Fault Management Instantiation Policy: In a system that supports Fault Management the Runtime Environment must be extended with the Fault Management Instantiation Policy. This logical extension enables instantiation of a Middleman upon the instantiation request for a Service instance.

8 Fault Management realization

8.1 Initial Situation

In absence of Fault Management, the following roles may be present in a system, relative to a given (untrusted) Service Instance:

— The Untrusted Service Instance (USI), which may have implemented interfaces, provides ports, attributes, and requires ports.

— The creator of the USI that initiates the instantiation.

— The Runtime Environment, including the standard instantiation policy, that determines the appropriate Component, loads it, and requests the creation of the Service Instance.

— The configurator of the USI (i.e. the entity that binds interface references to the requires ports which is known as 3rd party binding).

— The clients, i.e. Service Instances or applications that have references to interfaces (implemented interfaces and/or provided ports) implemented by USI.

— The servers, i.e. Service Instances from which interface references have been bound to USI's requires ports.

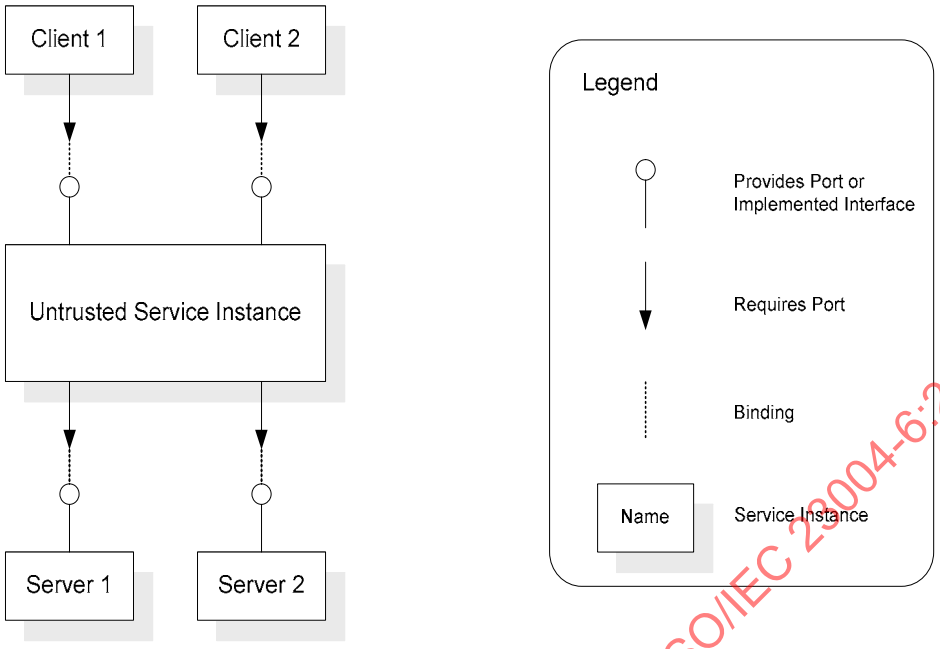

Figure 5 — Initial situation of bindings

NOTE Only explicit dependencies that are expressed through requires ports are taken into consideration here. The USI can also have implicit dependencies, e.g. through the direct creation of Service Instances using the Runtime Environment.

NOTE The above situation does not provide any fault management and may lead to failures.

8.2 The basic principle

A client of the USI calls one of the interface methods of the USI. The thread of control goes inside the USI code and may in turn go to (some of) the USI's servers. The basic idea for adding fault tolerance is that the thread of control between the client and the USI and between the USI and its servers can be

— intercepted and

— redirected

without any of the entities implementing the roles listed in the previous paragraph being modified in any manner.

This basic interception and redirection provides the following capabilities:

— A number of fault tolerant technique implementations may be inserted in the thread of control without modifying the original Component.

— The USI can be dynamically replaced by another instance of the same Service or of a compliant Service.

— Under certain conditions [3)], the Component that hosts the USI may be un-initialized, de-activated and un-installed from the system without risk to the clients or to the system integrity.

3) If no other service instances have been created from this Component.

8.3 Roles

8.3.1 Middleman

Interception and redirection, which is called "wrapping", is done using a new entity, called the Middleman. This Middleman poses as the USI, intercepting calls to its interface methods. It may redirect calls to the USI or to an alternate implementation. Furthermore it may insert implementations of fault management techniques before and after redirected calls. Finally it may notify a Fault Manager (introduced below) of certain conditions.

