The Consultative Committee for Space Data Systems

Draft Recommendation for
Space Data System Practices

MISSION OPERATIONS MESSAGE ABSTRACTION LAYER—C++ API

DRAFT RECOMMENDED PRACTICE

CCSDS 523.2-R-1

RED BOOK

March 2017

The Consultative Committee for Space Data Systems

Draft Recommendation for
Space Data System Practices

MISSION OPERATIONS MESSAGE ABSTRACTION LAYER—C++ API

DRAFT RECOMMENDED PRACTICE

CCSDS 523.2-R-1

RED BOOK

March 2017

AUTHORITY

Issue:	Red Book, Issue 1
Date:	March 2017
Location:	Not Applicable

(WHEN THIS RECOMMENDED PRACTICE IS FINALIZED, IT WILL CONTAIN THE FOLLOWING STATEMENT OF AUTHORITY:)

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and represents the consensus technical agreement of the participating CCSDS Member Agencies. The procedure for review and authorization of CCSDS documents is detailed in *Organization and Processes for the Consultative Committee for Space Data Systems* (CCSDS A02.1-Y-4), and the record of Agency participation in the authorization of this document can be obtained from the CCSDS Secretariat at the e-mail address below.

This document is published and maintained by:

CCSDS Secretariat
National Aeronautics and Space Administration
Washington, DC, USA
E-mail: secretariat@mailman.ccsds.org

STATEMENT OF INTENT

(WHEN THIS RECOMMENDED PRACTICE IS FINALIZED, IT WILL CONTAIN THE FOLLOWING STATEMENT OF INTENT:)

The Consultative Committee for Space Data Systems (CCSDS) is an organization officially established by the management of its members. The Committee meets periodically to address data systems problems that are common to all participants, and to formulate sound technical solutions to these problems. Inasmuch as participation in the CCSDS is completely voluntary, the results of Committee actions are termed **Recommendations** and are not in themselves considered binding on any Agency.

CCSDS Recommendations take two forms: **Recommended Standards** that are prescriptive and are the formal vehicles by which CCSDS Agencies create the standards that specify how elements of their space mission support infrastructure shall operate and interoperate with others; and **Recommended Practices** that are more descriptive in nature and are intended to provide general guidance about how to approach a particular problem associated with space mission support. This **Recommended Practice** is issued by, and represents the consensus of, the CCSDS members. Endorsement of this **Recommended Practice** is entirely voluntary and does not imply a commitment by any Agency or organization to implement its recommendations in a prescriptive sense.

No later than five years from its date of issuance, this **Recommended Practice** will be reviewed by the CCSDS to determine whether it should: (1) remain in effect without change; (2) be changed to reflect the impact of new technologies, new requirements, or new directions; or (3) be retired or canceled.

In those instances when a new version of a **Recommended Practice** is issued, existing CCSDS-related member Practices and implementations are not negated or deemed to be non-CCSDS compatible. It is the responsibility of each member to determine when such Practices or implementations are to be modified. Each member is, however, strongly encouraged to direct planning for its new Practices and implementations towards the later version of the Recommended Practice.

FOREWORD

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Recommended Practice is therefore subject to CCSDS document management and change control procedures, which are defined in the *Organization and Processes for the Consultative Committee for Space Data Systems* (CCSDS A02.1-Y-4). Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this document should be sent to the CCSDS Secretariat at the e-mail address indicated on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were:

Member Agencies

- Agenzia Spaziale Italiana (ASI)/Italy.
- Canadian Space Agency (CSA)/Canada.
- Centre National d'Etudes Spatiales (CNES)/France.
- China National Space Administration (CNSA)/People's Republic of China.
- Deutsches Zentrum für Luft- und Raumfahrt (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Federal Space Agency (FSA)/Russian Federation.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- Japan Aerospace Exploration Agency (JAXA)/Japan.
- National Aeronautics and Space Administration (NASA)/USA.
- UK Space Agency/United Kingdom.

Observer Agencies

- Austrian Space Agency (ASA)/Austria.
- Belgian Federal Science Policy Office (BFSPO)/Belgium.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- China Satellite Launch and Tracking Control General, Beijing Institute of Tracking and Telecommunications Technology (CLTC/BITTT)/China.
- Chinese Academy of Sciences (CAS)/China.
- Chinese Academy of Space Technology (CAST)/China.
- Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
- Danish National Space Center (DNSC)/Denmark.
- Departamento de Ciência e Tecnologia Aeroespacial (DCTA)/Brazil.
- Electronics and Telecommunications Research Institute (ETRI)/Korea.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Geo-Informatics and Space Technology Development Agency (GISTDA)/Thailand.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Institute of Space Research (IKI)/Russian Federation.
- Korea Aerospace Research Institute (KARI)/Korea.
- Ministry of Communications (MOC)/Israel.
- Mohammed Bin Rashid Space Centre (MBRSC)/United Arab Emirates.
- National Institute of Information and Communications Technology (NICT)/Japan.
- National Oceanic and Atmospheric Administration (NOAA)/USA.
- National Space Agency of the Republic of Kazakhstan (NSARK)/Kazakhstan.
- National Space Organization (NSPO)/Chinese Taipei.
- Naval Center for Space Technology (NCST)/USA.
- Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- Scientific and Technological Research Council of Turkey (TUBITAK)/Turkey.
- South African National Space Agency (SANSA)/Republic of South Africa.
- Space and Upper Atmosphere Research Commission (SUPARCO)/Pakistan.
- Swedish Space Corporation (SSC)/Sweden.
- Swiss Space Office (SSO)/Switzerland.
- United States Geological Survey (USGS)/USA.

PREFACE

This document is a draft CCSDS Recommended Practice. Its 'Red Book' status indicates that the CCSDS believes the document to be technically mature and has released it for formal review by appropriate technical organizations. As such, its technical contents are not stable, and several iterations of it may occur in response to comments received during the review process.

Implementers are cautioned **not** to fabricate any final equipment in accordance with this document's technical content.

DOCUMENT CONTROL

Document	Title	Date	Status
CCSDS 523.2-R-1	Mission Operations Message Abstraction Layer—C++ API, Draft Recommended Practice, Issue 1	March 2017	Current draft

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

Table Page

CONTENTS (continued)

Table Page

CONTENTS (continued)

Table Page

CONTENTS (continued)

CONTENTS (continued)

1 INTRODUCTION

1.1 PURPOSE OF THIS RECOMMENDED PRACTICE

This Recommended Practice defines a C++ API for the Mission Operations (MO) Message Abstraction Layer (MAL) specified in reference [1]. This book has been adapted for C++ from the CCSDS Recommended Practice for MO MAL JAVA API specified in reference [B3].

1.2 SCOPE

The scope of this Recommended Practice is the definition of all concepts and terms that establish a C++ API for consuming and providing MO services on top of the MAL. The MAL C++ API is intended to maximize the portability of the MO components across various underlying MAL implementations and transport protocols.

1.3 APPLICABILITY

This Recommended Practice serves as a specification of a C++ API providing all the concepts defined by the MAL, in particular the interaction patterns, the access control and transport interfaces.

The API supports version 1 of the MAL as specified in reference [1].

The API depends on C++ 11.

1.4 RATIONALE

The goal of this Recommended Practice is to document how to develop and utilize MAL-based services using C++.

Another objective is to create a guide for promoting portability between various software namespaces implementing the MAL C++ API and applications using the MAL C++ API.

1.5 DOCUMENT STRUCTURE

This Recommended Practice is organized as follows:

a) section 1 provides purpose and scope, and lists definitions, conventions, and references used throughout the Recommended Practice;

b) section 2 gives an overview of the API;

c) section 3 defines the MAL API that represents the MAL service interface;

d) section 4 defines how a MAL service specification is mapped to C++;

e) section 5 defines the transport API that represents the MAL transport interface;

f) section 6 defines the access control API that represents the MAL access control interface;

g) section 7 defines the encoding API, an optional set of interfaces that can be used by the transport layer in order to externalize the encoding behaviour.

NOTE – Some application code examples are given in annex C.

1.6 DEFINITIONS

Attribute: Either

– a field of a C++ class called ‘attribute’;

– the data type MAL::Attribute; or

– the interface Attribute defined in the MAL C++ API.

1.7 CONVENTIONS

1.7.1 NOMENCLATURE

The following conventions apply for the normative specifications in this Recommended Practice:

a) the words ‘shall’ and ‘must’ imply a binding and verifiable specification;

b) the word ‘should’ implies an optional, but desirable, specification;

c) the word ‘may’ implies an optional specification;

d) the words ‘is’, ‘are’, and ‘will’ imply statements of fact.

NOTE – These conventions do not imply constraints on diction in text that is clearly informative in nature.

1.7.2 INFORMATIVE TEXT

In the normative sections of this document (sections 3-7), informative text is set off from the normative specifications either in notes or under one of the following subsection headings:

– Overview;

– Background;

– Rationale;

– Discussion.

1.7.3 USE OF CAPITAL LETTERS

Names of interfaces and classes of the MAL C++ API are shown with the first letter of each word capitalized.

1.7.4 CODE TEMPLATE VARIABLES

Some code templates are given throughout this document. Those templates are parameterized with variables. A variable is used by placing its name between the enclosing characters '<<' and '>>'. For example, the variable 'method name' is used in the following code line:

```
public void <<method name>>()
```

If a variable contains a list of values, then the values are iterated with the following syntax:

```
<<variable name [i]>> … <<variable name [N]>>
```

The variable 'i' is the iteration index starting from 0.

The variable 'N' is the size of the list minus one.

If the list size is zero, then nothing is written in the code template.

If the list size is greater than zero, then the <<variable name [i]>> part is iterated N times and the <<variable name [N]>> part is finally written in the code template.

For example, the variable 'type' contains a list of types. A method 'm' taking those types as parameters is declared as follows:

```
public void m(<<type [i]>> p<<i>>, … <<type [N]>> p<<N>>)
```

If the list is empty, then the result is:

```
public void m()
```

If the list contains one element 'A' then the result is:

```
public void m(A p0)
```

The names of variables are not case-sensitive: 'field name' and 'FIELD NAME' designate the same variable.

However, the value of the variable is case sensitive following the rules explained in table 1-1.

Table 1-1: Variable Value Case Rules

Variable name case	Variable value case
All the letters in lower case. Example: `<<variable name>>`	No change
First letter in upper case and the other in lower case. Example: `<<Variable name>>`	No change except the first letter in upper case
All the letters in upper case. Example: `<<VARIABLE NAME>>`	Upper case
All the letters in lower case and between an initial and final character '!'. Example: `<<!variable name!>>`	Lower case

1.7.5 C++ CLASS NAMESPACES

In the code templates, the namespace of classes is not indicated if there is no ambiguity. However, the real C++ code needs either to prefix the namespace name to the class name or to declare the namespace in a 'using' clause at the beginning of the class definition.

1.8 REFERENCES

The following documents contain provisions which, through reference in this text, constitute provisions of this Recommended Practice. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Recommended Practice are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS documents.

[1] *Mission Operations Message Abstraction Layer*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 521.0-B-2. Washington, D.C.: CCSDS, March 2013.

NOTE – Informative references are listed in annex B.

2 OVERVIEW

2.1 GENERAL

This Recommended Practice provides an interface specification that translates the abstract service definitions of the MAL and MO Services into a C++-specific interface that can be used by programmers. The API must be faithful to the abstract services defined in the specifications and it must, by some means, provide access to the features of the services and all of their options. It plays a useful role in application portability, but plays no role in interoperability.

The following diagram presents the set of standards documentation in support of the Mission Operations Services Concept. The MO MAL C++ API belongs to the language mappings documentation.

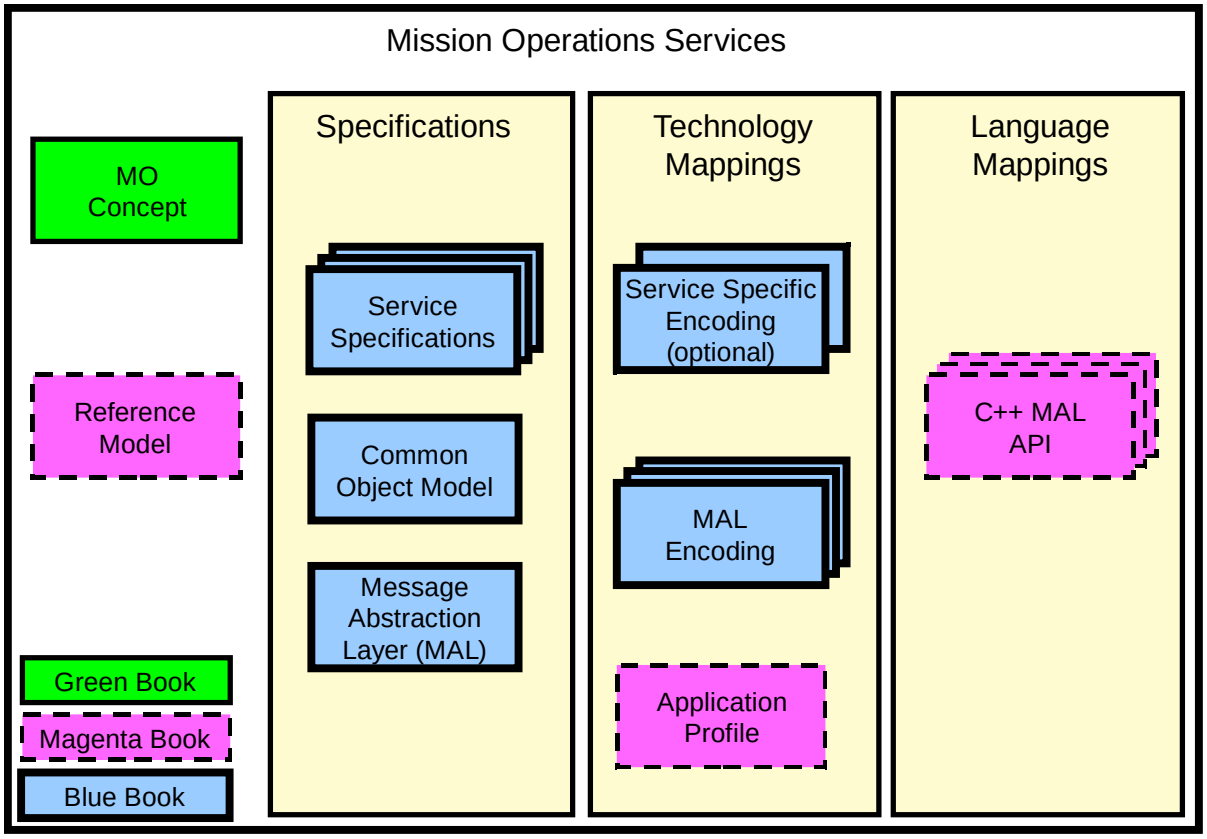

Figure 2-1: Mission Operations Services Concept Document Set

For each programming language there is only required a single mapping of the MAL to that language as the abstract services are defined in terms of the MAL and therefore their language-specific API is derived from the mapping:

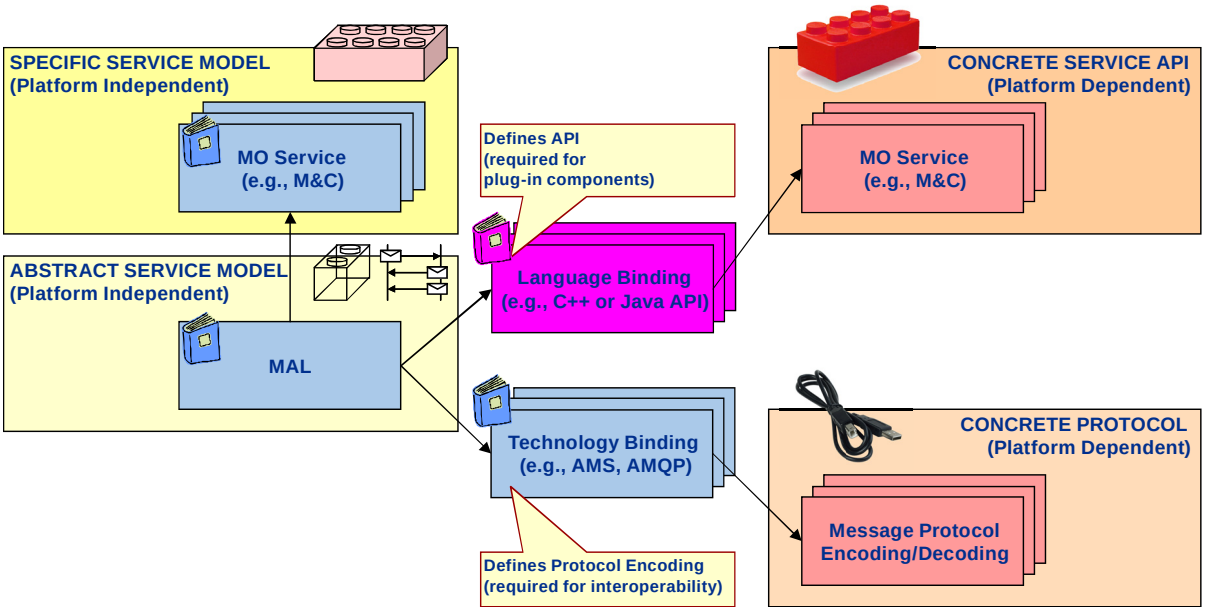

Figure 2-2: Relationship of MO Books

This Recommended Practice defines a mapping from the abstract notation of the MAL to an unambiguous C++ API, more specifically:

a) how the specific MAL abstract services are mapped to C++;

b) how any service errors or issues shall be communicated to higher layers;

c) the physical representation of the abstract MAL messages at the interface necessary to constitute the operation templates;

d) the mapping of the message structure rules for C++;

e) the physical representation of the abstract MAL data types in C++.

It does not specify:

a) individual application services, implementations or products;

b) the implementation of entities or interfaces within real systems;

c) the methods or technologies required to acquire data;

d) the management activities required to schedule a service;

e) the representation of any service specific Protocol Data Units (PDUs) or operations (this is derived from the C++ transform defined in this document).

2.2 MO SERVICE FRAMEWORK C++ MAPPING

The CCSDS Spacecraft Monitoring & Control (SM&C) working group has developed a concept for a Mission Operations (MO) Service Framework, which follows the principles of Service Oriented Architectures. It defines two important aspects, the first is a protocol for interaction between two separate entities, and the second is a framework of common services providing functionality common to most uses of the service framework.

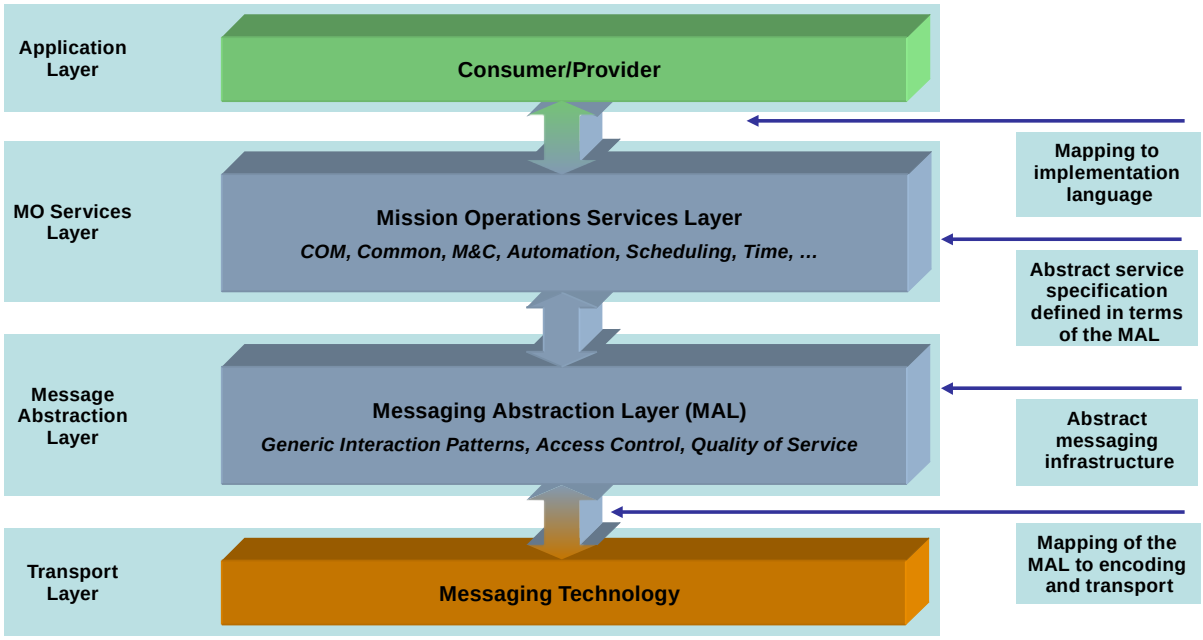

Figure 2-3: Overview of the Mission Operations Service Framework

This Recommended Practice defines the following C++ mapping of the MO Service Framework:

a) the application layer is mapped to C++ application modules consuming and providing MO services;

b) the MO services layer is mapped to service stubs and skeletons enabling the application to consume and provide MO services on top of the MAL;

c) the MAL Interaction Patterns and Quality of Service features are mapped to a MAL module;

d) the MAL access control feature is mapped to an access control module used by the MAL module;

e) the transport layer is mapped to a transport module in charge of using a messaging technology;

f) the transport module can manage the MAL message encoding internally or delegate this function to an encoding module.

The layer diagram displayed in figure 2-4 presents the C++ modules resulting from the MO Service Framework mapping. It also gives the name of the programming interfaces used and implemented by the modules.