A Middleman instance MI is dedicated to the interception of calls directed to the USI. The Middleman definition shall comply with the Untrusted Service (US) definition, in order to guarantee a similar appearance to the binder and clients of the USI. However, it can also implement additional interfaces and provide additional ports.

A Middleman is a Service as any other, i.e. it is part of a Component. As such, in order to be used, a Middleman must be registered and instantiated, just as with any other Service. For each USI, an instance of the Middleman is created to wrap that USI.

If diversity in fault managing a given untrusted Service is required, this can be realized in the following two ways:

— Hard coded in the Middleman, potentially controlled by a dedicated interface or attribute

— Multiple Middleman implementations in different Components that are selectively registered with the Runtime Environment (but only one at any given point in time).

Figure 6 shows the same example as described in Figure 5, with the addition of a Middleman, i.e. the grey box, that intercepts all incoming calls on the provides ports of the untrusted Service. Note that this can also be done for implemented interfaces.

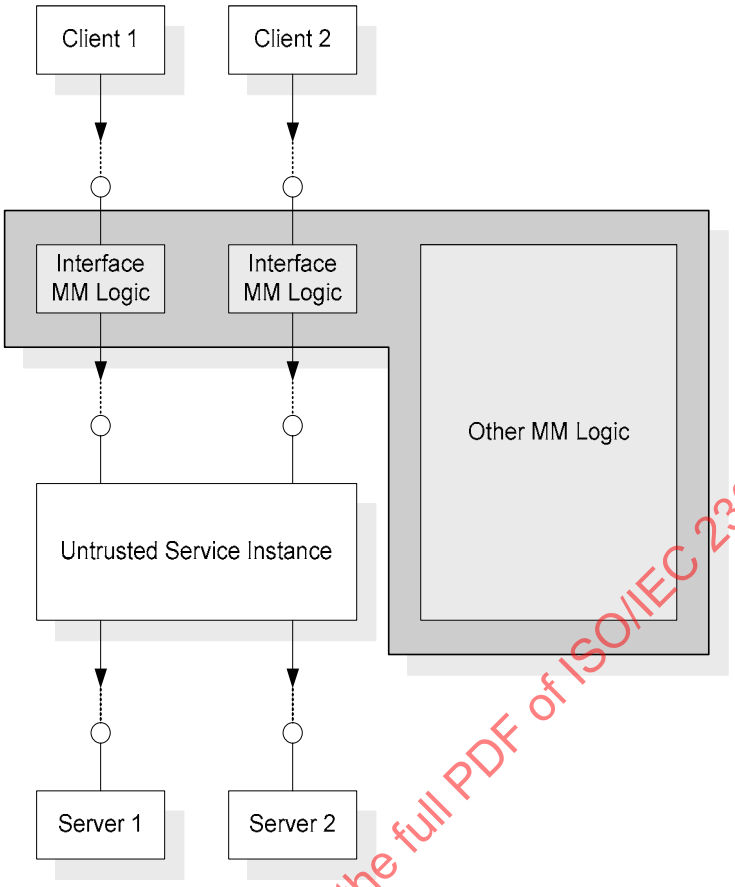

Figure 6 — Fault Managed Situation

Figure 7 shows the same example as described in Figure 6, with the addition that the Middleman also intercepts all outgoing calls through the requires ports of the USI.