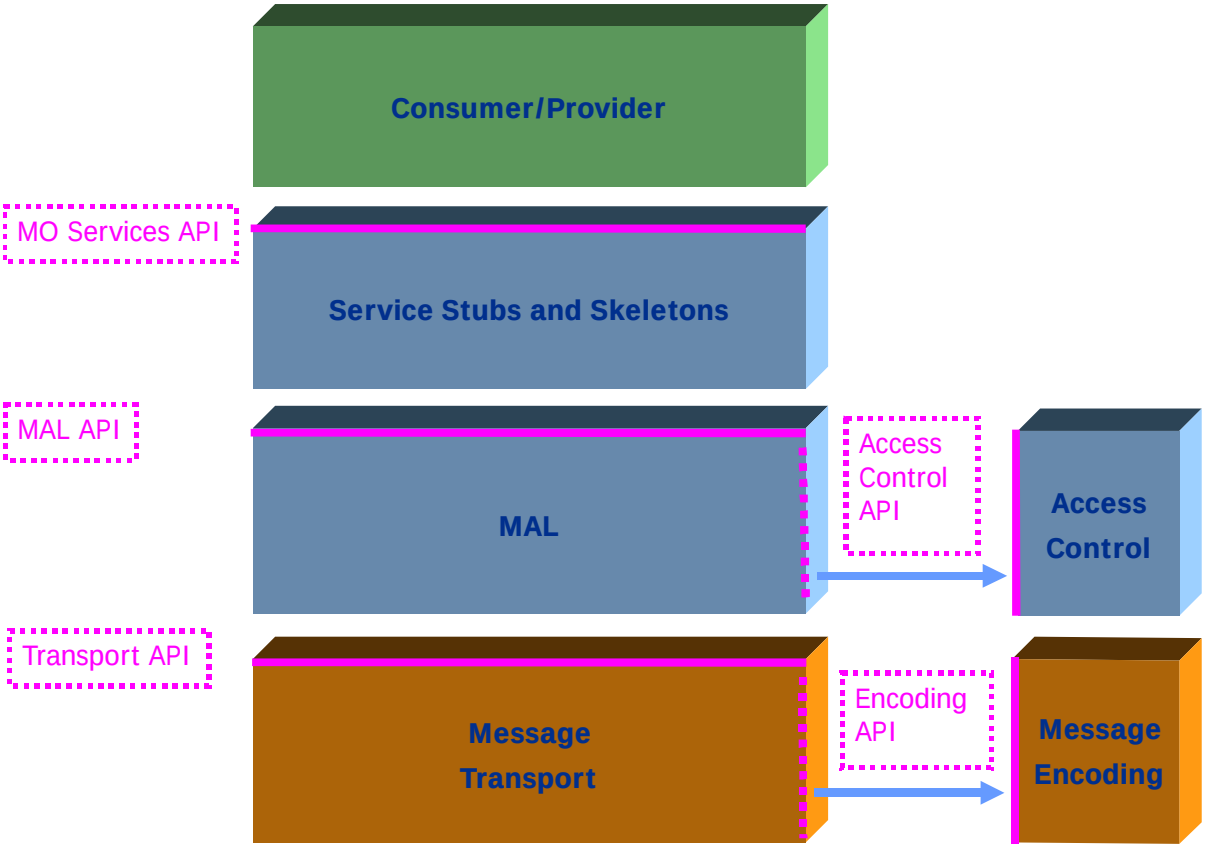

Figure 2-4: MO Framework C++ Mapping

The programming interfaces are:

a) MAL API;

b) MO Services API generated from the Service Mapping;

c) Transport API;

d) Access Control API;

e) Encoding API.

The MO Services API is derived from the services specifications and the C++ Service Mapping rules.

2.3 MAPPING FROM MAL DOCUMENT

This Recommended Practice defines a mapping from the abstract notation of the MAL to an unambiguous C++ API. The structure of the MAL document (reference [1]) is mapped to this document as follows:

a) the subsections ‘Transaction Handling’ (3.2), ‘State Transitions’ (3.3), ‘Message Composition’ (3.4), ‘MAL Service Interface’ (3.5) are mapped to the interaction pattern handling methods of the service consumer, provider and broker defined in the section ‘MAL API’ (section 3);

b) the sections ‘MAL Data Type Specification’ (section 4) and ‘MAL Errors’ (section 5) are mapped to the section ‘MAL API’ and ‘Service Mapping’ (section 4);

   1) the C++ representation of the MAL fundamental and attribute data types is defined in the section ‘MAL API’;

   2) the C++ representation of the MAL data structures is generated according to the section ‘Service Mapping’ except for the structures that need a specific mapping which is defined in the section ‘MAL API’;

   3) the C++ representation of the MAL errors is generated according to the ‘Service Mapping’;

c) the section ‘Service Specification XML’ (section 6) is mapped to the section ‘Service Mapping’;

d) the subsection ‘Transport Interface’ (3.7) is mapped to the section ‘Transport API’ (section 5);

e) the subsection ‘Access Control Interface’ (3.6) is mapped to the section ‘Access Control API’ (section 6).

The Mission Operations Reference Model (reference [B2] describes an encoding component that is responsible for the conversion from the abstract message format of the MAL into the on-the-wire representation used by the transport layer. The encoding component is mapped to this document as follows:

a) the encoding of the message header is internally handled by the transport layer;

b) the encoding of the message body is handled either internally by the transport layer or externally by an encoding module separated from the transport layer by the encoding API.

3 MAL API

3.1 GENERAL

3.1.1 The MAL API shall define the interfaces and classes required by the MAL service interface defined in reference [1].

3.1.2 The MAL API shall comprise the following levels:

a) the generic API used by every MAL clients;

b) the data structures and related interfaces;

c) the service consumer API;

d) the service provider API;

e) the broker API.

3.1.3 The main interfaces provided by the API shall be those listed in table 3-1.

Table 3-1: API Main Interfaces

API level	Interface name	Description
Generic	MALContextFactory	MALContext factory
	MALContext	Context enabling a client to use the communication functions provided by the MAL layer
Consumer	MALConsumerManager	MALConsumer factory and activator
	MALConsumer	Communication context enabling a client to initiate interaction patterns
Provider	MALProviderManager	MALProvider factory and activator.
	MALProvider	Execution context of a service provider handling all the interaction patterns and initiating the Publish/Subscribe interaction pattern as a publisher
Broker	MALBrokerManager	MALBroker factory and activator
	MALBroker	Execution context of a shared broker

NOTE – Figure 3-1 gives an overview of the relationships between the main interfaces.

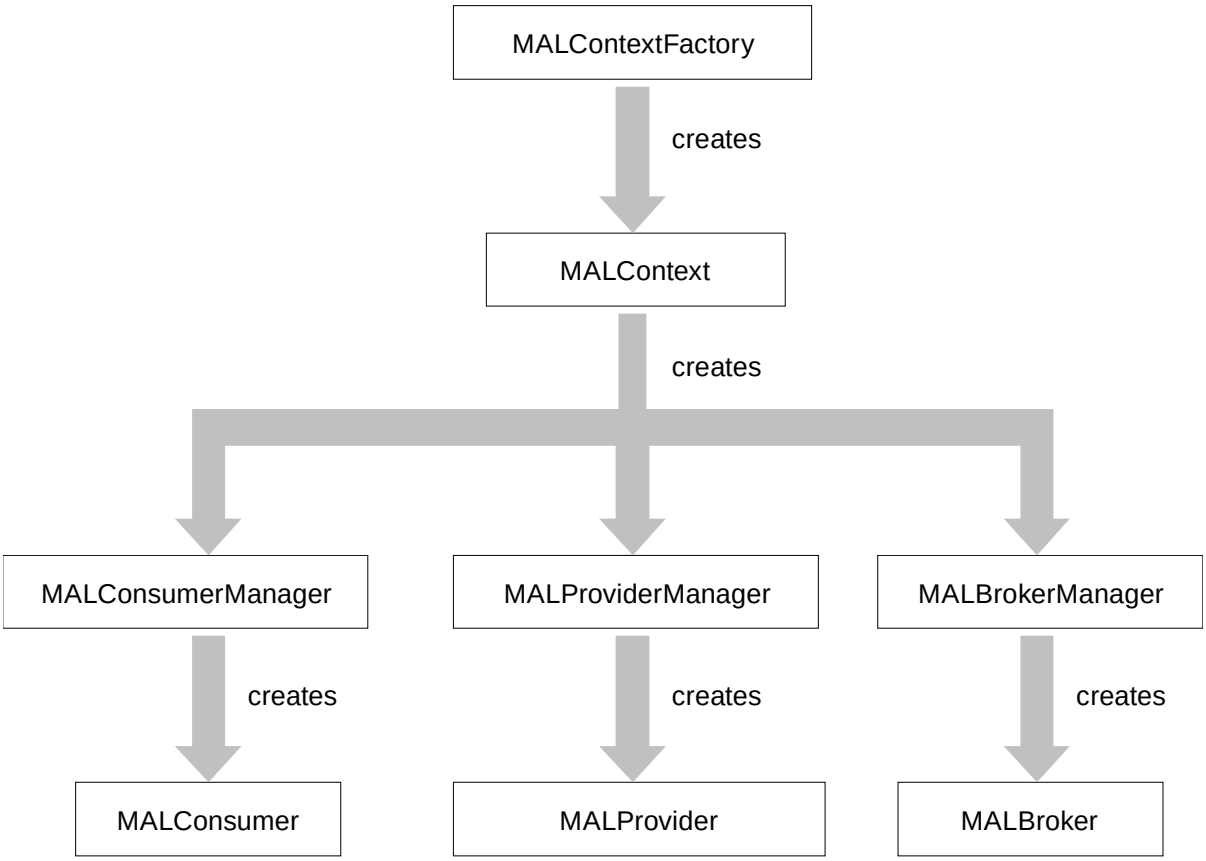

Figure 3-1: Relationships between the API Main Interfaces

3.1.4 The namespaces provided by the API shall be:

a) the MAL namespace: mo::mal;

b) the data structures namespace: mo::mal::structures;

c) the consumer namespace: mo::mal::consumer;

d) the provider namespace: mo::mal::provider;

e) the broker namespace: mo::mal::broker.

3.1.5 The MAL namespace name shall be added to the SANA registry C++BindingNamespace and shall refer to the MAL C++ API document ‘CCSDS 523.1-M-1’.

3.2 MAL NAMESPACE

3.2.1 OVERVIEW

This subsection defines the content of the generic MAL namespace:

`mo::mal`

3.2.2 MALCONTEXTFACTORY

3.2.2.1 Definition

3.2.2.1.1 A MALContextFactory class shall be defined in order to enable a MAL client to instantiate and configure a MALContext.

3.2.2.1.2 The MALContextFactory class shall provide static repositories:

a) the MALContext factory class repository shall map the MALContextFactory implementation classes to their class names;

b) the MALArea repository shall map the area names and numbers to the MALArea descriptions;

c) the error repository shall map the error codes to the error names.

3.2.2.1.3 The MALContextFactory class shall provide a MALElementFactoryRegistry.

3.2.2.2 Class Initialization

The MALContextFactory class initialization shall:

a) create the static repositories;

b) instantiate the MALElementFactoryRegistry;

c) call the static method ‘init’ provided by the MALHelper by passing the MALElementFactoryRegistry.

3.2.2.3 Factory Class Registration

3.2.2.3.1 The MALContextFactory class shall provide a static method ‘registerFactoryClass’ in order to register the class of a specific MALContextFactory.

3.2.2.3.2 The signature of the public method ‘registerFactoryClass’ shall be:

```
static void
registerFactoryClass(const shared_ptr<MALContextFactory>& factoryClass)
```

3.2.2.3.3 The parameter of the method ‘registerFactoryClass’ shall be assigned as described in table 3-2.

Table 3-2: MALContextFactory ‘registerFactoryClass’ Parameter

Parameter	Description
factoryClass	shared_ptr reference to class extending MALContextFactory

3.2.2.3.4 The method ‘registerFactoryClass’ shall store the class in the factory class repository.

3.2.2.4 Factory Class Deregistration

3.2.2.4.1 The MALContextFactory class shall provide a static method ‘deregisterFactoryClass’ in order to deregister the class of a specific MALContextFactory.

3.2.2.4.2 The signature of the public method ‘deregisterFactoryClass’ shall be:

```
static void
deregisterFactoryClass(shared_ptr<MALContextFactory>& factoryClass)
```

3.2.2.4.3 The parameter of the method ‘deregisterFactoryClass’ shall be assigned as described in table 3-3.

Table 3-3: MALContextFactory ‘deregisterFactoryClass’ Parameter

Parameter	Description
factoryClass	shared_ptr reference to class extending MALContextFactory

3.2.2.4.4 The method ‘deregisterFactoryClass’ shall remove the class from the factory class repository.

3.2.2.4.5 The method ‘deregisterFactoryClass’ shall do nothing if the class is not found in the factory class repository.

3.2.2.5 MALContextFactory Creation

3.2.2.5.1 A static method ‘newFactory’ shall be provided in order to return a MALContextFactory instance.

3.2.2.5.2 The signature of the public method ‘newFactory’ shall be:

```
template<typename ContextFactoryType>
static shared_ptr<MALContextFactory> newFactory()
```

3.2.2.5.3 The method ‘newFactory’ shall use a C++ template to specify the specific type of the MALContextFactory implementation class to instantiate.

3.2.2.5.4 The method ‘newFactory’ shall lookup the MALContextFactory implementation class from the factory class repository.

3.2.2.5.5 If the MALContextFactory implementation class is not found in the factory class repository, then it shall create a new MALContextFactory of the specific type in the template.

3.2.2.5.6 The method ‘newFactory’ shall not return the value NULL.

3.2.2.5.7 If no MALContextFactory can be returned, then a MALException shall be raised.

3.2.2.6 MALContext Instantiation

3.2.2.6.1 The MALContextFactory class shall provide an abstract public method ‘createMALContext’ in order to instantiate a MALContext.

3.2.2.6.2 The signature of the method ‘createMALContext’ shall be:

```
shared_ptr<MALContext> createMALContext(const MALProperties& properties)
```

3.2.2.6.3 The parameter of the method ‘createMALContext’ shall be assigned as described in table 3-4.

Table 3-4: MALContextFactory ‘createMALContext’ Parameter

Parameter	Description
properties	Properties required by the specific MALContext implementation

3.2.2.6.4 The method ‘createMALContext’ shall not return the value NULL.

3.2.2.6.5 If no MALContext can be returned, then a MALException shall be raised.

3.2.2.6.6 The method ‘createMALContext’ shall be implemented by the specific factory class.

3.2.2.7 MALArea Registration

3.2.2.7.1 The MALContextFactory class shall provide a static method ‘registerArea’ in order to register MALAreas.

3.2.2.7.2 The signature of the method ‘registerArea’ shall be:

```
static void registerArea(const shared_ptr<MALArea>& area)
```

3.2.2.7.3 The parameter of the method ‘registerArea’ shall be assigned as described in table 3-5.

Table 3-5: MALContextFactory ‘registerArea’ Parameter

Parameter	Description
area	MALArea to register

3.2.2.7.4 If a MALArea having the same name and the same version has already been registered and if the registered MALArea is not the same object as the MALArea to register, then a MALException shall be raised.

3.2.2.8 MALArea Query

3.2.2.8.1 The MALContextFactory class shall provide two static methods ‘lookupArea’ in order to look up a MALArea from:

a) the name and the version of the area;

b) the number and the version of the area.

3.2.2.8.2 The signatures of the method ‘lookupArea’ shall be:

```
static shared_ptr<MALArea> lookupArea(
      const Identifier& areaName,
      const uint8_t& areaVersion)

static shared_ptr<MALArea> lookupArea(
      const uint16_t& areaNumber,
      const uint8_t& areaVersion)
```

3.2.2.8.3 The parameters of the method ‘lookupArea’ shall be assigned as described in table 3-6.

Table 3-6: MALContextFactory 'lookupArea' Parameters

Parameter	Description
areaName	Name of the area to look up
areaNumber	Number of the area to look up
areaVersion	Version of the area to look up

3.2.2.8.4 The method 'lookupArea' shall return NULL if the MALArea is not found.

3.2.2.9 Error Registration

3.2.2.9.1 The MALContextFactory class shall provide a static method 'registerError' in order to register errors.

3.2.2.9.2 The signature of the method 'registerError' shall be:

```
static void registerError(
            const uint32_t& errorNumber,
            const Identifier& errorName)
```

3.2.2.9.3 The parameters of the method 'registerError' shall be assigned as described in table 3-7.

Table 3-7: MALContextFactory 'registerError' Parameters

Parameter	Description
errorNumber	Error code
errorName	Name of the error

3.2.2.9.4 If an error having the same code is already registered, then a MALException shall be raised.

3.2.2.10 Error Query

3.2.2.10.1 The MALContextFactory class shall provide a static method 'lookupError' in order to look up an error name from an error code.

3.2.2.10.2 The signature of the method 'lookupError' shall be:

```
static Identifier lookupError(const uint32_t& errorNumber)
```

3.2.2.10.3 The parameter of the method ‘lookupError’ shall be assigned as described in table 3-8.

Table 3-8: MALContextFactory ‘lookupError’ Parameter

Parameter	Description
errorNumber	Code of the error to look up

3.2.2.10.4 The method ‘lookupError’ shall return NULL if the error is not found.

3.2.2.11 MALElementFactoryRegistry Getter

3.2.2.11.1 The MALContextFactory class shall provide a static method ‘getElementFactoryRegistry’ in order to get the MALElementFactoryRegistry.

3.2.2.11.2 The signature of the method ‘getElementFactoryRegistry’ shall be:

```
static MALElementFactoryRegistry getElementFactoryRegistry()
```

3.2.2.11.3 The method ‘getElementFactoryRegistry’ shall return the MALElementFactoryRegistry that has been instantiated during the MALContextFactory class initialization.

3.2.3 MALCONTEXT

3.2.3.1 Definition

3.2.3.1.1 A MALContext interface shall be defined in order to enable a client to consume and/or provide services.

3.2.3.1.2 A MALContext shall be a factory of:

a) MALConsumerManager;

b) MALProviderManager;

c) MALBrokerManager.

3.2.3.2 MALContext Creation

A MALContext shall be created by calling the method ‘createMALContext’ provided by the MALContextFactory.

NOTE – Because of the infrastructure and communication setup done when a MALContext is created, a MALContext is a heavyweight object. Most clients will initiate and handle all the interactions with a single MALContext. However, some applications may use several MALContext objects.

3.2.3.3 MALConsumerManager Creation

3.2.3.3.1 A method ‘createConsumerManager’ shall be defined in order to enable to create and activate multiple MALConsumerManagers.

3.2.3.3.2 The signature of the method ‘createConsumerManager’ shall be:

```
shared_ptr<MALConsumerManager> createConsumerManager()
```

3.2.3.3.3 The method ‘createConsumerManager’ shall not return the value NULL.

3.2.3.3.4 If no MALConsumerManager can be returned, then a MALException shall be raised.

3.2.3.3.5 If the MALContext is closed, then a MALException shall be raised.

3.2.3.4 MALProviderManager Creation

3.2.3.4.1 A method ‘createProviderManager’ shall be defined in order to enable creation and activation of multiple MALProviderManagers.

3.2.3.4.2 The signature of the method ‘createProviderManager’ shall be:

```
shared_ptr<MALProviderManager> createProviderManager()
```

3.2.3.4.3 The method ‘createProviderManager’ shall not return the value NULL.

3.2.3.4.4 If no MALProviderManager can be returned, then a MALException shall be raised.

3.2.3.4.5 If the MALContext is closed, then a MALException shall be raised.

3.2.3.5 MALBrokerManager Creation

3.2.3.5.1 A method ‘createBrokerManager’ shall be defined in order to enable creation and activation of multiple MALBrokerManagers.

3.2.3.5.2 The signature of the method ‘createBrokerManager’ shall be:

```
shared_ptr<MALBrokerManager> createBrokerManager()
```

3.2.3.5.3 The method ‘createBrokerManager’ shall not return the value NULL.

3.2.3.5.4 If no MALBrokerManager can be returned, then a MALException shall be raised.

3.2.3.5.5 If the MALContext is closed, then a MALException shall be raised.

3.2.3.6 Get a Transport

3.2.3.6.1 Two methods ‘getTransport’ shall be defined in order to get the reference of a MALTransport:

a) the first method ‘getTransport’ shall take as a parameter the name of the protocol to be supported by the MALTransport;

b) the second method ‘getTransport’ shall take as a parameter a URI which protocol is to be supported by the MALTransport.

3.2.3.6.2 The signatures of the method ‘getTransport’ shall be:

```
shared_ptr<MALTransport> getTransport(const string& protocol)

shared_ptr<MALTransport> getTransport(const URI& uri)
```

3.2.3.6.3 The parameters of the method ‘getTransport’ shall be assigned as described in table 3-9.

Table 3-9: MALContext ‘getTransport’ Parameters

Parameter	Description
protocol	Name of the protocol to be supported
uri	URI which protocol is to be supported

3.2.3.6.4 The method ‘getTransport’ shall return a unique MALTransport instance for a given protocol.

3.2.3.6.5 If there is still no instance of the MALTransport for the specified protocol, then such an instance shall be created.

3.2.3.6.6 If no MALTransport can be returned, then a MALException shall be raised.

3.2.3.6.7 If the MALContext is closed, then a MALException shall be raised.

3.2.3.7 Get the Access Control

3.2.3.7.1 A method ‘getAccessControl’ shall be defined in order to get the reference of the MALAccessControl used by the MALContext.

3.2.3.7.2 The signature of the method ‘getAccessControl’ shall be:

```
shared_ptr<MALAccessControl> getAccessControl()
```

3.2.3.7.3 If no MALAccessControl can be returned, then a MALException shall be raised.

3.2.3.8 Close the MALContext

3.2.3.8.1 A method ‘close’ shall be defined in order to synchronously close all the MALConsumerManager, MALProviderManager, and MALBrokerManager instances that have been created by this MALContext.

NOTE – In order to release the resources allocated by a MALContext, clients should close it when it is no longer used. The MALContext should be explicitly closed because it may be referenced by internal threads, and therefore the memory referenced by the shared_ptr is never released.

3.2.3.8.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.2.3.8.3 If an internal error occurs, then a MALException shall be raised.

3.2.4 MALSERVICE

3.2.4.1 Definition

3.2.4.1.1 A MALService class shall be defined in order to represent the specification of a service (reference [1]).

3.2.4.1.2 The MALService class shall define the attributes specified in table 3-10.

Table 3-10: MALService Attributes

Attribute	Type
number	uint16_t
name	Identifier
version	uint8_t
area	MALArea

3.2.4.2 Creation

3.2.4.2.1 The MALService constructor signature shall be:

```
public MALService(
      const uint16_t& number,
      const Identifier& name)
```

3.2.4.2.2 The MALService constructor parameters shall be assigned as described in table 3-11.

Table 3-11: MALService Constructor Parameters

Parameter	Description
number	Number of the service
name	Name of the service

3.2.4.2.3 The attribute ‘number’ shall be assigned with the value of the parameter ‘number’.

3.2.4.2.4 The attribute ‘name’ shall be assigned with the value of the parameter ‘name’.

3.2.4.3 Getters

3.2.4.3.1 The MALService class shall provide the following public getters:

a) Name getter:

```
Identifier getName()
```

b) Area getter:

```
MALArea getArea()
```

c) Number getter:

```
public uint16_t getNumber()
```

d) Operation getters:

1) Getter from the operation number:

```
shared_ptr<MALOperation> getOperationByNumber(
                           const uint16_t& opNumber)
```

2) Getter from the operation name:

```
shared_ptr<MALOperation> getOperationByName(
                           const Identifier& opName)
```

3) Getter from the capability set:

```
vector<shared_ptr<MALOperation>> getOperationsByCapabilitySet(
                           const uint16_t& capabilitySet)
```

4) Send IP operation getter:

```
vector<shared_ptr<MALSendOperation>> getSendOperations()
```

5) Submit IP operation getter:

```
vector<shared_ptr<MALSubmitOperation>> getSubmitOperations()
```

6) Request IP operation getter:

```
vector<shared_ptr<MALRequestOperation>> getRequestOperations()
```

7) Invoke IP operation getter:

```
vector<shared_ptr<MALInvokeOperation>> getInvokeOperations()
```

8) Progress IP operation getter:

```
vector<shared_ptr<MALProgressOperation>> getProgressOperations()
```

9) Publish-Subscribe IP operation getter:

```
vector<shared_ptr<MALPubSubOperation>> getPubSubOperations()
```

3.2.4.3.2 The methods ‘getOperationByNumber’ and ‘getOperationByName’ shall return NULL if the operation is not found.

3.2.4.3.3 The methods ‘getOperationsByCapabilitySet’ and ‘get<<Ip>>Operations’ shall return an empty vector if no result is found.