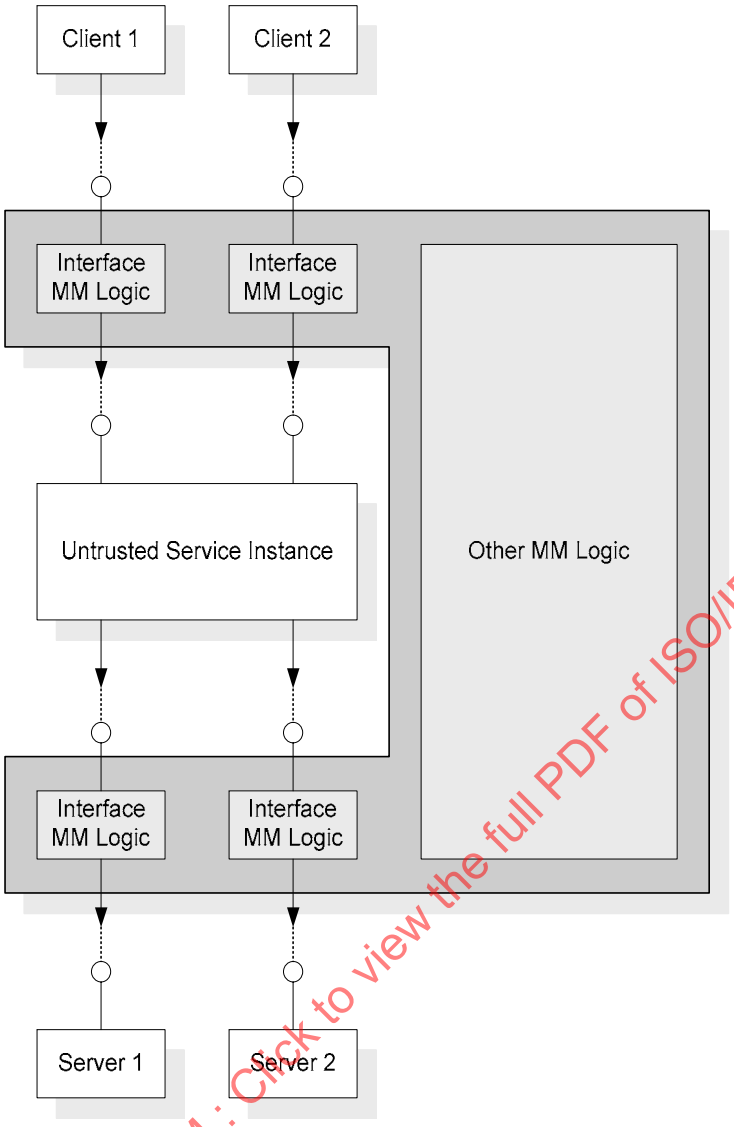

Figure 7 — Fault Managed Situation

A Middleman complies with the Service it wraps. Therefore it appears to its clients as identical to the wrapped Service (i.e. exposes the same interfaces), and cannot be distinguished from it. It also means that the Middleman's Service Factory must comply with the Untrusted Service's Service Factory.

As can be seen from the above figures, the Middleman has inserted some logic between its interfaces and the corresponding interfaces of the USI, in the figure denoted as "Interface MM logic". This interception logic can include any fault management technique. Furthermore there may be fault management logic not directly in the interface call path, as denoted by "Other MM Logic".

In dealing with each of the USI's requires ports, the Middleman has two principle options:

— Forward the `bindTo<portname>()` and `unbind<portname>()` operations from its Service Specific interface to the USI's Service Specific interface. This option is depicted in Figure 6. Effectively this means that the binding is done on the USI's requires ports. As a consequence there is no interception of method calls to the USI's servers.

— Bind an interface it provides to the USI's requires port, and effectuate the `bindTo<portname>()` and `unbind<portname>()` operations on its own corresponding requires port. This option is depicted in Figure 7 for both requires ports. Effectively this means that the outgoing calls made by the USI on its

requires port can be intercepted by the Middleman instance. It may, as part of its operation, forward those calls to the USI's server.

A Middleman must be registered with the Fault Management Instantiation Policy.

8.3.1.1 Interfaces

8.3.1.1.1 Introduction

The Middleman is a Service that takes the place of, and complies with, a Service that is untrusted. From the perspectives of the creator, the configurator, and the clients of the untrusted Service Instance the Middleman is indistinguishable from the untrusted Service Instance. From the perspective of the Fault Management Framework, the Middleman may have the following additional fault management interface:

— `rcIMiddleman`, which is used by the Fault Manager to control the Middleman's fault management logic. In the case where the Middleman is entirely autonomous, this interface does not have to be provided.

8.3.1.1.2 rcIMiddleman

```
interface rcIMiddleman { cdd56af6-7d0c-4031-b1e3-57cc43342037 } {
    void designate( in String notificationId );
};
```

The `designate()` operation issues fault management notifications to the Middleman. The notification consists of a `notificationId` which identifies the desired fault management operation. The format and semantics of the `notificationId` are specific to the type of the Middleman and its Fault Manager; their specification is outside the scope of the Fault Management Framework.

8.3.2 Fault Manager

The scope of a Middleman is restricted to the Service it wraps. In case where fault management techniques must be coordinated over multiple Middlemans, a coordinating entity, called Fault Manager may be used. The M3W Fault Management Framework specifies the interaction pattern between the Middlemans and the Fault Manager but leaves the handled information and the internal logic of the Fault Manager and Middlemans out of the scope, as this is system specific.