3.2.4.4 Set the Area

3.2.4.4.1 A method ‘setArea’ shall be defined in order to set the value of the attribute ‘area’.

3.2.4.4.2 The method ‘setArea’ shall be called only by the MALArea method ‘addService’.

3.2.4.4.3 The method ‘setArea’ control access level shall be ‘namespace’.

3.2.4.4.4 The signature of the method ‘setArea’ shall be:

```
void setArea(const shared_ptr<MALArea>& area)
```

3.2.4.4.5 The parameter of the method ‘setArea’ shall be assigned as described in table 3-12.

Table 3-12: MALService ‘setArea’ Parameter

Parameter	Description
area	MALArea that owns this MALService

3.2.4.4.6 The attribute ‘area’ shall be assigned with the value of the parameter ‘area’.

3.2.4.5 Add an Operation

3.2.4.5.1 A method ‘addOperation’ shall be defined in order to enable population of the list of operations provided by a service.

3.2.4.5.2 The signature of the method ‘addOperation’ shall be:

```
virtual void addOperation(shared_ptr<MALOperation>& operation)
```

3.2.4.5.3 The parameter of the method ‘addOperation’ shall be assigned as described in table 3-13.

Table 3-13: MALService ‘addOperation’ Parameter

Parameter	Description
operation	MALOperation to be added into this MALService

3.2.4.5.4 The method ‘setService’ provided by the MALOperation shall be called with the reference of this MALService as a parameter.

3.2.5 MALOPERATION

3.2.5.1 Definition

3.2.5.1.1 A MALOperation class shall be defined in order to represent the specification of an operation provided by a service (reference [1]).

3.2.5.1.2 The MALOperation class shall be abstract.

3.2.5.1.3 The MALOperation class shall be extended by one class defined for each interaction pattern.

3.2.5.1.4 The MALOperation class shall define the attributes specified in table 3-14.

Table 3-14: MALOperation Attributes

Attribute	Type
number	uint16_t
name	Identifier
replayable	bool
interactionType	InteractionType
capabilitySet	uint16_t
service	MALService

3.2.5.2 Creation

3.2.5.2.1 The constructor signature of the MALOperation class shall be:

```
public MALOperation(
      const uint16_t& number,
      const Identifier& name,
      const bool& replayable,
      const InteractionType& interactionType,
      const uint16_t& capabilitySet)
```

3.2.5.2.2 The MALOperation constructor parameters shall be assigned as described in table 3-15.

Table 3-15: MALOperation Constructor Parameters

Parameter	Description
number	Number of the operation
name	Name of the operation
replayable	bool that indicates whether the operation is replayable or not
interactionType	Interaction type of the operation
capabilitySet	Capability set number of the operation

3.2.5.2.3 The attribute ‘number’ shall be assigned with the value of the parameter ‘number’.

3.2.5.2.4 The attribute ‘name’ shall be assigned with the value of the parameter ‘name’.

3.2.5.2.5 The attribute ‘replayable’ shall be assigned with the value of the parameter ‘replayable’.

3.2.5.2.6 The attribute ‘interactionType’ shall be assigned according to the parameter ‘interactionType’.

3.2.5.2.7 The attribute ‘capabilitySet’ shall be assigned with the value of the parameter ‘capabilitySet’.

3.2.5.3 Getters

The MALOperation class shall provide the following getters:

a) Name getter:

```
Identifier getName()
```

b) Number getter:

```
uint16_t getNumber()
```

c) InteractionType getter:

```
InteractionType getInteractionType()
```

d) Replayable getter:

```
bool isReplayable()
```

e) Service getter:

```
shared_ptr<MALService> getService()
```

f) CapabilitySet getter:

```
uint16_t getCapabilitySet()
```

3.2.5.4 Get a MALOperationStage

3.2.5.4.1 An abstract method ‘getOperationStage’ shall be defined in order to return a MALOperationStage from its number.

3.2.5.4.2 The signature of the method ‘getOperationStage’ shall be:

```
shared_ptr<MALOperationStage> getOperationStage(const uint8_t& stageNumber)
```

3.2.5.4.3 The parameter of the method ‘getOperationStage’ shall be assigned as described in table 3-16.

Table 3-16: MALOperation ‘getOperationStage’ Parameter

Parameter	Description
stageNumber	Number of the MALOperationStage

3.2.5.4.4 The method ‘getOperationStage’ shall return NULL only if the stage does not exist for that operation IP.

3.2.5.4.5 The method ‘getOperationStage’ shall be implemented by the specific MAL<<Ip>>Operation classes.

3.2.5.5 Set the Service

3.2.5.5.1 A method ‘setService’ shall be defined in order to set the value of the attribute ‘service’.

3.2.5.5.2 The method ‘setService’ shall be called by the MALService constructor.

3.2.5.5.3 The ‘setService’ control access level shall be ‘namespace’.

3.2.5.5.4 The signature of the method ‘setService’ shall be:

```
void setService(const shared_ptr<MALService>& service)
```

3.2.5.5.5 The parameter of the method ‘setService’ shall be assigned as described in table 3-17.

Table 3-17: MALOperation 'setService' Parameter

Parameter	Description
service	Service that owns this operation

3.2.5.5.6 The attribute 'service' shall be assigned with the value of the parameter 'service'.

3.2.6 MAL<<IP>>OPERATION

3.2.6.1 Definition

3.2.6.1.1 A MAL<<Ip>>Operation class shall be defined for each interaction pattern.

3.2.6.1.2 The MAL<<Ip>>Operation class names shall be:

a) MALSendOperation;

b) MALSubmitOperation;

c) MALRequestOperation;

d) MALInvokeOperation;

e) MALProgressOperation;

f) MALPubSubOperation.

3.2.6.2 Interaction Stages Constants

3.2.6.2.1 The MAL<<Ip>>Operation shall define constants giving the interaction stage values for the stages listed in table 3-18.

Table 3-18: Interaction Stages Constants

IP	stage name	stage value (HEX)
Send	-	-
Submit	SUBMIT	0x1
	SUBMIT_ACK	0x2
Request	REQUEST	0x1
	REQUEST_RESPONSE	0x2
Invoke	INVOKE	0x1
	INVOKE_ACK	0x2
	INVOKE_RESPONSE	0x3
Progress	PROGRESS	0x1
	PROGRESS_ACK	0x2
	PROGRESS_UPDATE	0x3
	PROGRESS_RESPONSE	0x4
PubSub	REGISTER	0x1
	REGISTER_ACK	0x2
	PUBLISH_REGISTER	0x3
	PUBLISH_REGISTER_ACK	0x4
	PUBLISH	0x5
	NOTIFY	0x6
	DEREGISTER	0x7
	DEREGISTER_ACK	0x8
	PUBLISH_DEREGISTER	0x9
	PUBLISH_DEREGISTER_ACK	0xA

3.2.6.2.2 The constants shall be declared as specified below:

```
static const char _<<STAGE NAME>>_STAGE =
                                    static_cast<char>(0x<<stage value>>)

uint8_t <<STAGE NAME>>_STAGE = (uint8_t) _<<STAGE NAME>>_STAGE
```

3.2.6.3 Creation

3.2.6.3.1 For every interaction type except PUBLISH-SUBSCRIBE, the constructor signature of the MAL<<Ip>>Operation shall be:

```
MAL<<Ip>>Operation(
      const uint16_t& number,
      const Identifier& name,
      const bool& replayable,
```

```
      const uint16_t& capabilitySet,
      const shared_ptr<MALOperationStage>& stage)
```

3.2.6.3.2 The variable ‘stage’ value shall enable passing of a MALOperationStage parameter for each interaction stage.

3.2.6.3.3 The operation stages shall be declared according to the IP variable as described in table 3-19.

Table 3-19: Operation Stages

IP	Stages
Send	MALOperationStage sendStage
Submit	MALOperationStage submitStage
Request	MALOperationStage requestStage, MALOperationStage responseStage
Invoke	MALOperationStage invokeStage, MALOperationStage invokeAckStage, MALOperationStage invokeResponseStage
Progress	MALOperationStage progressStage, MALOperationStage progressAckStage, MALOperationStage progressUpdateStage, MALOperationStage progressResponseStage

3.2.6.3.4 The MALPubSubOperation constructor signature shall be:

```
public MALPubSubOperation(
      const uint16_t& number,
      const Identifier& name,
      const bool& replayable,
      const uint16_t& capabilitySet,
      const shared_ptr<vector<long long>>& updateListShortForms,
      const shared_ptr<vector<long long>>& lastUpdateListShortForms)
```

3.2.6.3.5 The MAL<<Ip>>Operation constructor parameters shall be assigned as described in table 3-20.

Table 3-20: MAL<<Ip>>Operation Constructor Parameters

Parameter	Description
number	Number of the operation
name	Name of the operation

Parameter	Description
replayable	bool that indicates whether the operation is replayable or not
capabilitySet	Capability set number of the operation
sendStage	MALOperationStage representing the first stage of a SEND operation
submitStage	MALOperationStage representing the first stage of a SUBMIT operation
requestStage	MALOperationStage representing the first stage of a REQUEST operation
responseStage	MALOperationStage representing the first stage of a REQUEST operation
invokeStage	MALOperationStage representing the first stage of an INVOKE operation
invokeAckStage	MALOperationStage representing the ACK stage of an INVOKE operation
invokeResponseStage	MALOperationStage representing the RESPONSE stage of an INVOKE operation
progressStage	MALOperationStage representing the first stage of a PROGRESS operation
progressAckStage	MALOperationStage representing the ACK stage of a PROGRESS operation
progressUpdateStage	MALOperationStage representing the UPDATE stage of a PROGRESS operation
progressResponseStage	MALOperationStage representing the RESPONSE stage of a PROGRESS operation
updateListShortForms	Absolute short forms of the update lists transmitted by the PUBLISH/NOTIFY message of a PUBLISH-SUBSCRIBE operation
lastUpdateListShortForms	Absolute short forms of the update lists that can be assigned to the last element of the PUBLISH/NOTIFY message body

3.2.6.3.6 The method ‘setOperation’ provided by the MALOperationStage parameters shall be called with the reference of this MALOperation as a parameter.

3.2.6.3.7 If the PUBLISH/NOTIFY message body is empty, then the ‘updateListShortForms’ and the ‘lastUpdateListShortForms’ shall be empty arrays.

3.2.6.3.8 The ‘updateListShortForms’ shall not contain the value NULL except in the last item of the vector if the declared type of the last element is abstract.

3.2.6.4 Get a MALOperationStage

3.2.6.4.1 The method ‘getOperationStage’ inherited from MALOperation shall be implemented by returning the MALOperationStage representing the specified stage.

3.2.6.4.2 MALSendOperation shall return the MALOperationStage representing the Send stage.

3.2.6.4.3 MALPubSubOperation shall return one MALOperationStage for each Publish-Subscribe stage.

3.2.7 MALOPERATIONSTAGE

3.2.7.1 Definition

3.2.7.1.1 A MALOperationStage class shall be defined in order to represent all the element types used by an operation during a stage as specified by the reference [1].

3.2.7.1.2 The MALOperationStage class shall define the attributes specified in table 3-21.

Table 3-21: MALOperationStage Attributes

Attribute	Type
number	uint8_t
operation	MALOperation
elementShortForms	std::vector<long long>
lastElementShortForms	std::vector<long long>

3.2.7.2 Creation

3.2.7.2.1 The MALOperationStage constructor signature shall be:

```
public MALOperationStage(
      const uint8_t& number,
      const vector<long long>& elementShortForms,
      const vector<long long>& lastElementShortForms )
```

3.2.7.2.2 The MALOperationStage constructor parameters shall be assigned as described in table 3-22.

Table 3-22: MALOperationStage Constructor Parameters

Parameter	Description
number	Number of the interaction stage
elementShortForms	Short forms of all the element types declared by the message body
lastElementShortForms	Short forms of the types that can be used for the last element of the message body in case of polymorphism

3.2.7.2.3 If the message body is empty, then the parameters ‘elementShortForms’ and ‘lastElementShortForms’ shall be an empty vector.

3.2.7.2.4 The parameter ‘elementShortForms’ shall not contain the value NULL except in the last item of the vector if the declared type of the last element is abstract.

3.2.7.2.5 If the MALOperationStage represents the Publish-Subscribe Publish stage then the ‘elementShortForms’ shall contain the short forms of UpdateHeaderList and the update list types.

3.2.7.2.6 If the MALOperationStage represents the Publish-Subscribe Notify stage then the ‘elementShortForms’ shall contain the short forms of Identifier, UpdateHeaderList and the update lists.

3.2.7.2.7 The attribute ‘number’ shall be assigned with the value of the parameter ‘number’.

3.2.7.2.8 The attribute ‘elementShortForms’ shall be assigned with the value of the parameter ‘elementShortForms’.

3.2.7.2.9 The attribute ‘lastElementShortForms’ shall be assigned with the value of the parameter ‘lastElementShortForms’.

3.2.7.3 Getters and setters

The MALOperation class shall provide the following getters and setters:

a) Number getter:

```
uint8_t getNumber()
```

b) Body element short forms getter:

```
vector<long long> getElementShortForms()
```

c) Last element short forms getter and setter:

```
vector<long long> getLastElementShortForms()

void setLastElementShortForms(const vector<long long>& shortForms)
```

d) Operation getter:

```
shared_ptr<MALOperation> getOperation()
```

3.2.7.4 Set the Operation

3.2.7.4.1 A method ‘setOperation’ shall be defined in order to set the value of the attribute ‘operation’.

3.2.7.4.2 The method ‘setOperation’ shall be called by the MALOperation constructor.

3.2.7.4.3 The method ‘setOperation’ control access level shall be ‘namespace’.

3.2.7.4.4 The signature of the method ‘setOperation’ shall be:

```
void setOperation(const shared_ptr<MALOperation>& operation)
```

3.2.7.4.5 The parameter of the method ‘setOperation’ shall be assigned as described in table 3-23.

Table 3-23: MALOperation ‘setOperation’ Parameter

Parameter	Description
operation	Operation that owns this MALOperationStage

3.2.7.4.6 The attribute ‘operation’ shall be assigned with the value of the parameter ‘operation’.

3.2.8 MALAREA

3.2.8.1 Definition

3.2.8.1.1 A MALArea class shall be defined in order to represent the specification of an area of services (reference [1]).

3.2.8.1.2 The MALArea class shall define the attributes specified in table 3-24.

Table 3-24: MALService Attributes

Attribute	Type
number	uint16_t
name	Identifier
version	uint8_t

3.2.8.2 Creation

3.2.8.2.1 The MALArea constructor signature shall be:

```
MALArea(
      const uint16_t& number,
      const Identifier& name,
      const uint8_t& version)
```

3.2.8.2.2 The MALArea constructor parameters shall be assigned as described in table 3-25.

Table 3-25: MALArea Constructor Parameters

Parameter	Description
number	Number of the area
name	Name of the area
version	Version of the area

3.2.8.2.3 The attribute ‘number’ shall be assigned with the value of the parameter ‘number’.

3.2.8.2.4 The attribute ‘name’ shall be assigned with the value of the parameter ‘name’.

3.2.8.2.5 The attribute ‘version’ shall be assigned with the value of the parameter ‘version’.

3.2.8.3 Getters

The MALArea class shall provide the following getters:

a) Number getter:

```
uint16_t getNumber()
```

b) Name getter:

```
Identifier getName()
```

c) Version getter:

```
uint8_t getVersion()
```

d) Services getter:

```
vector<shared_ptr<<MALService>> getServices()
```

e) Service getter by name:

```
shared_ptr<MALService> getServiceByName(
                                const Identifier& serviceName)
```

f) Service getter by number:

```
shared_ptr<MALService> getServiceByNumber(
                                const uint16_t& serviceNumber)
```

3.2.8.4 Add a Service

3.2.8.4.1 A method ‘addService’ shall be defined in order to add a MALService to this MALArea.

3.2.8.4.2 The signature of the method ‘addService’ shall be:

```
void addService(const shared_ptr<MALService>& service)
```

3.2.8.4.3 The parameter of the method ‘addService’ shall be assigned as described in table 3-26.

Table 3-26: MALArea ‘addService’ Parameter

Parameter	Description
service	Service to add to the area

3.2.8.4.4 If a service with the same name or number is already owned by the area, then a MALException shall be raised.

3.2.8.4.5 The method ‘setArea’ provided by the MALService shall be called with the reference of this MALArea as a parameter.

3.2.9 MALHELPER

The MALHelper class shall be generated from the class template defined in 4.8.3 for the area ‘MAL’.

3.2.10 MALSTANDARDERROR

3.2.10.1 Definition

3.2.10.1.1 A MALStandardError class shall be defined in order to represent a MAL error.

3.2.10.2 Creation

3.2.10.2.1 The MALStandardError constructor signature shall be:

```
MALStandardError(
  const uint32_t& errorNumber,
  const std::string& extraInformation)
```

3.2.10.2.2 The MALStandardError constructor parameters shall be assigned as described in table 3-27.

Table 3-27: MALService Constructor Parameters

Parameter	Description
errorNumber	Number of the MAL standard error
extraInformation	Extra information associated with the error

3.2.10.2.3 The parameter ‘extraInformation’ may be NULL.

3.2.10.3 Getters

The MALStandardError class shall provide the following getters:

a) Error number getter:

```
uint32_t getErrorNumber()
```

b) Extra information getter:

```
std::string getExtraInformation()
```

3.2.10.4 Get the Error Name

3.2.10.4.1 A method ‘getErrorName’ shall be defined in order to return the name associated with the error number specified by the ERROR message.

3.2.10.4.2 The signature of the method ‘getErrorName’ shall be:

```
Identifier getErrorName()
```

3.2.10.4.3 The method ‘getErrorName’ shall resolve the name of the error from the MALContextFactory error repository.

3.2.10.5 toString

The method ‘toString’ shall be redefined by returning a String formatted as follows:

a) the first character shall be ‘(’;

b) the string ‘errorNumber=’ shall be appended;

c) the toString representation of the field ‘errorNumber’ shall be appended;

d) the character ‘,’ shall be appended;

e) the string ‘errorName=’ shall be appended;

f) the name of the error shall be resolved from the MALContextFactory error repository and appended;

g) the character ‘,’ shall be appended;

h) the string ‘extraInformation=’ shall be appended;

i) if ‘extraInformation’ is empty, then nothing shall be appended otherwise the toString representation of the field ‘extraInformation’ shall be appended;

j) the last character shall be ‘)’.

3.2.11 MALEXCEPTION

3.2.11.1 Definition

3.2.11.1.1 A MALException class shall be defined in order to enable the MAL API to raise an error that is not a MAL standard error as an exception.

3.2.11.1.2 The MALException class shall inherit from the C++ class std::runtime_error which in turn inherits from std::exception.

3.2.11.2 Creation

3.2.11.2.1 Three MALException constructors shall be defined in order to create a MALException:

a) from a std::string;

b) from a MALStandardError;

c) from another MALException (copy constructor).

3.2.11.2.2 The MALException constructor signatures shall be:

```
MALException(const std::string& message)
MALException(MALStandardError& error)
MALException(const MALException& other)
```

3.2.11.2.3 The MALException constructor parameters shall be assigned as described in table 3-28.

Table 3-28: MALException Constructor Parameters

Parameter	Description
message	Error message
error	The MALStandardError related to the cause of the MALException

3.2.11.2.4 The parameter 'message' shall be assigned with a message explaining why this MALException is raised.

3.2.11.2.5 If this MALException is causally linked to an exception, then the parameter 'error' shall be assigned with the linked exception.

3.2.11.2.6 The MALException constructor shall call the C++ std::runtime_error constructor having the same signature and pass the parameters.

3.2.12 MALINTERACTIONEXCEPTION

3.2.12.1 Definition

3.2.12.1.1 A MALInteractionException class shall be defined in order to raise a MAL standard error (defined in reference [1]) as a C++ std::exception.

3.2.12.1.2 The MALInteractionException class shall inherit from the class C++ std::exception.

3.2.12.2 Creation

3.2.12.2.1 A public MALInteractionException constructor shall be defined with a MALStandardError parameter.

3.2.12.2.2 The MALInteractionException constructor signature shall be:

```
MALInteractionException(const MALStandardError& standardError)
```

3.2.12.2.3 The MALInteractionException constructor parameter shall be assigned as described in table 3-29.

Table 3-29: MALInteractionException Constructor Parameter

Parameter	Description
standardError	Error to be raised

3.2.12.2.4 The MALInteractionException constructor shall call the C++ Exception constructor with the String representation of the MALStandardError 'extraInformation' field, or NULL if the 'extraInformation' is NULL.

3.2.12.3 Getter

A MALInteractionException getter method 'getStandardError' shall be defined in order to return the MALStandardError:

```
MALStandardError getStandardError()
```

3.2.13 MALELEMENTFACTORY

3.2.13.1 Definition

A MALElementFactory interface shall be defined in order to allow the creation of an element in a generic way, i.e., using the MAL::Element polymorphism.

3.2.13.2 Create an Element

3.2.13.2.1 A method 'createElement' shall be provided in order to instantiate an element.

3.2.13.2.2 The signature of the method 'createElement' shall be:

```
shared_ptr<Element> createElement()
```

3.2.14 MALELEMENTFACTORYREGISTRY

3.2.14.1 Definition

3.2.14.1.1 A MALElementFactoryRegistry class shall be defined in order to register MALElementFactory instances.

3.2.14.1.2 The MALElementFactoryRegistry class and its methods shall not be final in order that an extension class can be defined.

3.2.14.2 Register a MALElementFactory

3.2.14.2.1 A method ‘registerElementFactory’ shall be provided in order to register a MALElementFactory with the absolute short form of the created element.

3.2.14.2.2 The signature of the method ‘registerElementFactory’ shall be:

```
void registerElementFactory(
                  const long long& elementShortForm,
                  const shared_ptr<MALElementFactory>& elementFactory)
```

3.2.14.2.3 The parameters of the method ‘registerElementFactory’ shall be assigned as described in table 3-30.

Table 3-30: MALElementFactoryRegistry ‘registerElementFactory’ Parameters

Parameter	Description
elementShortForm	Absolute short form of the element created by the registered MALElementFactory
elementFactory	The registered MALElementFactory

3.2.14.3 Lookup a MALElementFactory

3.2.14.3.1 A method ‘lookupElementFactory’ shall be provided in order to get a MALElementFactory from the absolute short form of an element.

3.2.14.3.2 The signature of the method ‘lookupElementFactory’ shall be:

```
shared_ptr<MALElementFactory> lookupElementFactory(
            const long long& elementShortForm)
```

3.2.14.3.3 The parameter of the method ‘lookupElementFactory’ shall be assigned as described in table 3-31.

Table 3-31: MALElementFactoryRegistry ‘lookupElementFactory’ Parameter

Parameter	Description
elementShortForm	Absolute short form of an element

3.2.14.3.4 The method shall return NULL if the short form is not found.

3.2.14.4 Deregister a MALElementFactory

3.2.14.4.1 A method ‘deregisterElementFactory’ shall be provided in order to remove a registered MALElementFactory from this MALElementFactoryRegistry.

3.2.14.4.2 The signature of the method ‘deregisterElementFactory’ shall be:

```
bool deregisterElementFactory(const long long& elementShortForm)
```

3.2.14.4.3 The parameter of the method ‘deregisterElementFactory’ shall be assigned as described in table 3-32.

Table 3-32: MALElementFactoryRegistry ‘deregisterElementFactory’ Parameter

Parameter	Description
elementShortForm	Absolute short form of the registered MALElementFactory to remove

3.2.14.4.4 The method ‘deregisterElementFactory’ shall return TRUE if the MALElementFactory is found and removed.

3.2.14.4.5 The method ‘deregisterElementFactory’ shall return FALSE if the MALElementFactory is not found.

3.2.15 MALENCODER AND MALDECODER

3.2.15.1 Definition

3.2.15.1.1 The MALEncoder and MALDecoder interfaces shall be defined in order to provide encoding and decoding methods for the following data:

a) an <<Attribute>>;

b) a MAL::Element, which type is statically defined by the service;

c) a MAL::Attribute, which type is not statically defined by the service (Attribute polymorphism).

3.2.15.1.2 The MALEncoder and MALDecoder interfaces shall also provide methods for creating MALListEncoder and MALListDecoder.

3.2.15.1.3 The encoding methods shall be defined in the interface MALEncoder.

3.2.15.1.4 The decoding methods shall be defined in the interface MALDecoder.

3.2.15.1.5 Two methods shall be provided in order to encode elements that may be NULL or not.

3.2.15.1.6 Two methods shall be provided in order to decode elements that may be NULL or not.

3.2.15.1.7 The variables defined in table 3-33 shall be applied to the following code templates.

Table 3-33: MALEncoder and MALDecoder Variables

Variable name	Content
Attribute	Type name of the MAL attribute
C++ mapping type	C++ type used to represent the MAL attribute

3.2.15.2 Encode an <<Attribute>>

3.2.15.2.1 Two encoding methods shall be defined for each MAL attribute type.

3.2.15.2.2 The signatures of the encoding methods shall be:

```
public void encode<<Attribute>>(
            const shared_ptr<<<C++ mapping type>>>& attribute)

public void encodeNullable<<Attribute>>(
            const shared_ptr<<<C++ mapping type>>>& attribute)
```

3.2.15.2.3 The parameter of the encoding methods shall be assigned as described in table 3-34.

Table 3-34: MALEncoder 'encode[Nullable]<<Attribute>>' Parameter

Parameter	Description
attribute	Attribute to encode

3.2.15.2.4 If the method is 'encodeNullable<<Attribute>>', then the parameter 'attribute' may be NULL.

3.2.15.2.5 If an error occurs, then a MALException shall be raised.

3.2.15.3 Decode an <<Attribute>>

3.2.15.3.1 Two decoding methods shall be defined for each MAL attribute type.

3.2.15.3.2 The signatures of the decoding methods shall be:

```
public <<C++ mapping type>> decode<<Attribute>>() throws MALException

public <<C++ mapping type>> decodeNullable<<Attribute>>()
  throws MALException
```

3.2.15.3.3 If an error occurs, then a MALException shall be raised.

3.2.15.4 Encode an Element from a List

3.2.15.4.1 The elements from a list shall be indirectly encoded using a MALListEncoder.

3.2.15.4.2 A method ‘createListEncoder’ shall be defined for creating a MALListEncoder.

3.2.15.4.3 The List shall be passed as a parameter of the method ‘createListEncoder’ in order to enable the MALListEncoder to prepare the list encoding.

3.2.15.4.4 The signature of the method ‘createListEncoder’ shall be:

```
shared_ptr<MALListEncoder> createListEncoder(
                        const AbstractList<Element> *list)
```

3.2.15.4.5 The parameter of the method ‘createListEncoder’ shall be assigned as described in table 3-35.

Table 3-35: MALListEncoder ‘createListEncoder’ Parameter

Parameter	Description
list	List to be encoded

3.2.15.5 Decode an Element from a List

3.2.15.5.1 The elements from a list shall be indirectly decoded using a MALListDecoder.

3.2.15.5.2 A method ‘createListDecoder’ shall be defined for creating a MALListDecoder.

3.2.15.5.3 The List shall be passed as a parameter of the method ‘createListDecoder’ in order to enable the MALListDecoder to handle the list decoding.

3.2.15.5.4 The signature of the method ‘createListDecoder’ shall be:

```
shared_ptr<MALListDecoder> createListDecoder(
                        const AbstractList<Element> *list)
```

3.2.15.5.5 The parameter of the method ‘createListDecoder’ shall be assigned as described in table 3-36.

Table 3-36: MALListDecoder 'createListDecoder' Parameter

Parameter	Description
list	List to be decoded

3.2.15.6 Encode an Element

3.2.15.6.1 Two encoding methods shall be defined for the Element type.

3.2.15.6.2 The signatures of the encoding methods shall be:

```
void encodeElement(const shared_ptr<Element>& element)

void encodeNullableElement(const shared_ptr<Element>& element)
```

3.2.15.6.3 The parameter of the method 'encode[Nullable]Element' shall be assigned as described in table 3-37.

Table 3-37: MALEncoder 'encode[Nullable]Element' Parameter

Parameter	Description
element	Element to encode

3.2.15.6.4 If the method is 'encodeNullableElement', then the parameter 'element' may be NULL.

3.2.15.6.5 If an error occurs, then a MALException shall be raised.

3.2.15.7 Decode an Element

3.2.15.7.1 Two decoding methods shall be defined for the Element type.

3.2.15.7.2 The signatures of the decoding methods shall be:

```
shared_ptr<Element> decodeElement(const shared_ptr<Element>& element)

shared_ptr<Element> decodeNullableElement(
                  const shared_ptr<Element>& element)
```

3.2.15.7.3 The parameter of the method 'decode[Nullable]Element' shall be assigned as described in table 3-38.

Table 3-38: MALDecoder 'decode[Nullable]Element' Parameter

Parameter	Description
element	Element to decode

3.2.15.7.4 The returned element may be not the same instance as the parameter 'element'.

3.2.15.7.5 If an error occurs, then a MALException shall be raised.

3.2.15.8 Encode an Attribute

3.2.15.8.1 Two encoding methods shall be defined for the Attribute type.

3.2.15.8.2 The signatures of the encoding methods shall be:

```
void encodeAttribute(const shared_ptr<Attribute>& attribute)

void encodeNullableAttribute(const shared_ptr<Attribute>& attribute)
```

3.2.15.8.3 The parameter of the method 'encode[Nullable]Attribute' shall be assigned as described in table 3-39.

Table 3-39: MALEncoder 'encode[Nullable]Attribute' Parameter

Parameter	Description
attribute	Attribute to encode

3.2.15.8.4 If the method is 'encodeNullableAttribute', then the parameter 'attribute' may be NULL.

3.2.15.8.5 If an error occurs, then a MALException shall be raised.

3.2.15.9 Decode an Attribute

3.2.15.9.1 Two decoding methods shall be defined for the Attribute type.

3.2.15.9.2 The signatures of the decoding methods shall be:

```
shared_ptr<Attribute> decodeAttribute()
```

```
shared_ptr<Attribute> decodeNullableAttribute()
```

3.2.15.9.3 If an error occurs, then a MALException shall be raised.

3.2.16 MALLISTENCODER AND MALLISTDECODER

3.2.16.1 Definition

The MALListEncoder and MALListDecoder interfaces shall be defined in order to provide methods for encoding and decoding the elements of a list.

3.2.16.2 Encoding

3.2.16.2.1 The MALListEncoder interface shall extend the interface MALEncoder in order to provide the methods required for encoding the elements that belong to the list.

3.2.16.2.2 A method ‘close’ shall be defined in order to notify the MALListEncoder that the list is over.

3.2.16.2.3 The signature of the method ‘close’ shall be:

```
void close();
```

3.2.16.3 Decoding

3.2.16.3.1 The MALListDecoder interface shall extend the interface MALDecoder in order to provide the methods required for decoding the elements that belong to the list.

3.2.16.3.2 A method ‘hasNext’ shall be provided in order to check if the list has been entirely decoded:

a) the method ‘hasNext’ shall return TRUE if there is still at least one element to decode;

b) otherwise, it shall return FALSE: all the elements of the list have been decoded.

3.2.16.3.3 The signature of the method ‘hasNext’ shall be:

```
bool hasNext();
```

NOTE – A stream-based implementation of the method ‘hasNext’ may check that the number of elements in the list is less than the expected size, implying that the list size has been encoded in the stream ahead of the elements. As the List is passed as a parameter when calling the method ‘createListDecoder’ the MALListDecoder can compare the current size of the List and the size it should reach once all the elements have been decoded.

3.3 DATA STRUCTURES NAMESPACE

3.3.1 OVERVIEW

This subsection defines the classes and interfaces that are related to the MAL data types as defined in reference [1]. They belong to the C++ namespace:

```
mo::mal::structures
```

3.3.2 ELEMENT

3.3.2.1 Definition

3.3.2.1.1 An Element interface shall be defined in order to represent the MAL::Element type.

3.3.2.1.2 The Element interface shall define a pair of methods for encoding and decoding the element.

3.3.2.2 Short Form Query

3.3.2.2.1 A method ‘getShortForm’ shall be defined in order to return the absolute short form of the element type.

3.3.2.2.2 The signature of the method ‘getShortForm’ shall be:

```
long long getShortForm()
```

3.3.2.3 Area Number Query

3.3.2.3.1 A method ‘getAreaNumber’ shall be defined in order to return the number of the area this element type belongs to.

3.3.2.3.2 The signature of the method ‘getAreaNumber’ shall be:

```
uint16_t getAreaNumber()
```

3.3.2.4 Area Version Query

3.3.2.4.1 A method ‘getAreaVersion’ shall be defined in order to return the version of the area this element type belongs to.

3.3.2.4.2 The signature of the method ‘getAreaNumber’ shall be:

```
uint8_t getAreaVersion()
```

3.3.2.5 Service Number Query

3.3.2.5.1 A method ‘getServiceNumber’ shall be defined in order to return the number of the service to which this element type belongs.

3.3.2.5.2 The signature of the method ‘getServiceNumber’ shall be:

```
uint16_t getServiceNumber()
```

3.3.2.6 Type Short Form Query

3.3.2.6.1 A method ‘getTypeShortForm’ shall be defined in order to return the relative short form of the element type.

3.3.2.6.2 The signature of the method ‘getTypeShortForm’ shall be:

```
int32_t getTypeShortForm()
```

3.3.2.7 Create an Element

3.3.2.7.1 A method ‘createElement’ shall be defined in order to allow the creation of an element having the same type as this Element.

3.3.2.7.2 The signature of the method ‘createElement’ shall be:

```
shared_ptr<Element> createElement()
```

3.3.2.8 Encoding

3.3.2.8.1 A method ‘encode’ shall be defined in order to encode this Element.

3.3.2.8.2 The signature of the method ‘encode’ shall be:

```
void encode(MALEncoder& encoder)
```

3.3.2.8.3 The parameter of the method ‘encode’ shall be assigned as described in table 3-40.

Table 3-40: Element 'encode' Parameter

Parameter	Description
encoder	MALEncoder to be used in order to encode this Element

3.3.2.8.4 If an error occurs, then a MALException shall be raised.

3.3.2.9 Decoding

3.3.2.9.1 A method 'decode' shall be defined in order to decode an Element.

3.3.2.9.2 The signature of the method 'decode' shall be:

```
shared_ptr<Element> decode(MALDecoder& decoder) throws MALException
```

3.3.2.9.3 The parameter of the method 'decode' shall be assigned as described in table 3-41.

Table 3-41: Element 'decode' Parameter

Parameter	Description
decoder	MALDecoder to be used in order to decode an Element

3.3.2.9.4 If an error occurs, then a MALException shall be raised.

3.3.2.9.5 The method 'decode' shall not return the value NULL.

3.3.2.9.6 The returned Element may be not the same instance as this Element.

3.3.3 ATTRIBUTE

3.3.3.1 Definition

3.3.3.1.1 An Attribute interface shall be defined in order to represent the MAL::Attribute type.

NOTE – The mapping of the MAL::Attribute types is defined in 3.3.6.

3.3.3.1.2 The Attribute interface shall extend the Element interface.

NOTE – Interface refers to a class with only pure virtual methods.

3.3.3.2 Area Service Number

The Attribute interface shall declare the following constant:

```
static const long ABSOLUTE_AREA_SERVICE_NUMBER = 0x1000001000000L
```

3.3.3.3 Short Form Declaration

For each attribute type, the Attribute interface shall declare the following constants:

```
static const int _<<ATTRIBUTE>>_TYPE_SHORT_FORM =
  <<attribute short form>>

static const int <<ATTRIBUTE>>_TYPE_SHORT_FORM =
  _<<ATTRIBUTE>>_TYPE_SHORT_FORM

static const long <<ATTRIBUTE>>_SHORT_FORM =
  ABSOLUTE_AREA_SERVICE_NUMBER + _<<ATTRIBUTE>>_TYPE_SHORT_FORM
```

3.3.4 COMPOSITE

3.3.4.1.1 A Composite interface shall be defined in order to represent the MAL::Composite type.

3.3.4.1.2 The Composite interface shall be implemented by the classes representing the MAL::Composite data types.

3.3.4.1.3 The Composite interface shall extend the Element interface.

NOTE – This interface defines no method; it is a marker interface.

3.3.5 ENUMERATION

3.3.5.1 Definition

3.3.5.1.1 An Enumeration abstract class shall be defined in order to represent the MAL::Enumeration type.

3.3.5.1.2 The Enumeration class shall be extended by the classes representing a MAL enumeration type.

NOTE – The way to extend the Enumeration class is described in 4.5.6.

3.3.5.1.3 The Enumeration class shall implement the Element interface.

3.3.5.2 Creation

3.3.5.2.1 The Enumeration class shall define a protected constructor.

3.3.5.2.2 The Enumeration constructor shall take as a parameter the index of the enumerated item, i.e., its position in the enumeration declaration starting from zero.

3.3.5.2.3 The protected Enumeration constructor signature shall be:

```
Enumeration(int ordinal)
```

3.3.5.2.4 The Enumeration constructor parameter shall be assigned as described in table 3-42.

Table 3-42: Enumeration Constructor Parameter

Parameter	Description
ordinal	The index of the enumerated item

3.3.5.3 Get the Ordinal

3.3.5.3.1 The Enumeration class shall define a getter method ‘getOrdinal’ in order to return the index of the enumerated item.

3.3.5.3.2 The signature of the method ‘getOrdinal’ shall be:

```
int getOrdinal()
```

3.3.5.4 Get the Numeric Value

3.3.5.4.1 A method ‘getNumericValue’ shall be defined in order to return the numeric value of the enumerated item.

3.3.5.4.2 The abstract method signature shall be:

```
uint32_t getNumericValue()
```

3.3.5.5 Operator==

The C++ overloaded operator== shall be defined as follows:

a) the ‘operator==’ shall return TRUE if the parameter has the same final type as this Enumeration and has the same ordinal value;

b) otherwise it shall return FALSE.

3.3.6 ATTRIBUTES MAPPING

3.3.6.1.1 The MAL::Attribute types that are listed in table 3-43 shall be mapped to a C++ type.

Table 3-43: MAL::Attribute Types Mapped to a C++ Type

MAL::Attribute	C++ type
Boolean	bool
Float	float
Double	C++ double
Octet	int8_t
Short	int16_t
Integer	int
Long	int64_t
String	std::string

NOTES

1 C++ primitive types (e.g., ‘bool’, ‘float’, ‘int’) are not used because they cannot represent the value NULL.

2 As the C++ types listed above do not implement the Element interface, a Union class is provided in order to enable the assignment of one of the mapped MAL::Attribute to an Element variable.

3.3.6.1.2 The other MAL::Attribute types shall be mapped to an Attribute class:

a) Blob;

b) Duration;

c) Identifier;

d) Time;

e) FineTime;

f) UInteger;

g) ULong;

h) UOctet;

i) UShort;

j) URI.

3.3.7 UNION

3.3.7.1 Definition

3.3.7.1.1 A Union class shall be defined in order to assign a MAL::Attribute mapped to a non-Element C++ type (e.g., bool, string) to an Element variable.

3.3.7.1.2 The Union class shall implement the Attribute interface.

3.3.7.1.3 The variables defined in table 3-44 shall be applied to the Union code templates.

Table 3-44: Union Variables

Variable name	Content
Attribute type	Type name of a MAL::Attribute mapped to a C++ type without the 'MAL::' prefix
Attribute class	Class name of the C++ type used to map the MAL::Attribute
Short form	Absolute short form of the MAL::Attribute

3.3.7.2 Creation

3.3.7.2.1 The Union class shall provide a separate constructor for each MAL::Attribute type.

3.3.7.2.2 The Union constructor signature shall be:

```
Union(<<Attribute class>> value)
```

3.3.7.2.3 The Union constructor parameter shall be assigned as described in table 3-45.

Table 3-45: Union Constructor Parameter

Parameter	Description
value	Attribute value

3.3.7.3 Getters

3.3.7.3.1 The Union class shall provide a getter for each MAL::Attribute type.

3.3.7.3.2 The Union getter signature shall be:

```
<<Attribute class>> get<<Attribute type>>Value()
```

3.3.7.4 Get the Short Form

3.3.7.4.1 The Union class shall implement the method ‘getShortForm’ inherited from the Element interface.

3.3.7.4.2 The method ‘getShortForm’ shall return the absolute short form of the MAL::Attribute represented by this Union.

3.3.7.5 Get the Area Number

3.3.7.5.1 The Union class shall implement the method ‘getAreaNumber’ inherited from the Element interface.

3.3.7.5.2 The method ‘getAreaNumber’ shall return the MALHelper constant MAL_AREA_NUMBER.

3.3.7.6 Get the Service Number

3.3.7.6.1 The Union class shall implement the method ‘getServiceNumber’ inherited from the Element interface.

3.3.7.6.2 The method ‘getServiceNumber’ shall return the MALService constant NULL_SERVICE_NUMBER.

3.3.7.7 Get the Type Short Form

3.3.7.7.1 The Union class shall implement the method ‘getTypeShortForm’ inherited from the Element interface.

3.3.7.7.2 The method ‘getTypeShortForm’ shall return the relative short form of the MAL::Attribute represented by this Union.

3.3.7.8 Operator==

The C++ overloaded operator method ‘operator==’ shall be redefined as follows:

a) the method ‘operator==’ shall return TRUE if the type of the parameter is Union and if its value is equal to the value of this Union;

b) otherwise it shall return FALSE.

3.3.7.9 ToString

The C++ method ‘toString’ shall be defined by returning the string representation of the Union ‘value’.

3.3.7.10 Create an Element

The method ‘createElement’ defined by the interface Element shall be implemented by calling the Union constructor, declaring the attribute type, passing any value, and returning the result.

3.3.7.11 Encode

The method ‘encode’ defined by the Element interface shall be implemented by encoding the Union value with the method ‘encode<<Attribute>>’ provided by the MALEncoder.

3.3.7.12 Decode

The method ‘decode’ defined by the Element interface shall be implemented by decoding the Union value with the method ‘decode<<Attribute>>’ provided by the MALDecoder and returning a new Union.

3.3.8 BLOB

3.3.8.1 Definition

3.3.8.1.1 A Blob class shall be defined in order to represent a MAL Blob.

3.3.8.1.2 The Blob class shall implement the Attribute interface.

3.3.8.2 Short Form

The Blob class shall declare the constants and implement the methods as specified in 4.5.5.

3.3.8.3 Empty Constructor

3.3.8.3.1 The Blob class shall provide an empty constructor.

3.3.8.3.2 The Blob constructor signature shall be:

```
Blob()
```

3.3.8.4 8-bit Unsigned Integer Vector Constructor

3.3.8.4.1 The Blob class shall provide two constructors taking a vector of 8-bit unsigned integers as a parameter:

```
Blob(vector<uint8_t> blobValue)

Blob( vector<uint8_t>::iterator begin,
      vector<uint8_t>::iterator end )
```

3.3.8.4.2 The parameter of the Blob 8-bit unsigned integer vector constructor shall be assigned as described in table 3-46.

Table 3-46: Blob 8-bit Unsigned Integer Vector Constructor Parameters

Parameter	Description
blobValue	Vector of 8-bit unsigned integers to be wrapped in this Blob
begin	Vector iterator for start of the data of this Blob
end	Vector iterator for the end of the data of this Blob

3.3.8.4.3 The vector containing the data should not be modified after the Blob constructor has been called.

NOTE – If the vector containing the data is modified after the constructor has been called, then the Blob behaviour is unspecified.

3.3.8.5 URL Constructor

3.3.8.5.1 The Blob class shall provide a constructor taking a URL typed string as a parameter.

3.3.8.5.2 The Blob URL constructor signature shall be:

```
Blob(const string& url)
```

3.3.8.5.3 The parameter of the Blob URL constructor shall be assigned as described in table 3-47.

Table 3-47: Blob URL Constructor Parameter

Parameter	Description
url	URL which designated content shall be loaded in this Blob

3.3.8.5.4 The resource identified by the URL should not be modified until the method 'detach' is called.

NOTE – If the resource identified by the URL is modified after the constructor has been called, then the Blob behaviour is unspecified.

3.3.8.6 Check if URL Based

3.3.8.6.1 A method 'isURLBased' shall be defined in order to indicate whether the Blob contains a URL or not.

3.3.8.6.2 The signature of the method 'isURLBased' shall be:

```
bool isURLBased()
```

3.3.8.6.3 The method 'isURLBased' shall return TRUE if the Blob contains a URL; otherwise it shall return FALSE.

3.3.8.7 Get the vector<uint8_t> Value

3.3.8.7.1 A method 'getValue' shall be defined in order to return the value of this Blob as a byte array.

3.3.8.7.2 The signature of the method 'getValue' shall be:

```
vector<uint8_t> getValue()
```

3.3.8.7.3 If the Blob contains a URL, then the designated content shall be loaded and copied in the vector<uint8_t> returned by 'getValue'.

3.3.8.7.4 A MALException shall be thrown if an internal error occurs.

3.3.8.7.5 The byte array returned by 'getValue' should not be modified.

NOTE – If the returned byte array is modified, then the Blob behaviour is unspecified.

3.3.8.8 Get the URL

3.3.8.8.1 A method 'getURL' shall be defined in order to return the URL of this Blob.

3.3.8.8.2 The signature of the method ‘getURL’ shall be:

```
string getURL()
```

3.3.8.8.3 If the Blob does not contain a URL, then the method ‘getURL’ shall return NULL.

3.3.8.8.4 The resource identified by the URL should not be modified until the method ‘detach’ is called.

NOTE – If the resource identified by the URL is modified, then the Blob behaviour is unspecified.

3.3.8.9 Detach

3.3.8.9.1 A method ‘detach’ shall be defined in order not to delete the resource designated by the URL when this Blob is deleted.

3.3.8.9.2 The signature of the method ‘detach’ shall be:

```
void detach()
```

3.3.8.10 Get the Offset

3.3.8.10.1 A method ‘getOffset’ shall be defined in order to return the index in the array of the first byte of this Blob.

3.3.8.10.2 The signature of the method ‘getOffset’ shall be:

```
int getOffset()
```

3.3.8.11 Get the Length

3.3.8.11.1 A method ‘getLength’ shall be defined in order to return the number of bytes belonging to this Blob.

3.3.8.11.2 The signature of the method ‘getLength’ shall be:

```
int getLength()
```

3.3.8.12 Operator==

3.3.8.12.1 The C++ overloaded operator method ‘operator==’ shall be redefined as follows:

a) the method ‘operator==’ shall return TRUE if the following conditions are true:

   1) the type of the parameter is Blob, and

   2) the Blob parameter has the same byte array as this Blob;

b) otherwise it shall return FALSE.