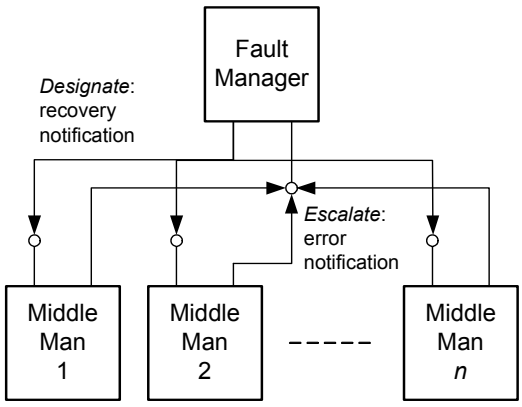

Figure 8 — Fault Manager coordinating multiple Middlemans

A Fault Manager is a singleton Service. The runtime environment returns the same instance every time a Middleman asks for a Fault Manager instance. Therefore all Middlemans that use the same type of Fault Manager get the same Fault Manager Instance [4)]. During execution a Middleman may notify its Fault Manager about some state that requires fault management actions that exceed its scope of its control. Based on the its fault management policy, the Fault Manager takes appropriate action, e.g. communicate back to the involved Middlemans the fault management actions they should take.

8.3.2.1 Interfaces

8.3.2.1.1 Introduction

The Fault Manager is a singleton Service. It shall implement the following interface:

— `rcIFaultManager`, which is used by Middlemans to register and unregister with the Fault Manager, and to notify the Fault Manager about some state that requires fault management actions that are beyond the scope of the Middleman.

8.3.2.1.2 rcIFaultManager

```
interface rcIFaultManager { aace1505-e854-4a62-bb5a-d6673f6650a4 } {
    Int32 registerMiddleman( in rcIMiddleman middlemanIntf
                             in pUUID middlemanId );
    void unregisterMiddleman( in Int32 middlemanId );
    void escalate( in Int32 middlemanId, in String notificationId );
};
```

The `registerMiddleman()` operation registers a Middleman Instance with its Fault Manager. The Middleman must supply its `rcIMiddleman` interface reference as the `middlemanIntf` parameter, and must supply the UUID of the Middleman Service type as the `middlemanId` parameter. The Fault Manager may use the `middlemanUuid` value to determine what type of Middleman is being registered. The Fault Manager shall assign a unique identification to this Middleman instance and return it for subsequent identification of the Middleman.

The `unregisterMiddleman()` operation unregisters a previously registered Middleman, using its `middlemanId` as key.

The `escalate()` operation issues fault management notifications to the Fault Manager. The notification consists of a `middlemanId`, to identify the calling Middleman instance and a `notificationId`, to identify the desired fault management operation. The format and semantics of the `notificationId` are specific to the type of the Middleman and its Fault Manager; their specification is outside the scope of the Fault Management Framework.

Fault Management Instantiation Policy

The Fault Management Framework extends the standard Runtime Environment instantiation policy in order to create a Middleman instead of the requested fault managed Service Instance. The following interfaces are specified:

— `rcIFaultManagementControl`, to register and unregister Middlemans for a particular Service type.

4) Typically a Middleman will use the Runtime Environment's getServiceInstance() operation to obtain a reference to the instance of its Fault Manager. Because the Fault Manager is a singleton, the first Middleman will create the instance and subsequent Middlemans get a reference to that instance.

8.3.2.1.3 rclFaultManagementControl

```
interface rcIFaultManagementControl
{ 6bf7c2b7-ed1b-4f5d-941d-8c9d7c5bb6a5 }  {
    void  setMiddleman( in pUUID middleman, in pUUID untrusted, in pUUID dummy )
        raises RcXNotAllowed;
    void clearMiddleman( in pUUID middleman, in pUUID untrusted )
};
```

This interface is provided as a system library, just as the other direct RRE interfaces.

The `setMiddleman()` operation registers the `Middleman` uuid for the `untrusted` uuid, so that subsequent instantiations of the untrusted Service Factory will create the Middleman Service Factory instead. It also registers the dummy uuid that is used by the middleman in order to create an instance of the untrusted service. This dummy uuid is translated into the uuid of the untrusted service by the Fault Management Instantiation Policy.

The `clearMiddleman()` operation clears the registration between the `middleman` uuid and the `untrusted` uuid.

8.3.3 Interaction between roles

Figure 9 shows two sequences of interactions between one Middleman instance, one un-trusted Service Instance (USI) and one client instance calling a method of this un-trusted Service. They are given as examples and show two possible behaviors of the client thread of control.

The first sequence (up to step 6) involves an un-trusted Service that does return from the call of its method. In this case, the un-trusted Service may behave incorrectly but it does terminate, possibly with an error code or any erroneous result. In this case, the Middleman indicates the situation to the Fault Manager, which takes a decision in accordance to the system policy, and returns its decision to the Middleman. The recovery action is then applied by the Middleman to the un-trusted Service (for example a method retry).