3.3.8.12.2 If a Blob is based on a URL, then the binary content shall be loaded in order to make a deep comparison; i.e., the method shall check that the binary contents are the same.

3.3.8.13 Delete

3.3.8.13.1 A method ‘delete’ shall be defined in order to delete the resource designated by the URL.

3.3.8.13.2 The signature of the method ‘delete’ shall be:

```
void delete()
```

3.3.8.13.3 If the Blob is attached and if the URL protocol is ‘file’, then the resource designated by the URL shall be deleted.

3.3.8.13.4 A MALException shall be thrown if an internal error occurs.

3.3.8.14 Create an Element

The method ‘createElement’ defined by the Element interface shall be implemented by calling the Blob empty constructor and returning the result.

3.3.8.15 Encode

The method ‘encode’ defined by the Element interface shall be implemented by encoding this Blob with the method ‘encodeBlob’ provided by the MALEncoder.

3.3.8.16 Decode

The method ‘decode’ defined by the Element interface shall be implemented by decoding a Blob with the method ‘decodeBlob’ provided by the MALDecoder and returning the result.

3.3.9 ATTRIBUTE CLASSES

3.3.9.1 Overview

The MAL::Attribute types listed in table 3-48 shall be represented by an Attribute class wrapping a C++ type. The Time and FineTime types wrap the C++ structures MALTimeData and MALFineTimeData represented in tables 3-49 and 3-50. The seconds attribute represents the number of seconds since EPOCH which is defined as January 1, 1970 but does not account for leap seconds.

Table 3-48: MAL::Attribute Types Represented by a Specific Class

MAL::Attribute	Wrapped C++ type
Duration	double
Identifier	std::string
Time	MALTimeData
URI	std::string
FineTime	MALFineTimeData
UOctet	uint8_t
UShort	uint16_t
UInteger	uint32_t
ULong	uint64_t

Table 3-49: MAL::MALTimeData

Attribute	C++ type
seconds	int64_t
milliseconds	int64_t

Table 3-50: MAL::MALFineTimeData

Attribute	C++ type
seconds	int64_t
milliseconds	int64_t
microseconds	int64_t
nanoseconds	int64_t

3.3.9.2 Class Definition

3.3.9.2.1 The <<Attribute>> class shall be final.

3.3.9.2.2 The <<Attribute>> class shall implement the interfaces:

a) Attribute;

b) Comparable< <<Attribute>> >.

3.3.9.3 Short Form

The <<Attribute>> class shall declare the constants and implement the methods as specified in 4.5.5.

3.3.9.4 Empty Constructor

3.3.9.4.1 The <<Attribute>> class shall provide an empty constructor.

3.3.9.4.2 The <<Attribute>> class constructor signature shall be:

```
public <<Attribute>>()
```

3.3.9.4.3 The initial value of the MAL::Attribute shall be assigned as specified by table 3-51.

Table 3-51: Initial Value Assigned by the <<Attribute>> Empty Constructor

C++ type	Default value
double	0.0
int32_t	0
std::string	empty
int64_t	0
int16_t	0
uint32_t	0
uint8_t	0
uint16_t	0
uint64_t	0

3.3.9.5 Constructor

3.3.9.5.1 The <<Attribute>> class shall define a constructor taking the C++ type as a parameter.

3.3.9.5.2 The <<Attribute>> class constructor signature shall be:

```
<<Attribute>>(const <<C++ type>>& value)
```

3.3.9.6 Get the Value

3.3.9.6.1 The method ‘getValue’ shall return the <<Attribute>> value.

3.3.9.6.2 The signature of the method ‘getValue’ shall be:

```
<<C++ type>> getValue()
```

3.3.9.7 Operator==

The C++ overloaded ‘operator==’ method shall be defined as follows:

a) the overloaded ‘operator==’ shall return TRUE if the type of the parameter is <<Attribute>> and if its value is equal to the value of this <<Attribute>>;

b) otherwise it shall return FALSE.

3.3.9.8 ToString

The C++ method ‘toString’ shall be defined to return the string representation of the attribute ‘value’.

3.3.9.9 Create an Element

The method ‘createElement’ defined by the Element interface shall be implemented by calling the empty constructor and returning the result.

3.3.9.10 Encode

The method ‘encode’ defined by the Element interface shall be implemented by calling the method ‘encode<<Attribute>>’ provided by the MALEncoder.

3.3.9.11 Decode

The method ‘decode’ defined by the Element interface shall be implemented by calling the method ‘decode<<Attribute>>’ provided by the MALDecoder and returning the result.

3.3.10 ELEMENTLIST

3.3.10.1.1 An ElementList interface shall be defined in order to represent a list of MAL::Element.

3.3.10.1.2 The ElementList shall declare a generic type variable ‘T’.

NOTE – No type constraint can be defined because of MAL::Attribute types mapped to non-Element C++ classes.

3.3.10.1.3 The ElementList interface shall extend the Composite interface.

NOTE – This interface defines no method; it is a marker interface.

3.3.11 ATTRIBUTELIST

3.3.11.1.1 An AttributeList interface shall be defined in order to represent a list of MAL::Attribute.

3.3.11.1.2 The AttributeList shall declare a generic type variable ‘T’.

NOTE – No type constraint can be defined because of MAL::Attribute types mapped to non-Element C++ classes.

3.3.11.1.3 The AttributeList interface shall extend the ElementList<T> interface.

NOTE – This interface defines no method; it is a marker interface.

3.3.12 COMPOSITELIST

3.3.12.1.1 A CompositeList interface shall be defined in order to represent a list of MAL::Composite.

3.3.12.1.2 The CompositeList shall declare a generic type variable ‘T’.

3.3.12.1.3 The CompositeList interface shall inherit from the ElementList<T> interface and the C++ std::vector<T>.

NOTE – This interface defines no method; it is a marker interface.

3.4 CONSUMER NAMESPACE

3.4.1 OVERVIEW

This part of the API is dedicated to the MAL clients initiating interactions as service consumers. The consumer state diagrams are specified in reference [1].

The classes and interfaces belong to the C++ namespace:

```
mo::mal::consumer
```

3.4.2 MALCONSUMERMANAGER

3.4.2.1 Definition

3.4.2.1.1 A MALConsumerManager interface shall be defined in order to encapsulate the resources used to enable a MAL consumer to initiate interactions.

3.4.2.1.2 A MALConsumerManager shall be a MALConsumer factory.

3.4.2.2 MALConsumerManager Creation

3.4.2.2.1 A MALConsumerManager shall be created by calling the method ‘createConsumerManager’ provided by a MALContext.

3.4.2.2.2 Several consumers should be created in order to separate the resources used by the clients.

3.4.2.3 Create a Consumer

3.4.2.3.1 Two methods ‘createConsumer’ shall be defined in order to create a MALConsumer:

a) using a private MALEndpoint;

b) using a shared MALEndpoint.

3.4.2.3.2 The signatures of the methods ‘createConsumer’ shall be:

```
shared_ptr<MALConsumer> createConsumer(
      const string& localName,
      const URI& uriTo,
      const URI& brokerUri,
      const MALService& service,
      const shared_ptr<Blob>& authenticationId,
      const IdentifierList& domain,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
```

```
      const QoSLevel& qosLevel,
      const MALQoSProperties& qosProps,
      const uint32_t& priority)

shared_ptr<MALConsumer> createConsumer(
      const shared_ptr<MALEndpoint>& endpoint,
      const URI& uriTo,
      const URI& brokerUri,
      const MALService& service,
      const shared_ptr<Blob>& authenticationId,
      const IdentifierList& domain,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
      const QoSLevel& qosLevel,
      const MALQoSProperties& qosProps,
      const uint32_t& priority)
```

3.4.2.3.3 The parameters of the methods ‘createConsumer’ shall be assigned as described in table 3-52.

Table 3-52: MALConsumerManager ‘createConsumer’ Parameters

Parameter	Description
localName	Name of the private MALEndpoint to be created and used by the consumer
endpoint	Shared MALEndpoint to be used by the consumer
uriTo	URI of the service provider the consumer will interact with
brokerUri	URI of the broker used by the service provider to publish updates
service	Definition of the consumed service
authenticationId	Authentication identifier used by the consumer during all the interactions with the service provider
domain	Domain the service provider belongs to
networkZone	Network zone the provider belongs to
sessionType	Session type of the service
sessionName	Name of the session
qosLevel	QoS level required by the consumer for all the interactions with the provider
qosProps	QoS properties that are needed to configure the QoS level
priority	Message priority required by the consumer for all the interactions with the provider

3.4.2.3.4 The parameters ‘localName’, ‘uriTo’ and ‘brokerUri’ may be empty.

3.4.2.3.5 If the parameter ‘uriTo’ is empty, then the parameter ‘brokerUri’ shall not be empty.

3.4.2.3.6 If the parameter ‘brokerUri’ is empty, then the parameter ‘uriTo’ shall not be empty.

3.4.2.3.7 If the parameter ‘localName’ is not empty, then its value shall be unique for the MALContext instance and the protocol specified by the URI ‘uriTo’.

3.4.2.3.8 The protocol of the URI ‘brokerUri’ shall be the same as the protocol of the URI ‘uriTo’.

3.4.2.3.9 The parameter ‘qosProps’ may be empty.

3.4.2.3.10 The method ‘createConsumer’ shall not return the value NULL.

3.4.2.3.11 The creation of a MALConsumer, whatever the QoS level is, shall not raise a MALException if the service provider is not active.

3.4.2.3.12 If the MALConsumerManager is closed, then a MALException shall be raised.

3.4.2.3.13 If the consumer local name is not empty and if the consumer process starts again after a stop and creates the MALConsumer with the same local name (directly or through a shared MALEndpoint), then the MALConsumer shall recover the same URI as before the stop.

3.4.2.4 QoS Properties

The QoS properties given in table 3-53 shall be available.

Table 3-53: QoS Property

Property Name	Type	Description
timeToLive	int32_t	This is the time in milliseconds allowed by the consumer for delivering the messages to the provider. If the message cannot be delivered before, then it is dropped and the MAL standard error DELIVERY_TIMEDOUT is returned to the MALConsumer if the interaction pattern allows it.

3.4.2.5 Close

3.4.2.5.1 A method ‘close’ shall be defined in order to release the resources owned by a MALConsumerManager.

3.4.2.5.2 The signature of the public method ‘close’ shall be:

```
void close()
```

3.4.2.5.3 The method ‘close’ shall close all the MALConsumer instances that are owned by this MALConsumerManager.

3.4.2.5.4 The method ‘close’ shall return after all the MALConsumer instances have been closed.

3.4.2.5.5 If an internal error occurs, then a MALException shall be raised.

3.4.3 MALCONSUMER

3.4.3.1 Definition

3.4.3.1.1 A MALConsumer interface shall be defined in order to provide a communication context that initiates interaction patterns either in a synchronous or asynchronous way.

3.4.3.1.2 The resources used by each MALConsumer should be shared as long as it is possible for the implementation to do it.

3.4.3.1.3 The MAL message body elements shall be represented using the type ‘std::vector’.

3.4.3.1.4 The allowed MAL message body element types shall be:

a) MAL element types;

b) MALEncodedElement;

c) C++ types defined by a specific C++ mapping extension.

3.4.3.2 MALConsumer Creation

A MALConsumer shall be created by calling the method ‘createConsumer’ provided by a MALConsumerManager.

NOTE – A correct usage of the MALConsumer is to create one for each service provider a MAL consumer wants to interact with. Typically a MALConsumer is created for each service stub.

3.4.3.3 Get the URI

3.4.3.3.1 A getter method ‘getURI’ shall be defined in order to return the consumer URI.

3.4.3.3.2 The signature of the method ‘getURI’ shall be:

```
URI getURI()
```

3.4.3.4 SEND IP Initiation

3.4.3.4.1 Two methods ‘send’ shall be defined in order to initiate a SEND interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.4.2 The signatures of the method ‘send’ shall be:

```
shared_ptr<MALMessage> send(
                  const shared_ptr<MALSendOperation>& op,
                  const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> send(
                  const shared_ptr<MALSendOperation>& op,
                  const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.4.3 The parameters of the method ‘send’ shall be assigned as described in table 3-54.

Table 3-54: MALConsumer ‘send’ Parameters

Parameter	Description
op	SEND operation to initiate
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.4.4 The parameter ‘body’ may be empty vector.

3.4.3.4.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.4.6 The method ‘send’ shall return as soon as the initiation message has been sent.

3.4.3.4.7 The method ‘send’ shall return the MALMessage that has been sent.

3.4.3.4.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.4.9 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.4.3.4.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.5 Synchronous SUBMIT IP Initiation

3.4.3.5.1 Two methods ‘submit’ shall be defined in order to initiate a synchronous SUBMIT interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.5.2 The signatures of the method ‘submit’ shall be:

```
void submit(
            const shared_ptr<MALSubmitOperation>& op,
            const vector<shared_ptr<MALMessageBody>>& body)

void submit(
            const shared_ptr<MALSubmitOperation>& op,
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.5.3 The parameters of the method ‘submit’ shall be assigned as described in table 3-55.

Table 3-55: MALConsumer ‘submit’ Parameters

Parameter	Description
op	SUBMIT operation to initiate
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.5.4 The parameter ‘body’ may be empty vector.

3.4.3.5.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.5.6 The method ‘submit’ shall return as soon as the ACK message has been received.

3.4.3.5.7 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.5.8 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.4.3.5.9 If an ACK ERROR occurs,

a) a MALInteractionException shall be thrown;

b) the ACK ERROR message body shall be passed as the parameter ‘standardError’ of the MALException constructor.

3.4.3.5.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.6 Synchronous REQUEST IP Initiation

3.4.3.6.1 Two methods ‘request’ shall be defined in order to initiate a synchronous REQUEST interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.6.2 The signatures of the method ‘request’ shall be:

```
shared_ptr<MALMessageBody> request(
                  const shared_ptr<MALRequestOperation>& op,
                  const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessageBody> request(
                  const shared_ptr<MALRequestOperation>& op,
                  const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.6.3 The parameters of the method ‘request’ shall be assigned as described in table 3-56.

Table 3-56: MALConsumer ‘request’ Parameters

Parameter	Description
op	REQUEST operation to initiate
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.6.4 The parameter ‘body’ may be empty vector.

3.4.3.6.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.6.6 The method ‘request’ shall return as soon as the RESPONSE message has been received.

3.4.3.6.7 The RESPONSE message body shall be returned as the method ‘request’ result.

3.4.3.6.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.6.9 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.4.3.6.10 If a RESPONSE ERROR occurs,

a) a MALInteractionException shall be thrown;

b) the RESPONSE ERROR message body shall be passed as the parameter ‘standardError’ of the MALInteractionException constructor.

3.4.3.6.11 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.7 Synchronous INVOKE IP Initiation

3.4.3.7.1 Two methods ‘invoke’ shall be defined in order to initiate an INVOKE interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.7.2 The signatures of the method ‘invoke’ shall be:

```
shared_ptr<MALMessageBody> invoke(
                  const shared_ptr<MALInvokeOperation>& op,
                  const shared_ptr<MALInteractionListener>& listener,
                  const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessageBody> invoke(
                  const shared_ptr<MALInvokeOperation>& op,
                  const shared_ptr<MALInteractionListener>& listener,
                  const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.7.3 The parameters of the method ‘invoke’ shall be assigned as described in table 3-57.

Table 3-57: MALConsumer 'invoke' Parameters

Parameter	Description
op	INVOKE operation to initiate
listener	Listener in charge of receiving the messages RESPONSE and RESPONSE ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.7.4 The parameter 'body' may be NULL.

3.4.3.7.5 The parameter 'encodedBody' may be NULL.

3.4.3.7.6 The method 'invoke' shall return as soon as the ACK message has been received.

3.4.3.7.7 The ACK message body shall be returned as the method 'invoke' result.

3.4.3.7.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.7.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.7.10 If an ACK ERROR occurs,

a) a MALInteractionException shall be thrown;

b) the ACK ERROR message body shall be passed as the parameter 'standardError' of the MALInteractionException constructor.

3.4.3.7.11 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.7.12 The method 'invokeResponseReceived' provided by the parameter 'listener' shall be called as soon as the RESPONSE message has been delivered.

3.4.3.7.13 The method 'invokeResponseErrorReceived' provided by the parameter 'listener' shall be called if a RESPONSE ERROR occurs.

3.4.3.8 Synchronous PROGRESS IP Initiation

3.4.3.8.1 Two methods 'progress' shall be defined in order to initiate a PROGRESS interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.8.2 The signatures of the method ‘progress’ shall be:

```
shared_ptr<MALMessageBody> progress(
                  const shared_ptr<MALProgressOperation>& op,
                  const shared_ptr<MALInteractionListener>& listener,
                  const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessageBody> progress(
                  const shared_ptr<MALProgressOperation>& op,
                  const shared_ptr<MALInteractionListener>& listener,
                  const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.8.3 The parameters of the method ‘progress’ shall be assigned as described in table 3-58.

Table 3-58: MALConsumer ‘progress’ Parameters

Parameter	Description
op	PROGRESS operation to initiate
listener	Listener in charge of receiving the messages UPDATE, UPDATE ERROR, RESPONSE and RESPONSE ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.8.4 The parameter ‘body’ may be empty vector.

3.4.3.8.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.8.6 The method ‘progress’ shall return as soon as the ACK message has been received.

3.4.3.8.7 The ACK message body shall be returned as the method ‘progress’ result.

3.4.3.8.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.8.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.8.10 If an ACK ERROR occurs,

a) a MALInteractionException shall be thrown;

b) the ACK ERROR message body shall be passed as the parameter ‘standardError’ of the MALInteractionException constructor.

3.4.3.8.11 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.8.12 The method ‘progressUpdateReceived’ provided by the parameter ‘listener’ shall be called as soon as an UPDATE message has been delivered.

3.4.3.8.13 The method ‘progressUpdateErrorReceived’ provided by the parameter ‘listener’ shall be called if an UPDATE ERROR occurs.

3.4.3.8.14 The method ‘progressResponseReceived’ provided by the parameter ‘listener’ shall be called as soon as the RESPONSE message has been delivered.

3.4.3.8.15 The method ‘progressErrorReceived’ provided by the parameter ‘listener’ shall be called if a RESPONSE ERROR occurs.

3.4.3.9 Synchronous PUBLISH-SUBSCRIBE IP REGISTER Initiation

3.4.3.9.1 A method ‘register’ shall be defined in order to initiate a synchronous PUBLISH-SUBSCRIBE REGISTER interaction.

3.4.3.9.2 The signature of the method ‘syncRegister’ shall be:

```
void syncRegister(
            const shared_ptr<MALPubSubOperation>& op,
            const shared_ptr<Subscription>& subscription,
            const shared_ptr<MALInteractionListener>& listener)
```

3.4.3.9.3 The parameters of the method ‘syncRegister’ shall be assigned as described in table 3-59.

Table 3-59: MALConsumer ‘register’ Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation which REGISTER stage is to initiate.
subscription	Subscription to be registered.
listener	Listener in charge of receiving the messages NOTIFY and NOTIFY ERROR.

3.4.3.9.4 The method ‘register’ shall return as soon as the REGISTER ACK message has been received.

3.4.3.9.5 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.9.6 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.9.7 If a REGISTER ERROR occurs,

a) a MALInteractionException shall be thrown;

b) the REGISTER ERROR message body shall be passed as the parameter 'standardError' of the MALInteractionException constructor.

3.4.3.9.8 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.9.9 The method 'notifyReceived' provided by the parameter 'listener' shall be called as soon as a NOTIFY message has been delivered.

3.4.3.9.10 The method 'notifyErrorReceived' provided by the parameter 'listener' shall be called as soon as a NOTIFY ERROR message has been delivered.

3.4.3.9.11 If a call to 'syncRegister' is initiated while the consumer is already registered but with a different parameter 'listener', then only the second registered 'listener' shall receive the NOTIFY and NOTIFY ERROR messages.

3.4.3.10 Synchronous PUBLISH-SUBSCRIBE IP DEREGISTER Initiation

3.4.3.10.1 A method 'deregister' shall be defined in order to initiate a synchronous PUBLISH-SUBSCRIBE DEREGISTER interaction.

3.4.3.10.2 The signature of the method 'deregister' shall be:

```
void deregister(
            const shared_ptr<MALPubSubOperation>& op,
            const IdentifierList& subscriptionIdList)
```

3.4.3.10.3 The parameters of the method 'deregister' shall be assigned as described in table 3-60.

Table 3-60: MALConsumer 'deregister' Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation which DEREGISTER stage is to initiate
subscriptionIdList	List of the subscription identifiers to deregister

3.4.3.10.4 The method 'deregister' shall return as soon as the DEREGISTER ACK message has been received.

3.4.3.10.5 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.10.6 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.10.7 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.11 Asynchronous SUBMIT IP Initiation

3.4.3.11.1 Two methods ‘asyncSubmit’ shall be defined in order to initiate an asynchronous SUBMIT interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.11.2 The signatures of the method ‘asyncSubmit’ shall be:

```
shared_ptr<MALMessage> asyncSubmit(
            const shared_ptr<MALSubmitOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> asyncSubmit(
            const shared_ptr<MALSubmitOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.11.3 The parameters of the method ‘asyncSubmit’ shall be assigned as described in table 3-61.

Table 3-61: MALConsumer ‘asyncSubmit’ Parameters

Parameter	Description
op	SUBMIT operation to initiate
listener	Listener in charge of receiving the messages ACK and ACK ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.11.4 The parameter ‘body’ may be NULL.

3.4.3.11.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.11.6 The method ‘asyncSubmit’ shall return as soon as the initiation message has been sent.

3.4.3.11.7 The method ‘asyncSubmit’ shall return the MALMessage that has been sent.

3.4.3.11.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.11.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.11.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.11.11 The method ‘submitAckReceived’ provided by the parameter ‘listener’ shall be called as soon as the ACK message has been delivered.

3.4.3.11.12 The method ‘submitErrorReceived’ provided by the parameter ‘listener’ shall be called if an ACK ERROR occurs.

3.4.3.12 Asynchronous REQUEST IP Initiation

3.4.3.12.1 Two methods ‘asyncRequest’ shall be defined in order to initiate an asynchronous REQUEST interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.12.2 The signatures of the method ‘asyncRequest’ shall be:

```
shared_ptr<MALMessage> asyncRequest(
            const shared_ptr<MALRequestOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> asyncRequest(
            const shared_ptr<MALRequestOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.12.3 The parameters of the method ‘asyncRequest’ shall be assigned as described in table 3-62.

Table 3-62: MALConsumer 'asyncRequest' Parameters

Parameter	Description
op	REQUEST operation to initiate
listener	Listener in charge of receiving the messages RESPONSE and RESPONSE ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.12.4 The parameter 'body' may be NULL.

3.4.3.12.5 The parameter 'encodedBody' may be NULL.

3.4.3.12.6 The method 'asyncRequest' shall return as soon as the initiation message has been sent.

3.4.3.12.7 The method 'asyncRequest' shall return the MALMessage that has been sent.

3.4.3.12.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.12.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.12.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.12.11 The method 'requestResponseReceived' provided by the parameter 'listener' shall be called as soon as the RESPONSE message has been delivered.

3.4.3.12.12 The method 'requestErrorReceived' provided by the parameter 'listener' shall be called if an RESPONSE ERROR occurs.

3.4.3.13 Asynchronous INVOKE IP Initiation

3.4.3.13.1 Two methods 'asyncInvoke' shall be defined in order to initiate an asynchronous INVOKE interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.13.2 The signatures of the method 'asyncInvoke' shall be:

```
shared_ptr<MALMessage> asyncInvoke(
            const shared_ptr<MALInvokeOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
```

```
            const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> asyncInvoke(
            const shared_ptr<MALInvokeOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.13.3 The parameters of the method ‘asyncInvoke’ shall be assigned as described in table 3-63.

Table 3-63: MALConsumer ‘asyncInvoke’ Parameters

Parameter	Description
op	INVOKE operation to initiate
listener	Listener in charge of receiving the messages ACK, ACK ERROR, RESPONSE and RESPONSE ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.13.4 The parameter ‘body’ may be NULL.

3.4.3.13.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.13.6 The method ‘asyncInvoke’ shall return as soon as the initiation message has been sent.

3.4.3.13.7 The method ‘asyncInvoke’ shall return the MALMessage that has been sent.

3.4.3.13.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.13.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.13.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.13.11 The method ‘invokeAckReceived’ provided by the parameter ‘listener’ shall be called as soon as the ACK message has been delivered.

3.4.3.13.12 The method ‘invokeAckErrorReceived’ provided by the parameter ‘listener’ shall be called if an ACK ERROR occurs.

3.4.3.13.13 The method ‘invokeResponseReceived’ provided by the parameter ‘listener’ shall be called as soon as the RESPONSE message has been delivered.

3.4.3.13.14 The method ‘invokeResponseErrorReceived’ provided by the parameter ‘listener’ shall be called if a RESPONSE ERROR occurs.

3.4.3.14 Asynchronous PROGRESS IP Initiation

3.4.3.14.1 Two methods ‘asyncProgress’ shall be defined in order to initiate an asynchronous PROGRESS interaction:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.4.3.14.2 The signatures of the method ‘asyncProgress’ shall be:

```
shared_ptr<MALMessage> asyncProgress(
            const shared_ptr<MALProgressOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> asyncProgress(
            const shared_ptr<MALProgressOperation>& op,
            const shared_ptr<MALInteractionListener> listener&,
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.4.3.14.3 The parameters of the method ‘asyncProgress’ shall be assigned as described in table 3-64.

Table 3-64: MALConsumer ‘asyncProgress’ Parameters

Parameter	Description
op	PROGRESS operation to initiate
listener	Listener in charge of receiving the messages ACK, ACK ERROR, UPDATE, UPDATE ERROR, RESPONSE and RESPONSE ERROR
body	Body elements to transmit in the initiation message
encodedBody	Encoded body to transmit in the initiation message

3.4.3.14.4 The parameter ‘body’ may be NULL.

3.4.3.14.5 The parameter ‘encodedBody’ may be NULL.

3.4.3.14.6 The method ‘asyncProgress’ shall return as soon as the initiation message has been sent.

3.4.3.14.7 The method ‘asyncProgress’ shall return the MALMessage that has been sent.

3.4.3.14.8 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.14.9 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.14.10 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.14.11 The method ‘progressAckReceived’ provided by the parameter ‘listener’ shall be called as soon as the ACK message has been delivered.

3.4.3.14.12 The method ‘progressAckErrorReceived’ provided by the parameter ‘listener’ shall be called if an ACK ERROR occurs.

3.4.3.14.13 The method ‘progressUpdateReceived’ provided by the parameter ‘listener’ shall be called as soon as an UPDATE message has been delivered.

3.4.3.14.14 The method ‘progressUpdateErrorReceived’ provided by the parameter ‘listener’ shall be called if an UPDATE ERROR occurs.

3.4.3.14.15 The method ‘progressResponseReceived’ provided by the parameter ‘listener’ shall be called as soon as the RESPONSE message has been delivered.

3.4.3.14.16 The method ‘progressResponseErrorReceived’ provided by the parameter ‘listener’ shall be called if a RESPONSE ERROR occurs.

3.4.3.15 Asynchronous PUBLISH-SUBSCRIBE IP REGISTER Initiation

3.4.3.15.1 A method ‘asyncRegister’ shall be defined in order to initiate an asynchronous PUBLISH-SUBSCRIBE REGISTER interaction.

3.4.3.15.2 The signature of the method ‘asyncRegister’ shall be:

```
shared_ptr<MALMessage> asyncRegister(
                  const shared_ptr<MALPubSubOperation>& op,
                  const shared_ptr<Subscription>& subscription,
                  const shared_ptr<MALInteractionListener>& listener)
```

3.4.3.15.3 The parameters of the method ‘asyncRegister’ shall be assigned as described in table 3-65.

Table 3-65: MALConsumer 'asyncRegister' Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation which REGISTER stage is to initiate
subscription	Subscription to be registered
listener	Listener in charge of receiving the messages REGISTER ACK, REGISTER ERROR, NOTIFY and NOTIFY ERROR

3.4.3.15.4 The method 'asyncRegister' shall return as soon as the initiation message has been sent.

3.4.3.15.5 The method 'asyncRegister' shall return the MALMessage that has been sent.

3.4.3.15.6 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.15.7 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.15.8 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.15.9 The method 'registerAckReceived' provided by the parameter 'listener' shall be called as soon as the REGISTER ACK message has been delivered.

3.4.3.15.10 The method 'registerAckErrorReceived' provided by the parameter 'listener' shall be called if an REGISTER ERROR occurs.

3.4.3.15.11 The method 'notifyReceived' provided by the parameter 'listener' shall be called as soon as a NOTIFY message has been delivered.

3.4.3.15.12 The method 'notifyErrorReceived' provided by the parameter 'listener' shall be called as soon as a NOTIFY ERROR message has been delivered.

3.4.3.15.13 If a call to 'asyncRegister' is initiated while the consumer is already registered but with a different parameter 'listener', then only the second registered 'listener' shall receive the NOTIFY and NOTIFY ERROR messages.

3.4.3.16 Asynchronous PUBLISH-SUBSCRIBE IP DEREGISTER Initiation

3.4.3.16.1 A method 'asyncDeregister' shall be defined in order to initiate an asynchronous PUBLISH-SUBSCRIBE DEREGISTER interaction.

3.4.3.16.2 The signature of the method 'asyncDeregister' shall be:

```
shared_ptr<MALMessage> asyncDeregister(
```

```
const shared_ptr<MALPubSubOperation>& op,
const IdentifierList& subscriptionIdList,
const shared_ptr<MALInteractionListener>& listener)
```

3.4.3.16.3 The parameters of the method ‘asyncDeregister’ shall be assigned as described in table 3-66.

Table 3-66: MALConsumer ‘asyncDeregister’ Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation which DEREGISTER stage is to initiate
subscriptionIdList	List of the subscription identifiers to deregister
listener	Listener in charge of receiving the messages DEREGISTER ACK, DEREGISTER ERROR

3.4.3.16.4 The method ‘asyncDeregister’ shall return as soon as the initiation message has been sent.

3.4.3.16.5 The method ‘asyncDeregister’ shall return the MALMessage that has been sent.

3.4.3.16.6 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.4.3.16.7 A MALException shall be thrown if an error occurs during the initiation message sending.

3.4.3.16.8 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.16.9 The method ‘deregisterAckReceived’ provided by the parameter ‘listener’ shall be called as soon as the DEREGISTER ACK message has been delivered.

3.4.3.17 Continue an Interaction

3.4.3.17.1 A method ‘continueInteraction’ shall be defined in order to continue an interaction that has been interrupted.

3.4.3.17.2 The signature of the method ‘continueInteraction’ shall be:

```
void continueInteraction(
  shared_ptr<MALOperation>& op,
  const uint8_t lastInteractionStage,
  const Time initiationTimestamp,
  const int64_t transactionId,
  shared_ptr<MALInteractionListener>& listener)
```

3.4.3.17.3 The parameters of the method ‘continueInteraction’ shall be assigned as described in table 3-67.

Table 3-67: MALConsumer ‘continueInteraction’ Parameters

Parameter	Description
op	The operation to continue
lastInteractionStage	The last stage of the interaction to continue
initiationTimestamp	Timestamp of the interaction initiation message
transactionId	Transaction identifier of the interaction to continue
listener	Listener in charge of receiving the messages from the service provider

3.4.3.17.4 The method ‘continueInteraction’ should be called before the endpoint message delivery is started otherwise some messages may be lost.

3.4.3.17.5 A MALInteractionException shall be thrown if a MAL standard error occurs.

3.4.3.17.6 A MALException shall be thrown if a non-MAL error occurs.

3.4.3.17.7 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.18 Set the Transmit Error Listener

3.4.3.18.1 A method ‘setTransmitErrorListener’ shall be defined in order to set a MALTransmitErrorListener.

3.4.3.18.2 The signature of the method ‘setTransmitErrorListener’ shall be:

```
void setTransmitErrorListener(
                  const shared_ptr<MALTransmitErrorListener>& listener)
```

3.4.3.18.3 The parameter of the method ‘setTransmitErrorListener’ shall be assigned as described in table 3-68.

Table 3-68: MALConsumer ‘setTransmitErrorListener’ Parameter

Parameter	Description
listener	Listener in charge of receiving every asynchronous TRANSMIT ERROR that cannot be returned as a MAL message

3.4.3.18.4 The parameter ‘listener’ may be NULL.

3.4.3.18.5 The MALTransmitErrorListener shall be called when:

a) a TRANSMIT ERROR is asynchronously returned to the consumer;

b) the TRANSMIT ERROR cannot be returned as a MAL message.

3.4.3.18.6 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.18.7 If an internal error occurs, then a MALException shall be raised.

3.4.3.19 Get the Transmit Error Listener

3.4.3.19.1 A method ‘getTransmitErrorListener’ shall be defined in order to return the MALTransmitErrorListener.

3.4.3.19.2 The signature of the method ‘getTransmitErrorListener’ shall be:

```
shared_ptr<MALTransmitErrorListener> getTransmitErrorListener()
```

3.4.3.19.3 If no MALTransmitErrorListener has been set, then the method ‘getTransmitErrorListener’ shall return NULL.

3.4.3.19.4 If the MALConsumer is closed, then a MALException shall be raised.

3.4.3.19.5 If an internal error occurs, then a MALException shall be raised.

3.4.3.20 Close

3.4.3.20.1 A method ‘close’ shall be defined in order to terminate all pending interactions initiated by the MALConsumer.

3.4.3.20.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.4.3.20.3 The synchronous IP initiation methods shall return with a response or a raised MALException depending on whether or not there was a message available at the time of the close.

3.4.3.20.4 All the MALInteractionListeners that were running at the time of the close shall have returned before the method ‘close’ returns.

3.4.3.20.5 If the consumer owns a private MALEndpoint, then the method ‘close’ provided by the MALEndpoint shall be called.

3.4.3.20.6 If an internal error occurs, then a MALException shall be raised.

3.4.3.20.7 If a provider or a broker try to interact with a closed consumer and if the QoS level is QUEUED, then the message shall be queued and delivered as soon as the consumer message delivery is started again.

3.4.4 MALINTERACTIONLISTENER

3.4.4.1 Definition

A MALInteractionListener interface shall be defined in order to enable a consumer to asynchronously receive a message.

3.4.4.2 Receive a SUBMIT ACK Message

3.4.4.2.1 A method ‘submitAckReceived’ shall be defined in order to receive the ACK message defined by the IP SUBMIT.

3.4.4.2.2 The signature of the method ‘submitAckReceived’ shall be:

```
void submitAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const MALQoSProperties& qosProperties)
```

3.4.4.2.3 The parameters of the method ‘submitAckReceived’ shall be assigned as described in table 3-69.

Table 3-69: MALInteractionListener ‘submitAckReceived’ Parameters

Parameter	Description
header	Header of the ACK message
qosProperties	QoS properties of the ACK message

3.4.4.2.4 The parameter ‘header’ shall not be NULL.

3.4.4.2.5 The parameter ‘qosProperties’ may be NULL.

3.4.4.2.6 If an error occurs, then a MALException may be raised.

3.4.4.3 Receive a SUBMIT ERROR Message

3.4.4.3.1 A method ‘submitErrorReceived’ shall be defined in order to receive the ERROR message defined by the IP SUBMIT.

3.4.4.3.2 The signature of the method ‘submitErrorReceived’ shall be:

```
void submitErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.3.3 The parameters of the method ‘submitErrorReceived’ shall be assigned as described in table 3-70.

Table 3-70: MALInteractionListener ‘submitErrorReceived’ Parameters

Parameter	Description
header	Header of the ERROR message
body	Body of the ERROR message
qosProperties	QoS properties of the ERROR message

3.4.4.3.4 The parameters ‘header’ and ‘body’ shall not be NULL.

3.4.4.3.5 The parameter ‘qosProperties’ may be NULL.

3.4.4.3.6 If an error occurs, then a MALException may be raised.

3.4.4.4 Receive a REQUEST RESPONSE Message

3.4.4.4.1 A method ‘requestResponseReceived’ shall be defined in order to receive the RESPONSE message defined by the IP REQUEST.

3.4.4.4.2 The signature of the method ‘requestResponseReceived’ shall be:

```
void requestResponseReceived(
                        const shared_ptr<MALMessageHeader>& header,
                        const shared_ptr<MALMessageBody>& body,
                        const MALQoSProperties& qosProperties)
```

3.4.4.4.3 The parameters of the method ‘requestResponseReceived’ shall be assigned as described in table 3-71.

Table 3-71: MALInteractionListener 'requestResponseReceived' Parameters

Parameter	Description
header	Header of the RESPONSE message
body	Body of the RESPONSE message
qosProperties	QoS properties of the RESPONSE message

3.4.4.4.4 The parameter 'header' shall not be NULL.

3.4.4.4.5 The parameters 'body' and 'qosProperties' may be NULL.

3.4.4.4.6 If the RESPONSE message is empty, then the parameter 'body' shall be NULL.

3.4.4.5 Receive a REQUEST ERROR Message

3.4.4.5.1 A method 'requestErrorReceived' shall be defined in order to receive the ERROR message defined by the IP REQUEST.

3.4.4.5.2 The signature of the method 'requestErrorReceived' shall be:

```
void requestErrorReceived(
                        const shared_ptr<MALMessageHeader>& header,
                        const shared_ptr<MALErrorBody>& body,
                        const MALQoSProperties& qosProperties)
```

3.4.4.5.3 The parameters of the method 'requestErrorReceived' shall be assigned as described in table 3-72.

Table 3-72: MALInteractionListener 'requestErrorReceived' Parameters

Parameter	Description
header	Header of the ERROR message
body	Body of the ERROR message
qosProperties	QoS properties of the ERROR message

3.4.4.5.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.5.5 The parameter 'qosProperties' may be NULL.

3.4.4.5.6 If an error occurs, then a MALException may be raised.

3.4.4.6 Receive an INVOKE ACK Message

3.4.4.6.1 A method ‘invokeAckReceived’ shall be defined in order to receive the ACK message defined by the IP INVOKE.

3.4.4.6.2 The signature of the method ‘invokeAckReceived’ shall be:

```
void invokeAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALMessageBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.6.3 The parameters of the method ‘invokeAckReceived’ shall be assigned as described in table 3-73.

Table 3-73: MALInteractionListener ‘invokeAckReceived’ Parameters

Parameter	Description
header	Header of the ACK message
body	Body of the ACK message
qosProperties	QoS properties of the ACK message

3.4.4.6.4 The parameter ‘header’ shall not be NULL.

3.4.4.6.5 The parameters ‘body’ and ‘qosProperties’ may be NULL.

3.4.4.6.6 If the ACK message is empty, then the parameter ‘body’ shall be NULL.

3.4.4.6.7 If an error occurs, then a MALException may be raised.

3.4.4.7 Receive an INVOKE ACK ERROR Message

3.4.4.7.1 A method ‘invokeAckErrorReceived’ shall be defined in order to receive the ACK ERROR message defined by the IP INVOKE.

3.4.4.7.2 The signature of the method ‘invokeAckErrorReceived’ shall be:

```
void invokeAckErrorReceived(
                       const shared_ptr<MALMessageHeader>& header,
                       const shared_ptr<MALErrorBody>& body,
                       const MALQoSProperties& qosProperties)
```

3.4.4.7.3 The parameters of the method ‘invokeAckErrorReceived’ shall be assigned as described in table 3-74.

Table 3-74: MALInteractionListener 'invokeAckErrorReceived' Parameters

Parameter	Description
header	Header of the ACK ERROR message
body	Body of the ACK ERROR message
qosProperties	QoS properties of the ACK ERROR message

3.4.4.7.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.7.5 The parameter 'qosProperties' may be NULL.

3.4.4.7.6 If an error occurs, then a MALException may be raised.

3.4.4.8 Receive an INVOKE RESPONSE Message

3.4.4.8.1 A method 'invokeResponseReceived' shall be defined in order to receive the RESPONSE message defined by the IP INVOKE.

3.4.4.8.2 The signature of the method 'invokeResponseReceived' shall be:

```
void invokeResponseReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALMessageBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.8.3 The parameters of the method 'invokeResponseReceived' shall be assigned as described in table 3-75.

Table 3-75: MALInteractionListener 'invokeResponseReceived' Parameters

Parameter	Description
header	Header of the RESPONSE message
body	Body of the RESPONSE message
qosProperties	QoS properties of the RESPONSE message

3.4.4.8.4 The parameter 'header' shall not be NULL.

3.4.4.8.5 The parameters 'body' and 'qosProperties' may be NULL.

3.4.4.8.6 If the RESPONSE message is empty, then the parameter 'body' shall be NULL.

3.4.4.8.7 If an error occurs, then a MALException may be raised.

3.4.4.9 Receive an INVOKE RESPONSE ERROR Message

3.4.4.9.1 A method ‘invokeResponseErrorReceived’ shall be defined in order to receive the RESPONSE ERROR message defined by the IP INVOKE.

3.4.4.9.2 The signature of the method ‘invokeResponseErrorReceived’ shall be:

```
void invokeResponseErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.9.3 The parameters of the method ‘invokeResponseErrorReceived’ shall be assigned as described in table 3-76.

Table 3-76: MALInteractionListener ‘invokeResponseErrorReceived’ Parameters

Parameter	Description
header	Header of the RESPONSE ERROR message
body	Body of the RESPONSE ERROR message
qosProperties	QoS properties of the RESPONSE ERROR message

3.4.4.9.4 The parameters ‘header’ and ‘body’ shall not be NULL.

3.4.4.9.5 The parameter ‘qosProperties’ may be NULL.

3.4.4.9.6 If an error occurs, then a MALException may be raised.

3.4.4.10 Receive a PROGRESS ACK Message

3.4.4.10.1 A method ‘progressAckReceived’ shall be defined in order to receive the ACK message defined by the IP PROGRESS.

3.4.4.10.2 The signature of the method ‘progressAckReceived’ shall be:

```
void progressAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALMessageBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.10.3 The parameters of the method ‘progressAckReceived’ shall be assigned as described in table 3-77.

Table 3-77: MALInteractionListener 'progressAckReceived' Parameters

Parameter	Description
header	Header of the ACK message
body	Body of the ACK message
qosProperties	QoS properties of the ACK message

3.4.4.10.4 The parameter 'header' shall not be NULL.

3.4.4.10.5 The parameters 'body' and 'qosProperties' may be NULL.

3.4.4.10.6 If the ACK message is empty, then the parameter 'body' shall be NULL.

3.4.4.10.7 If an error occurs, then a MALException may be raised.

3.4.4.11 Receive a PROGRESS ACK ERROR Message

3.4.4.11.1 A method 'progressAckErrorReceived' shall be defined in order to receive the ACK ERROR message defined by the IP PROGRESS.

3.4.4.11.2 The signature of the method 'progressAckErrorReceived' shall be:

```
void progressAckErrorReceived(
                        const shared_ptr<MALMessageHeader>& header,
                        const shared_ptr<MALErrorBody>& body,
                        const MALQoSProperties& qosProperties)
```

3.4.4.11.3 The parameters of the method 'progressAckErrorReceived' shall be assigned as described in table 3-78.

Table 3-78: MALInteractionListener 'progressAckErrorReceived' Parameters

Parameter	Description
header	Header of the ACK ERROR message
body	Body of the ACK ERROR message
qosProperties	QoS properties of the ACK ERROR message

3.4.4.11.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.11.5 The parameter 'qosProperties' may be NULL.

3.4.4.11.6 If an error occurs, then a MALException may be raised.

3.4.4.12 Receive a PROGRESS UPDATE Message

3.4.4.12.1 A method ‘updateReceived’ shall be defined in order to receive the UPDATE message defined by the IP PROGRESS.

3.4.4.12.2 The signature of the method ‘progressUpdateReceived’ shall be:

```
void progressUpdateReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALMessageBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.12.3 The parameters of the method ‘progressUpdateReceived’ shall be assigned as described in table 3-79.

Table 3-79: MALInteractionListener ‘progressUpdateReceived’ Parameters

Parameter	Description
header	Header of the UPDATE message
body	Body of the UPDATE message
qosProperties	QoS properties of the UPDATE message

3.4.4.12.4 The parameter ‘header’ shall not be NULL.

3.4.4.12.5 The parameters ‘body’ and ‘qosProperties’ may be NULL.

3.4.4.12.6 If the UDPATE message is empty, then the parameter ‘body’ shall be NULL.

3.4.4.12.7 If an error occurs, then a MALException may be raised.

3.4.4.13 Receive a PROGRESS UPDATE ERROR Message

3.4.4.13.1 A method ‘progressUpdateErrorReceived’ shall be defined in order to receive the UPDATE ERROR message defined by the IP PROGRESS.

3.4.4.13.2 The signature of the method ‘progressUpdateErrorReceived’ shall be:

```
void progressUpdateErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.13.3 The parameters of the method ‘progressUpdateErrorReceived’ shall be assigned as described in table 3-80.

Table 3-80: MALInteractionListener 'progressUpdateErrorReceived' Parameters

Parameter	Description
header	Header of the UPDATE ERROR message
body	Body of the UPDATE ERROR message
qosProperties	QoS properties of the UPDATE ERROR message

3.4.4.13.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.13.5 The parameter 'qosProperties' may be NULL.

3.4.4.13.6 If an error occurs, then a MALException may be raised.

3.4.4.14 Receive a PROGRESS RESPONSE Message

3.4.4.14.1 A method 'progressResponseReceived' shall be defined in order to receive the RESPONSE message defined by the IP PROGRESS.

3.4.4.14.2 The signature of the method 'progressResponseReceived' shall be:

```
void progressResponseReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALMessageBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.14.3 The parameters of the method 'progressResponseReceived' shall be assigned as described in table 3-81.

Table 3-81: MALInteractionListener 'progressResponseReceived' Parameters

Parameter	Description
header	Header of the RESPONSE message
body	Body of the RESPONSE message
qosProperties	QoS properties of the RESPONSE message

3.4.4.14.4 The parameter 'header' shall not be NULL.

3.4.4.14.5 The parameters 'body' and 'qosProperties' may be NULL.

3.4.4.14.6 If the RESPONSE message is empty, then the parameter 'body' shall be NULL.

3.4.4.14.7 If an error occurs, then a MALException may be raised.

3.4.4.15 Receive a PROGRESS RESPONSE ERROR Message

3.4.4.15.1 A method ‘progressResponseErrorReceived’ shall be defined in order to receive the RESPONSE ERROR message defined by the IP PROGRESS.

3.4.4.15.2 The signature of the method ‘progressResponseErrorReceived’ shall be:

```
void progressResponseErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.15.3 The parameters of the method ‘progressResponseErrorReceived’ shall be assigned as described in table 3-82.