The second sequence (form step 9) involves an un-trusted Service that does not return from the call of its method. Such a situation is detected at the Middleman end by means of (for example) a timer that elapses before the call is returned. This situation is reported to the Fault Manager, which takes a decision in accordance to the system policy, and returns the decision to the Middleman. An example of a recovery action is the deletion of the un-trusted Service Instance.

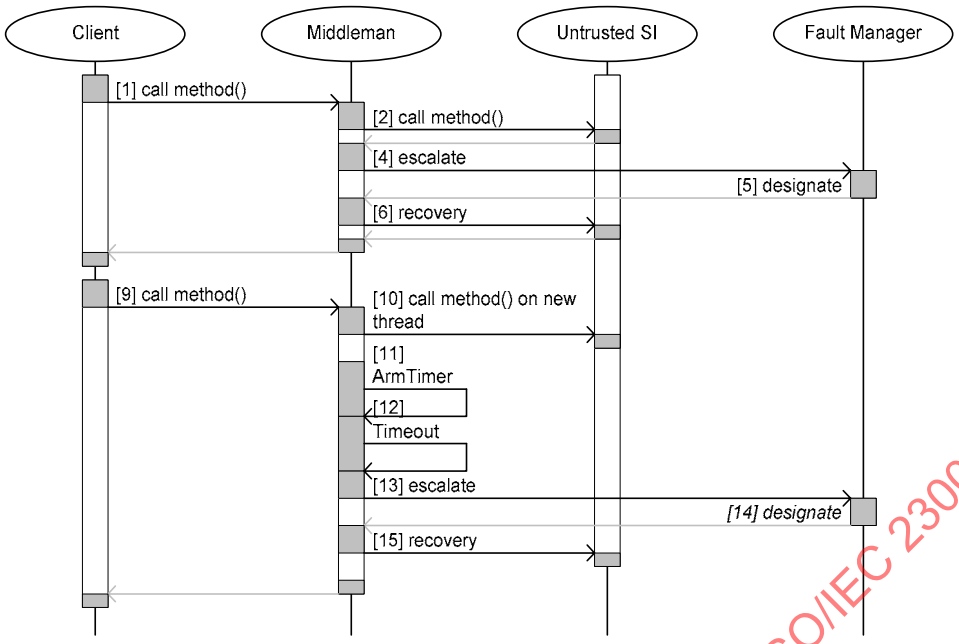

Figure 9 — Sequence chart involving one Middleman and the Fault Manager

8.3.4 Fault Management Instantiation Policy

In a system that supports Fault Management the Runtime Environment must be extended with the Fault Management Instantiation Policy. This logical extension to the Runtime Environment provides an interface to register a Middleman Service for a registered Service that needs to be fault managed. The Fault Management Instantiation policy records these relations in a registry that is a logical extension of the Runtime Environment registry.

Policies can be considered as a function on a uuid. Based on an input uuid (proposed or requested) a policy will give the uuid of the service that needs to be created according to that policy. This output can be used as input for another policy. The Fault Management Instantiation Policy will cause the registered Middleman's uuid to be returned for every uuid of a Service for which a Middleman has been registered.

8.3.4.1 Interfaces

8.3.4.1.1 Introduction

The Fault Management Instantiation Policy is a logical extension to the Runtime Environment. It shall implement the following interface.

— `rcIPolicy`, which is used by the Runtime Environment to ask policies what service should be instantiated based on the uuid of the (currently) proposed uuid. The first proposed uuid is the uuid of the requested service. The Runtime Environment can use a number of policies to determine the uuid of the service that in fact will be instantiated.

8.3.4.1.2 rcIPolicy

```
interface rcIPolicy { 8751b464-f875-4339-8195-85ded910c82d }  {
    pUUID getServiceUUID( in pUUID svcId )
};
```

The `getServiceUUID` operation returns the uuid of the service that should be instantiated according to the Fault Management Instantiation Policy based on the uuid (`svcId`) of the proposed / requested Service. When a middleman is registered for svcId the policy returns the uuid of the middleman. When `svcId` is equal to a dummy id of an untrusted service the policy returns the uuid of the untrusted service. Otherwise the policy returns `svcId`.

8.4 Interaction with the System Integrity Management Framework

The entities in the Fault Management Framework may notify the System Integrity Management Framework of any relevant fault management information. The System Integrity Management Framework is specified in ISO/IEC 23004-7.