Table 3-82: MALInteractionListener ‘progressResponseErrorReceived’ Parameters

Parameter	Description
header	Header of the RESPONSE ERROR message
body	Body of the RESPONSE ERROR message
qosProperties	QoS properties of the RESPONSE ERROR message

3.4.4.15.4 The parameters ‘header’ and ‘body’ shall not be NULL.

3.4.4.15.5 The parameter ‘qosProperties’ may be NULL.

3.4.4.15.6 If an error occurs, then a MALException may be raised.

3.4.4.16 Receive a REGISTER ACK Message

3.4.4.16.1 A method ‘registerAckReceived’ shall be defined in order to receive the REGISTER ACK message defined by the IP PUBLISH-SUBSCRIBE.

3.4.4.16.2 The signature of the method ‘registerAckReceived’ shall be:

```
void registerAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const MALQoSProperties& qosProperties
```

3.4.4.16.3 The parameters of the method ‘registerAckReceived’ shall be assigned as described in table 3-83.

Table 3-83: MALInteractionListener 'registerAckReceived' Parameters

Parameter	Description
header	Header of the REGISTER ACK message
qosProperties	QoS properties of the REGISTER ACK message

3.4.4.16.4 The parameter 'header' shall not be NULL.

3.4.4.16.5 The parameter 'qosProperties' may be NULL.

3.4.4.16.6 If an error occurs, then a MALException may be raised.

3.4.4.17 Receive a REGISTER ERROR Message

3.4.4.17.1 A method 'registerErrorReceived' shall be defined in order to receive the REGISTER ERROR message defined by the IP PUBLISH-SUBSCRIBE.

3.4.4.17.2 The signature of the method 'registerErrorReceived' shall be:

```
void registerErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.17.3 The parameters of the method 'registerErrorReceived' shall be assigned as described in table 3-84.

Table 3-84: MALInteractionListener 'registerErrorReceived' Parameters

Parameter	Description
header	Header of the REGISTER ERROR message
body	Body of the REGISTER ERROR message
qosProperties	QoS properties of the REGISTER ERROR message

3.4.4.17.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.17.5 The parameter 'qosProperties' may be NULL.

3.4.4.17.6 If an error occurs, then a MALException may be raised.

3.4.4.18 Receive a NOTIFY Message

3.4.4.18.1 A method ‘notifyReceived’ shall be defined in order to receive the NOTIFY message defined by the IP PUBLISH-SUBSCRIBE.

3.4.4.18.2 The signature of the method ‘notifyReceived’ shall be:

```
void notifyReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALNotifyBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.18.3 The parameters of the method ‘notifyReceived’ shall be assigned as described in table 3-85.

Table 3-85: MALInteractionListener ‘notifyReceived’ Parameters

Parameter	Description
header	Header of the NOTIFY message
body	Body of the NOTIFY message
qosProperties	QoS properties of the REGISTER ERROR message

3.4.4.18.4 The parameters ‘header’ and ‘body’ shall not be NULL.

3.4.4.18.5 The parameter ‘qosProperties’ may be NULL.

3.4.4.19 Receive a NOTIFY ERROR message

3.4.4.19.1 A method ‘notifyErrorReceived’ shall be defined in order to receive the NOTIFY ERROR message defined by the IP PUBLISH-SUBSCRIBE.

3.4.4.19.2 The signature of the method ‘notifyErrorReceived’ shall be:

```
void notifyErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.4.4.19.3 The parameters of the method ‘notifyErrorReceived’ shall be assigned as described in table 3-86.

Table 3-86: MALInteractionListener 'notifyErrorReceived' Parameters

Parameter	Description
header	Header of the UPDATE ERROR message
body	Body of the REGISTER ERROR message
qosProperties	QoS properties of the UPDATE ERROR message

3.4.4.19.4 The parameters 'header' and 'body' shall not be NULL.

3.4.4.19.5 The parameter 'qosProperties' may be NULL.

3.4.4.19.6 If an error occurs, then a MALException may be raised.

3.4.4.20 Receive a DEREGISTER ACK message

3.4.4.20.1 A method 'deregisterAckReceived' shall be defined in order to receive the DEREGISTER ACK message defined by the IP PUBLISH-SUBSCRIBE.

3.4.4.20.2 The signature of the method 'deregisterAckReceived' shall be:

```
void deregisterAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const MALQoSProperties& qosProperties)
```

3.4.4.20.3 The parameters of the method 'deregisterAckReceived' shall be assigned as described in table 3-87.

Table 3-87: MALInteractionListener 'deregisterAckReceived' Parameters

Parameter	Description
header	Header of the DEREGISTER ACK message
qosProperties	QoS properties of the DEREGISTER ACK message

3.4.4.20.4 The parameter 'header' shall not be NULL.

3.4.4.20.5 The parameter 'qosProperties' may be NULL.

3.4.4.20.6 If an error occurs, then a MALException may be raised.

3.4.5 MALINTERACTIONADAPTER

3.4.5.1 A MALInteractionAdapter abstract class shall be defined in order to enable a MAL client not to implement all the methods provided by the interface MALInteractionListener.

3.4.5.2 The MALInteractionAdapter class shall implement the MALInteractionListener interface.

3.4.5.3 The MALInteractionAdapter class shall implement all the methods provided by the MALInteractionListener interface with an empty implementation.

3.5 PROVIDER NAMESPACE

3.5.1 OVERVIEW

This part of the API is dedicated to the MAL clients handling interactions as service providers. The provider state diagrams are specified in reference [1].

The classes and interfaces belong to the C++ namespace:

```
mo::mal::provider
```

3.5.2 MALPROVIDERMANAGER

3.5.2.1 Definition

3.5.2.1.1 A MALProviderManager interface shall be defined in order to encapsulate the resources used to enable MAL providers to handle interactions.

3.5.2.1.2 A MALProviderManager shall be a MALProvider factory.

3.5.2.2 MALProviderManager Creation

3.5.2.2.1 A MALProviderManager shall be created by calling the method ‘createProviderManager’ provided by a MALContext.

3.5.2.2.2 Several MALProviderManager instances should be created in order to separate the resources used by the providers.

3.5.2.3 Create a Provider

3.5.2.3.1 Two methods ‘createProvider’ shall be defined in order to create a MALProvider:

a) using a private MALEndpoint;

b) using a shared MALEndpoint.

3.5.2.3.2 The signatures of the method ‘createProvider’ shall be:

```
shared_ptr<MALProvider> createProvider(
      const string& localName,
      const string& protocol,
      const shared_ptr<MALService>& service,
      const shared_ptr<Blob>& authenticationId,
      const shared_ptr<MALInteractionHandler>& handler,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& defaultQoSProperties,
```

```
      const bool& isPublisher,
      const URI& sharedBrokerUri)

shared_ptr<MALProvider> createProvider(
      const shared_ptr<MALEndpoint>& endpoint,
      const shared_ptr<MALService>& service,
      const shared_ptr<Blob>& authenticationId,
      const shared_ptr<MALInteractionHandler>& handler,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& defaultQoSProperties,
      const bool& isPublisher,
      const URI& sharedBrokerUri)
```

3.5.2.3.3 The parameters of the method ‘createProvider’ shall be assigned as described in table 3-88.

Table 3-88: MALProviderManager ‘createProvider’ Parameters

Parameter	Description
localName	Name of the private MALEndpoint to be created and used by the provider
protocol	Name of the protocol used to bind the provider
endpoint	Shared MALEndpoint to be used by the provider
service	Description of the provided service
authenticationId	Authentication identifier to be used by the provider
handler	Interaction handler
expectedQoS	QoS levels the provider can rely on
priorityLevelNumber	Number of priorities the provider uses
defaultQoSProperties	Default QoS properties used by the provider to send messages back to the consumer and to publish updates to a shared broker
isPublisher	Specifies whether the provider is a PUBLISH-SUBSCRIBE publisher or not
sharedBrokerUri	URI of the shared broker to be used

3.5.2.3.4 The parameter ‘localName’ may be NULL.

3.5.2.3.5 If the parameter ‘localName’ is not NULL, then its value shall be unique for the MALContext instance and the protocol specified by the parameter ‘protocol’.

3.5.2.3.6 The parameter ‘defaultQoSProperties’ may be NULL.

3.5.2.3.7 The method ‘createProvider’ shall not return the value NULL.

3.5.2.3.8 If the parameter ‘isPublisher’ is TRUE, then:

a) if the parameter ‘sharedBrokerUri’ is NULL, then a private broker shall be created;

b) otherwise the URI specified by the parameter ‘sharedBrokerUri’ shall be used as the destination URI by the provider when sending PUBLISH-SUBSCRIBE messages.

3.5.2.3.9 The URI specified by the parameter ‘sharedBrokerUri’ shall have the same protocol as specified by the parameter ‘protocol’.

3.5.2.3.10 The private broker shall be aggregated into the MALProvider; i.e., it shall be created and deleted at the same time as the provider to which it belongs.

3.5.2.3.11 The private broker shall be separately identified by its own URI.

3.5.2.3.12 The URI of the private broker may be the same as the URI of the provider to which it belongs.

3.5.2.3.13 A shared broker shall be created by using the broker part of this API.

3.5.2.3.14 If a private MALEndpoint is created, then the message delivery shall be started.

3.5.2.3.15 The provider shall be able to handle interactions through the interface MALInteractionHandler as soon as the method ‘createProvider’ returns.

3.5.2.3.16 If the MALProviderManager is closed or, if an internal error occurs, then a MALException shall be raised.

3.5.2.3.17 If the provider local name is not NULL and if the provider process starts again after a stop and creates the MALProvider with the same local name (directly or through a shared MALEndpoint), then the MALProvider shall recover the same URI as before the stop.

3.5.2.3.18 More than one MALProvider may be created using private brokers and sharing the same MALEndpoint.

3.5.2.3.19 The method ‘handleSend’ provided by the parameter ‘handler’ shall be called as soon as a SEND message has been delivered.

3.5.2.3.20 The method ‘handleSubmit’ provided by the parameter ‘handler’ shall be called as soon as a SUBMIT message has been delivered.

3.5.2.3.21 The method ‘handleRequest’ provided by the parameter ‘handler’ shall be called as soon as a REQUEST message has been delivered.

3.5.2.3.22 The method ‘handleInvoke’ provided by the parameter ‘handler’ shall be called as soon as an INVOKE message has been delivered.

3.5.2.3.23 The method ‘handleProgress’ provided by the parameter ‘handler’ shall be called as soon as a PROGRESS message has been delivered.

NOTE – The same instance of MALInteractionHandler can be passed several times as a parameter of the method ‘createProvider’. In this way the MALInteractionHandler will be activated by several MALProviders.

3.5.2.4 Close

3.5.2.4.1 A method ‘close’ shall be defined in order to release the resources owned by a MALProviderManager.

3.5.2.4.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.5.2.4.3 The method ‘close’ shall close all the MALProviders owned by this MALProviderManager.

3.5.2.4.4 The method ‘close’ shall return after all the MALProviders have been closed.

3.5.2.4.5 If an internal error occurs, then a MALException shall be raised.

3.5.3 MALPROVIDER

3.5.3.1 Definition

3.5.3.1.1 A MALProvider interface shall be defined in order to represent the execution context of a service provider for a given URI.

3.5.3.1.2 If a service provider is to be bound to several transport layers, then several MALProviders shall be created, each one representing a binding between the service provider and a transport layer.

NOTE – Those MALProviders can own the same MALInteractionHandler instance. However, it is not mandatory, as several MALInteractionHandlers can share the same state, for example, a common data base.

3.5.3.2 MALProvider Creation

3.5.3.2.1 A MALProvider shall be created by calling the method ‘createProvider’ provided by a MALProviderManager.

3.5.3.2.2 A MALProvider shall be active as soon as it has been instantiated.

3.5.3.3 Get the URI of the Provider

3.5.3.3.1 The MALProvider interface shall provide a getter method ‘getURI’ that returns the URI of the provider.

3.5.3.3.2 The signature of the method ‘getURI’ shall be:

```
URI getURI()
```

3.5.3.4 Publishing Enabled

3.5.3.4.1 A method ‘isPublisher’ shall be defined in order to indicate whether the provider can publish updates or not.

3.5.3.4.2 The signature of the method ‘isPublisher’ shall be:

```
bool isPublisher()
```

3.5.3.5 Get the URI of the Broker

3.5.3.5.1 The MALProvider interface shall provide a getter method ‘getBrokerURI’ that returns the URI of the private broker.

3.5.3.5.2 The signature of the method ‘getBrokerURI’ shall be:

```
URI getBrokerURI()
```

3.5.3.5.3 The method ‘getBrokerURI’ shall return:

a) NULL if the provider is not linked to a broker; i.e., the method ‘isPublisher’ returns false; or

b) the URI of the broker used by this provider.

3.5.3.6 Get the Authentication Identifier of the Private Broker

3.5.3.6.1 The MALProvider interface shall provide a getter method ‘getBrokerAuthenticationId’ that returns the authentication identifier of the private broker.

3.5.3.6.2 The signature of the method ‘getBrokerAuthenticationId’ shall be:

```
shared_ptr<Blob> getBrokerAuthenticationId()
```

3.5.3.6.3 The method ‘getBrokerAuthenticationId’ shall return:

a) NULL if the provider does not own a private broker; or

b) the authentication identifier of the private broker.

3.5.3.7 Get the service

3.5.3.7.1 The MALProvider interface shall provide a getter method ‘getService’ in order to return the MALService provided by this MALProvider.

3.5.3.7.2 The signature of the method ‘getService’ shall be:

```
shared_ptr<MALService> getService()
```

3.5.3.8 Create a Publisher

3.5.3.8.1 A method ‘createPublisher’ shall be defined in order to create a MALPublisher.

3.5.3.8.2 The signature of the method ‘createPublisher’ shall be:

```
shared_ptr<MALPublisher> createPublisher(
            const shared_ptr<MALPubSubOperation>& op,
            const IdentifierList& domain,
            const Identifier& networkZone,
            const SessionType& sessionType,
            const Identifier& sessionName,
            const QoSLevel& remotePublisherQos,
            const MALQoSProperties& remotePublisherQosProps,
            const uint32_t& remotePublisherPriority)
```

3.5.3.8.3 The parameters of the method ‘createPublisher’ shall be assigned as described in table 3-89.

Table 3-89: MALProvider ‘createPublisher’ Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation
domain	Domain of the PUBLISH messages
networkZone	Network zone of the PUBLISH messages
sessionType	Session type of the PUBLISH messages
sessionName	Session name of the PUBLISH messages
remotePublisherQos	QoS level of the PUBLISH messages
remotePublisherQosProps	QoS properties of the PUBLISH messages
remotePublisherPriority	Priority of the PUBLISH messages

3.5.3.8.4 The parameters ‘remotePublisherQos’, ‘remotePublisherQosProps’, and ‘remotePublisherPriority’ shall be ignored if the broker is local.

3.5.3.8.5 If the MALProvider is not a publisher, then a MALException shall be raised.

3.5.3.8.6 If the MALProvider is closed, then a MALException shall be raised.

3.5.3.8.7 If an internal error occurs, then a MALException shall be raised.

3.5.3.9 Set the Transmit Error Listener

3.5.3.9.1 A method ‘setTransmitErrorListener’ shall be defined in order to set a MALTransmitErrorListener.

3.5.3.9.2 The signature of the method ‘setTransmitErrorListener’ shall be:

```
void setTransmitErrorListener(
            const shared_ptr<MALTransmitErrorListener>& listener)
```

3.5.3.9.3 The parameter of the method ‘setTransmitErrorListener’ shall be assigned as described in table 3-90.

Table 3-90: MALProvider ‘setTransmitErrorListener’ Parameter

Parameter	Description
listener	Listener in charge of receiving every asynchronous TRANSMIT ERROR

3.5.3.9.4 The parameter ‘listener’ may be NULL.

3.5.3.9.5 The MALTransmitErrorListener shall be called when a TRANSMIT ERROR is asynchronously returned to the provider.

3.5.3.9.6 If the MALProvider is closed, then a MALException shall be raised.

3.5.3.9.7 If an internal error occurs, then a MALException shall be raised.

3.5.3.10 Get the Transmit Error Listener

3.5.3.10.1 A method ‘getTransmitErrorListener’ shall be defined in order to return the MALTransmitErrorListener.

3.5.3.10.2 The signature of the method ‘getTransmitErrorListener’ shall be:

```
shared_ptr<MALTransmitErrorListener> getTransmitErrorListener()
```

3.5.3.10.3 If no MALTransmitErrorListener has been set, then the method ‘getTransmitErrorListener’ shall return NULL.

3.5.3.10.4 If the MALProvider is closed, then a MALException shall be raised.

3.5.3.10.5 If an internal error occurs, then a MALException shall be raised.

3.5.3.11 Close

3.5.3.11.1 A method ‘close’ shall be defined in order to deactivate the MALInteractionHandler so that no interaction is handled any longer.

3.5.3.11.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.5.3.11.3 A ‘close’ should be called by a MAL client before the provider process is stopped for any operational reason.

3.5.3.11.4 The method ‘close’ shall deactivate this MALProvider.

3.5.3.11.5 If the MALInteractionHandler is being activated at the time of the close, then the closing process shall wait for the end of the handler’s execution.

3.5.3.11.6 Once a MALProvider has been closed, the message delivery shall be stopped in order that the MALInteractionHandler does not receive any message through this MALProvider anymore.

3.5.3.11.7 If an internal error occurs, then a MALException shall be raised.

3.5.3.11.8 If the provider owns a private MALEndpoint, then the MALEndpoint shall be closed.

3.5.3.11.9 Pending interactions shall be finalized by returning the error DESTINATION_LOST_OR_DIED to the consumers.

3.5.3.11.10 If the provider owns a private broker,

a) the private broker shall be deleted;

b) the subscribers list shall be cleared;

c) an error DESTINATION_LOST_OR_DIED shall be sent to the subscribers in order to inform them that they are no longer subscribed to the private broker.

3.5.3.11.11 If a consumer tries to interact with a closed provider and if the QoS level is not QUEUED, a DELIVERY_FAILED shall be returned to the consumer.

3.5.3.11.12 If a consumer tries to interact with a closed provider and if the QoS level is QUEUED, then the consumer’s request shall be queued and delivered as soon as the provider message delivery is started again.

3.5.4 MALINTERACTIONHANDLER

3.5.4.1 Definition

3.5.4.1.1 A MALInteractionHandler interface shall be defined in order to handle the interactions on the provider side.

3.5.4.1.2 The MALInteractionHandler interface shall not handle the PUBLISH-SUBSCRIBE REGISTER, PUBLISH REGISTER, PUBLISH, DEREGISTER and PUBLISH DEREGISTER messages.

NOTE – The messages REGISTER, PUBLISH REGISTER, PUBLISH, DEREGISTER and PUBLISH DEREGISTER are handled by a broker.

3.5.4.1.3 A MALInteractionHandler shall be passed as a parameter of the method 'createProvider' when creating a MALProvider.

NOTE – Several MALProviders may use the same instance of MALInteractionHandler.

3.5.4.2 Initialize

3.5.4.2.1 A method 'initialize' shall be defined in order to enable a MALInteractionHandler to be initialized when the provider is activated.

3.5.4.2.2 The signature of the method 'malInitialize' shall be:

```
void initialize(const shared_ptr<MALProvider>& provider)
```

3.5.4.2.3 The parameter of the method 'initialize' shall be assigned as described in table 3-91.

Table 3-91: MALInteractionHandler 'malInitialize' Parameter

Parameter	Description
provider	Created MALProvider

3.5.4.2.4 The method 'initialize' shall be called when the MALProvider is created.

NOTE – The method 'initialize' enables the handler to get the reference of the MALProvider and use it if necessary.

3.5.4.2.5 If an instance of MALInteractionHandler is used by several MALProviders, then the method 'initialize' shall be called once for each MALProvider.

3.5.4.2.6 A MALException may be raised if an error occurs.

3.5.4.3 Handle a SEND IP

3.5.4.3.1 A method ‘handleSend’ shall be defined in order to handle a SEND interaction.

3.5.4.3.2 The signature of the method ‘handleSend’ shall be:

```
void handleSend(
            const shared_ptr<MALInteraction>& interaction,
            const shared_ptr<MALMessageBody>& body)
```

3.5.4.3.3 The parameters of the method ‘handleSend’ shall be assigned as described in table 3-92.

Table 3-92: MALInteractionHandler ‘handleSend’ Parameters

Parameter	Description
interaction	SEND interaction context
body	Message body

3.5.4.3.4 The parameter ‘interaction’ shall not be NULL.

3.5.4.3.5 The parameter ‘body’ may be NULL.

3.5.4.3.6 If an error occurs, then a MALException may be raised.

3.5.4.3.7 A MALInteractionException may be raised.

3.5.4.4 Handle a SUBMIT IP

3.5.4.4.1 A method ‘handleSubmit’ shall be defined in order to handle a SUBMIT interaction.

3.5.4.4.2 The signature of the method ‘handleSubmit’ shall be:

```
void handleSubmit(
            const shared_ptr<MALSubmit>& interaction,
            const shared_ptr<MALMessageBody>& body)
```

3.5.4.4.3 The parameters of the method ‘handleSubmit’ shall be assigned as described in table 3-93.

Table 3-93: MALInteractionHandler 'handleSubmit' Parameters

Parameter	Description
interaction	SUBMIT interaction context
body	Message body

3.5.4.4.4 The parameter 'interaction' shall not be NULL.

3.5.4.4.5 The parameter 'body' may be NULL.

3.5.4.4.6 The ACK and ACK ERROR message body shall be returned through the MALSubmit context.

3.5.4.4.7 If a MAL standard error occurs it shall be returned through the MALSubmit context.

3.5.4.4.8 If a non-MAL error occurs, then a MALException may be raised.

3.5.4.4.9 A MALInteractionException may be raised.

3.5.4.5 Handle a REQUEST IP

3.5.4.5.1 A method 'handleRequest' shall be defined in order to handle a REQUEST interaction.

3.5.4.5.2 The signature of the method 'handleRequest' shall be:

```
void handleRequest(
            const shared_ptr<MALRequest>& interaction,
            const shared_ptr<MALMessageBody>& body)
```

3.5.4.5.3 The parameters of the method 'handleRequest' shall be assigned as described in table 3-94.

Table 3-94: MALInteractionHandler 'handleRequest' Parameters

Parameter	Description
interaction	REQUEST interaction context
body	Message body

3.5.4.5.4 The parameter 'interaction' shall not be NULL.

3.5.4.5.5 The parameter 'body' may be NULL.

3.5.4.5.6 The RESPONSE and RESPONSE ERROR message body shall be returned through the MALRequest context.

3.5.4.5.7 If a MAL standard error occurs it shall be returned through the MALRequest context.

3.5.4.5.8 If a non-MAL error occurs, then a MALException may be raised.

3.5.4.5.9 A MALInteractionException may be raised.

3.5.4.6 Handle an INVOKE IP

3.5.4.6.1 A method ‘handleInvoke’ shall be defined in order to handle a REQUEST interaction.

3.5.4.6.2 The signature of the method ‘handleInvoke’ shall be:

```
void handleInvoke(
            const shared_ptr<MALInvoke>& interaction,
            const shared_ptr<MALMessageBody>& body)
```

3.5.4.6.3 The parameters of the method ‘handleInvoke’ shall be assigned as described in table 3-95.

Table 3-95: MALInteractionHandler ‘handleInvoke’ Parameters

Parameter	Description
interaction	INVOKE interaction context
body	Message body

3.5.4.6.4 The parameter ‘interaction’ shall not be NULL.

3.5.4.6.5 The parameter ‘body’ may be NULL.

3.5.4.6.6 The ACK, ACK ERROR, RESPONSE and RESPONSE ERROR message body shall be returned through the MALInvoke context.

3.5.4.6.7 If a MAL standard error occurs it shall be returned through the MALInvoke context.

3.5.4.6.8 If a non-MAL error occurs, then a MALException may be raised.

3.5.4.6.9 A MALInteractionException may be raised.

3.5.4.7 Handle a PROGRESS IP

3.5.4.7.1 A method ‘handleProgress’ shall be defined in order to handle a PROGRESS interaction.

3.5.4.7.2 The signature of the method ‘handleProgress’ shall be:

```
void handleProgress(
            const shared_ptr<MALProgress>& interaction,
            const shared_ptr<MALMessageBody>& body
```

3.5.4.7.3 The parameters of the method ‘handleProgress’ shall be assigned as described in table 3-96.

Table 3-96: MALInteractionHandler ‘handleProgress’ Parameters

Parameter	Description
interaction	PROGRESS interaction context
body	Message body

3.5.4.7.4 The parameter ‘interaction’ shall not be NULL.

3.5.4.7.5 The parameter ‘body’ may be NULL.

3.5.4.7.6 The ACK, ACK ERROR, RESPONSE and RESPONSE ERROR message body shall be returned through the MALProgress context.

3.5.4.7.7 If a MAL standard error occurs it shall be returned through the MALProgress context.

3.5.4.7.8 If a non-MAL error occurs, then a MALException may be raised.

3.5.4.7.9 A MALInteractionException may be raised.

3.5.4.8 Finalize

3.5.4.8.1 A method ‘finalize’ shall be defined in order to enable a MALInteractionHandler to be notified when a MALProvider is closed.

3.5.4.8.2 The signature of the method ‘malFinalize’ shall be:

```
void finalize(const shared_ptr<MALProvider>& provider)
```

3.5.4.8.3 The parameter of the method ‘finalize’ shall be assigned as described in table 3-97.

Table 3-97: MALInteractionHandler 'finalize' Parameter

Parameter	Description
provider	Closed MALProvider

3.5.4.8.4 The method 'finalize' shall be called when the MALProvider is closed.

3.5.4.8.5 A MALException may be raised if an error occurs.

3.5.5 MALINTERACTION

3.5.5.1 Definition

3.5.5.1.1 A MALInteraction interface shall be defined in order to represent a generic IP handling context.

3.5.5.1.2 The MALInteraction interface shall be extended by a specific interaction interface for each IP except SEND and PUBLISH-SUBSCRIBE:

a) SEND shall be handled by passing a MALInteraction;

b) PUBLISH-SUBSCRIBE shall not be handled by a MALInteractionHandler.

3.5.5.2 Get the Message Header

3.5.5.2.1 A method 'getMessageHeader' shall be defined in order to return the header of the message that initiated the interaction.

3.5.5.2.2 The signature of the method 'getMessageHeader' shall be:

```
shared_ptr<MALMessageHeader> getMessageHeader()
```

3.5.5.3 Get the Operation

3.5.5.3.1 A method 'getOperation' shall be defined in order to return the operation called through this interaction.

3.5.5.3.2 The signature of the method 'getOperation' shall be:

```
shared_ptr<MALOperation> getOperation()
```

3.5.5.4 Get/Set a QoS Property

3.5.5.4.1 The methods 'setQoSProperty' and 'getQoSProperty' shall be defined in order to enable getting and setting of the value of a QoS property.

3.5.5.4.2 The signatures of ‘setQoSProperty’ and ‘getQoSProperty’ shall be:

```
void setQoSProperty(const string& name, const string& value)

string getQoSProperty(const string& name)
```

3.5.5.4.3 The parameters of ‘setQoSProperty’ and ‘getQoSProperty’ shall be assigned as described in table 3-98.

Table 3-98: MALInteractionHandler ‘get/setQoSProperty’ Parameters

Parameter	Description
name	Name of the property
value	Value of the property

3.5.6 MALSUBMIT

3.5.6.1 Definition

3.5.6.1.1 A MALSubmit interface shall be defined in order to represent a SUBMIT interaction handling context.

3.5.6.1.2 The MALSubmit interface shall inherit from the interface MALInteraction.

3.5.6.2 Send an ACK

3.5.6.2.1 A method ‘sendAcknowledgement’ shall be defined in order to send an ACK message.

3.5.6.2.2 The signature of the method ‘sendAcknowledgement’ shall be:

```
shared_ptr<MALMessage> sendAcknowledgement()
```

NOTE – The SUBMIT ACK message body is empty.

3.5.6.2.3 The method ‘sendAcknowledgement’ shall check the interaction state as follows:

a) if the state is INITIATED an ACK message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.6.2.4 The method ‘sendAcknowledgement’ shall return as soon as the ACK message has been sent.

3.5.6.2.5 The method ‘sendAcknowledgement’ shall return the MALMessage that has been sent.

3.5.6.2.6 If an error occurs, then a MALException shall be raised.

3.5.6.3 Send an ACK ERROR

3.5.6.3.1 A method ‘sendError’ shall be defined in order to send an ACK ERROR message.

3.5.6.3.2 The signature of the method ‘sendError’ shall be:

```
shared_ptr<MALMessage> sendError(
            const shared_ptr<MALStandardError>& error)
```

3.5.6.3.3 The parameter of the method ‘sendError’ shall be assigned as described in table 3-99.

Table 3-99: MALSubmit ‘sendError’ Parameter

Parameter	Description
error	Error to be transmitted to the consumer

3.5.6.3.4 The method ‘sendError’ shall check the interaction state as follows:

a) if the state is INITIATED an ACK ERROR message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.6.3.5 The method ‘sendError’ shall return as soon as the ACK ERROR message has been sent.

3.5.6.3.6 The method ‘sendError’ shall return the MALMessage that has been sent.

3.5.6.3.7 If an error occurs, then a MALException shall be raised.

3.5.7 MALREQUEST

3.5.7.1 Definition

3.5.7.1.1 A MALRequest interface shall be defined in order to represent a REQUEST interaction handling context.

3.5.7.1.2 The MALRequest interface shall inherit from the interface MALInteraction.

3.5.7.2 Send a RESPONSE

3.5.7.2.1 Two methods ‘sendResponse’ shall be defined in order to send a RESPONSE message:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.5.7.2.2 The signatures of the method ‘sendResponse’ shall be:

```
shared_ptr<MALMessage> sendResponse(
                  const vector<shared_ptr<MessageBody>>& body)

shared_ptr<MALMessage> sendResponse(
                  const shared_ptr<MALEncodedBody>& encodedBody)
```

3.5.7.2.3 The parameter of the method ‘sendResponse’ shall be assigned as described in table 3-100.

Table 3-100: MALResponse ‘sendResponse’ Parameter

Parameter	Description
body	Message body elements to be transmitted to the consumer
encodedBody	Encoded body to be transmitted to the consumer

3.5.7.2.4 The parameter ‘body’ may be empty vector.

3.5.7.2.5 The parameter ‘encodedBody’ may be NULL.

3.5.7.2.6 The method ‘sendResponse’ shall check the interaction state as follows:

a) if the state is INITIATED a RESPONSE message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.7.2.7 The method ‘sendResponse’ shall return as soon as the RESPONSE message has been sent.

3.5.7.2.8 The method ‘sendResponse’ shall return the MALMessage that has been sent.

3.5.7.2.9 If an error occurs, then a MALException shall be raised.

3.5.7.2.10 The allowed body element types shall be:

a) MAL element types;

b) MALEncodedElement;

c) C++ types defined by a specific C++ mapping extension.

3.5.7.3 Send a RESPONSE ERROR

3.5.7.3.1 A method ‘sendError’ shall be defined in order to send a RESPONSE message.

3.5.7.3.2 The signature of the method ‘sendError’ shall be:

```
shared_ptr<MALMessage> sendError(
                  const shared_ptr<MALStandardError>& error)
```

3.5.7.3.3 The parameter of the method ‘sendError’ shall be assigned as described in table 3-101.

Table 3-101: MALResponse ‘sendError’ Parameter

Parameter	Description
error	Error to be transmitted to the consumer

3.5.7.3.4 The method ‘sendError’ shall check the interaction state as follows:

a) if the state is INITIATED a RESPONSE ERROR message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.7.3.5 The method ‘sendError’ shall return as soon as the RESPONSE ERROR message has been sent.

3.5.7.3.6 The method ‘sendError’ shall return the MALMessage that has been sent.

3.5.7.3.7 If an error occurs, then a MALException shall be raised.

3.5.8 MALINVOKE

3.5.8.1 Definition

3.5.8.1.1 A MALInvoke interface shall be defined in order to represent an INVOKE interaction handling context.

3.5.8.1.2 The MALInvoke interface shall inherit from the interface MALRequest.

3.5.8.2 Send an ACK

3.5.8.2.1 Two methods ‘sendAcknowledgement’ shall be defined in order to send an ACK message:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.5.8.2.2 The signatures of the method ‘sendAcknowledgement’ shall be:

```
shared_ptr<MALMessage> sendAcknowledgement(
                  const vector<shared_ptr<MessageBody>>& body)

shared_ptr<MALMessage> sendAcknowledgement(
const shared_ptr<MALEncodedBody>& encodedBody)
```

3.5.8.2.3 The parameter of the method ‘sendAcknowledgement’ shall be assigned as described in table 3-102.

Table 3-102: MALInvoke ‘sendAcknowledgement’ Parameters

Parameter	Description
body	Message body elements to be transmitted to the consumer
encodedBody	Encoded body to be transmitted to the consumer

3.5.8.2.4 The method ‘sendAcknowledgement’ shall check the interaction state as follows:

a) if the state is INITIATED an ACK message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.8.2.5 The method ‘sendAcknowledgement’ shall return as soon as the ACK message has been sent.

3.5.8.2.6 The method ‘sendAcknowledgement’ shall return the MALMessage that has been sent.

3.5.8.2.7 If an error occurs, then a MALException shall be raised.

3.5.8.2.8 The allowed body element types shall be:

a) MAL element types;

b) MALEncodedElement;

c) C++ types defined by a specific C++ mapping extension.

3.5.8.3 Send a RESPONSE

3.5.8.3.1 The method ‘sendResponse’ inherited from MALRequest shall check the interaction state as follows:

a) if the state is ACKNOWLEDGED a RESPONSE message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.8.3.2 If an error occurs, then a MALException shall be raised.

3.5.8.4 Send an ACK or a RESPONSE ERROR

3.5.8.4.1 The method ‘sendError’ inherited from MALRequest shall check the interaction state as follows:

a) if the state is INITIATED an ACK ERROR message shall be sent;

b) otherwise, if the state is ACKNOWLEDGED, a RESPONSE ERROR message shall be sent;

c) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.8.4.2 If an error occurs, then a MALException shall be raised.

3.5.9 MALPROGRESS

3.5.9.1 Definition

3.5.9.1.1 A MALProgress interface shall be defined in order to represent a PROGRESS interaction handling context.

3.5.9.1.2 The MALProgress interface shall inherit from the MALInvoke interface.

3.5.9.2 Send an ACK

3.5.9.2.1 The method ‘sendAcknowledgement’ inherited from MALInvoke shall check the interaction state as follows:

a) if the state is INITIATED an ACK message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.9.2.2 If an error occurs, then a MALException shall be raised.

3.5.9.3 Send an UPDATE

3.5.9.3.1 Two methods ‘sendUpdate’ shall be defined in order to send an UPDATE message:

a) declaring parameters for the body elements;

b) declaring a parameter for the encoded body.

3.5.9.3.2 The signatures of the method ‘sendUpdate’ shall be:

```
shared_ptr<MALMessage> sendUpdate(
                  const vector<shared_ptr<MessageBody>>& body)

shared_ptr<MALMessage> sendUpdate(
            const shared_ptr<MALEncodedBody>& encodedBody)
```

3.5.9.3.3 The parameter of the method ‘sendUpdate’ shall be assigned as described in table 3-103.

Table 3-103: MALProgress ‘sendUpdate’ Parameters

Parameter	Description
body	Message body elements to be transmitted to the consumer
encodedBody	Encoded body to be transmitted to the consumer

3.5.9.3.4 The parameter ‘body’ may be NULL.

3.5.9.3.5 The parameter ‘encodedBody’ may be NULL.

3.5.9.3.6 The method ‘sendUpdate’ shall check the interaction state as follows:

a) if the state is either ACKNOWLEDGED or PROGRESSING an UPDATE message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.9.3.7 The method ‘sendUpdate’ shall return as soon as the UPDATE message has been sent.

3.5.9.3.8 The method ‘sendUpdate’ shall return the MALMessage that has been sent.

3.5.9.3.9 If an error occurs, then a MALException shall be raised.

3.5.9.3.10 The allowed body element types shall be:

a) MAL element types;

b) MALEncodedElement;

c) C++ types defined by a specific C++ mapping extension.

3.5.9.4 Send a RESPONSE

3.5.9.4.1 The method ‘sendResponse’ inherited from MALRequest shall check the interaction state as follows:

a) if the state is either ACKNOWLEDGED or PROGRESSING a RESPONSE message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.9.4.2 If an error occurs, then a MALException shall be raised.

3.5.9.5 Send an ACK or a RESPONSE ERROR

3.5.9.5.1 The method ‘sendError’ inherited from MALRequest shall check the interaction state as follows:

a) if the state is INITIATED an ACK ERROR message shall be sent;

b) if the state is either ACKNOWLEDGED or PROGRESSING a RESPONSE ERROR message shall be sent;

c) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.9.5.2 If an error occurs, then a MALException shall be raised.

3.5.9.6 Send an UPDATE ERROR

3.5.9.6.1 A method ‘sendUpdateError’ shall be defined in order to send an UPDATE ERROR message.

3.5.9.6.2 The signature of the method ‘sendUpdateError’ shall be:

```
shared_ptr<MALMessage> sendUpdateError(
                        const shared_ptr<MALStandardError>& error)
```

3.5.9.6.3 The parameter of the method ‘sendUpdateError’ shall be assigned as described in table 3-104.

Table 3-104: MALProgress 'sendUpdateError' Parameter

Parameter	Description
error	Error to be transmitted to the consumer

3.5.9.6.4 The method 'sendUpdateError' shall check the interaction state as follows:

a) if the state is either ACKNOWLEDGED or PROGRESSING an UPDATE ERROR message shall be sent;

b) otherwise a MALInteractionException containing a MAL::INCORRECT_STATE error shall be raised.

3.5.9.6.5 The method 'sendUpdateError' shall return as soon as the UPDATE ERROR message has been sent.

3.5.9.6.6 The method 'sendUpdateError' shall return the MALMessage that has been sent.

3.5.9.6.7 If an error occurs, then a MALException shall be raised.

3.5.10 MALPUBLISHER

3.5.10.1 Definition

A MALPublisher interface shall be defined in order to enable a provider to publish updates and errors to registered consumers.

3.5.10.2 Publish Updates

3.5.10.2.1 A method 'publish' shall be defined in order to publish a list of updates.

3.5.10.2.2 The signature of the method 'publish' shall be:

```
shared_ptr<MALMessage> publish(
            const shared_ptr<UpdateHeaderList>& updateHeaderList,
            const vector<UpdateList>& updateLists)
```

NOTE – UpdateList is defined as a vector<shared_ptr<Element>>

3.5.10.2.3 The parameters of the method 'publish' shall be assigned as described in table 3-105.

Table 3-105: MALPublisher 'publish' Parameters

Parameter	Description
updateHeaderList	Published UpdateHeaders
updateLists	Lists of updates to be published

3.5.10.2.4 The source URI of every UpdateHeader shall be set by the MAL layer.

3.5.10.2.5 The method 'publish' shall return as soon as the PUBLISH message has been sent.

3.5.10.2.6 The method 'publish' shall return the MALMessage that has been sent.

3.5.10.2.7 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.5.10.2.8 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.5.10.2.9 If the MALPublisher is closed, then a MALException shall be raised.

3.5.10.2.10 The allowed update list types shall be:

a) a MAL element list;

b) a List<MALEncodedElement> containing the encoded updates;

c) a List defined by a specific C++ mapping extension.

3.5.10.3 Register

3.5.10.3.1 A method 'syncRegister' shall be defined in order to enable a provider to synchronously register to its broker.

3.5.10.3.2 The signature of the method 'syncRegister' shall be:

```
void syncRegister(
            const shared_ptr<EntityKeyList>& entityKeyList,
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.10.3.3 The parameters of the method 'syncRegister' shall be assigned as described in table 3-106.

Table 3-106: MALPublisher 'syncRegister' Parameters

Parameter	Description
entityKeyList	Keys of the entities that are to be published
listener	Listener in charge of receiving the messages PUBLISH ERROR

3.5.10.3.4 The method 'syncRegister' shall return as soon as the PUBLISH REGISTER ACK message has been received.

3.5.10.3.5 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.5.10.3.6 A MALInteractionException shall be thrown if a PUBLISH REGISTER ERROR occurs.

3.5.10.3.7 The PUBLISH REGISTER ERROR message body shall be passed as the parameter 'standardError' of the MALInteractionException constructor.

3.5.10.3.8 If a MAL error occurs then a MALInteractionException shall be thrown.

3.5.10.3.9 The method 'errorReceived' provided by the parameter 'listener' shall be called if a PUBLISH ERROR occurs.

3.5.10.3.10 The call to 'errorReceived' shall have the following parameters:

a) the PUBLISH ERROR message header passed as the parameter 'header';

b) the PUBLISH ERROR message body passed as the parameter 'error'.

3.5.10.3.11 If a call to 'syncRegister' is initiated while the publisher is already registered but with a different parameter 'listener', then only the second registered 'listener' shall receive the PUBLISH ERROR messages.

3.5.10.3.12 If the MALPublisher is closed, then a MALException shall be raised.

3.5.10.3.13 If a non MAL error occurs then, a MALException shall be thrown.

3.5.10.4 Deregister

3.5.10.4.1 A method 'deregister' shall be defined in order to enable a provider to synchronously deregister from its broker.

3.5.10.4.2 The signature of the method 'deregister' shall be:

```
void deregister()
```

3.5.10.4.3 The method ‘deregister’ shall return as soon as the PUBLISH DEREGISTER ACK message has been received.

3.5.10.4.4 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.5.10.4.5 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.5.10.5 Asynchronous Register

3.5.10.5.1 A method ‘asyncRegister’ shall be defined in order to enable a provider to asynchronously register to its broker.

3.5.10.5.2 The signature of the method ‘asyncRegister’ shall be:

```
shared_ptr<MALMessage> asyncRegister(
            const shared_ptr<EntityKeyList>& entityKeyList,
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.10.5.3 The parameters of the method ‘asyncRegister’ shall be assigned as described in table 3-107.

Table 3-107: MALPublisher ‘asyncRegister’ Parameters

Parameter	Description
entityKeyList	Keys of the entities that are to be published
listener	Listener in charge of receiving the messages PUBLISH REGISTER ACK, PUBLISH REGISTER ERROR and PUBLISH ERROR

3.5.10.5.4 The method ‘asyncRegister’ shall return as soon as the PUBLISH REGISTER message has been sent.

3.5.10.5.5 The method ‘asyncRegister’ shall return the MALMessage that has been sent.

3.5.10.5.6 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.5.10.5.7 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.5.10.5.8 The method ‘publishAckReceived’ provided by the parameter ‘listener’ shall be called as soon as the PUBLISH REGISTER ACK message has been delivered.

3.5.10.5.9 The call to ‘acknowledgementReceived’ shall assign the PUBLISH REGISTER ACK message header to the parameter ‘header’.

3.5.10.5.10 The method ‘errorReceived’ provided by the parameter ‘listener’ shall be called if a PUBLISH REGISTER ERROR or a PUBLISH ERROR occurs.

3.5.10.5.11 The call to ‘errorReceived’ shall have the following parameters:

a) the ERROR message header passed as the parameter ‘header’;

b) the ERROR message body passed as the parameter ‘error’.

3.5.10.5.12 If a call to ‘register’ is initiated while the publisher is already registered but with a different parameter ‘listener’ then only the second registered ‘listener’ shall receive the PUBLISH ERROR messages.

3.5.10.5.13 If the MALPublisher is closed, then a MALException shall be raised.

3.5.10.6 Asynchronous Deregister

3.5.10.6.1 A method ‘asyncDeregister’ shall be defined in order to enable a provider to asynchronously deregister from its broker.

3.5.10.6.2 The signature of the method ‘asyncDeregister’ shall be:

```
shared_ptr<MALMessage> asyncDeregister(
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.10.6.3 The parameter of the method ‘asyncDeregister’ shall be assigned as described in table 3-108.

Table 3-108: MALPublisher ‘asyncDeregister’ Parameter

Parameter	Description
listener	Listener in charge of receiving the messages PUBLISH DEREGISTER ACK

3.5.10.6.4 The method ‘asyncDeregister’ shall return as soon as the PUBLISH DEREGISTER message has been sent.

3.5.10.6.5 The method ‘asyncDeregister’ shall return the MALMessage that has been sent.

3.5.10.6.6 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.5.10.6.7 A MALException shall be thrown if a non-MAL error occurs during the initiation message sending.

3.5.10.6.8 The method ‘acknowledgementReceived’ provided by the parameter ‘listener’ shall be called as soon as the PUBLISH DEREGISTER ACK message has been delivered.

3.5.10.6.9 The call to ‘acknowledgementReceived’ shall assign the PUBLISH DEREGISTER ACK message header to the parameter ‘header’.

3.5.10.6.10 If the MALPublisher is closed, then a MALException shall be raised.

3.5.10.7 Get the MALProvider

3.5.10.7.1 A getter method ‘getProvider’ shall be defined in order to return the MALProvider that created this MALPublisher.

3.5.10.7.2 The signature of the method ‘getProvider’ shall be:

```
shared_ptr<MALProvider> getProvider()
```

3.5.10.8 Close

3.5.10.8.1 A method ‘close’ shall be defined in order to release the resources owned by this MALPublisher.

3.5.10.8.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.5.10.8.3 A closed MALPublisher may still be registered.

3.5.10.8.4 A closed MALPublisher shall not deliver any asynchronous messages anymore, i.e., PUBLISH ERROR, PUBLISH REGISTER ACK, and PUBLISH DEREGISTER ACK.

3.5.11 MALPUBLISHINTERACTIONLISTENER

3.5.11.1 Definition

A MALPublishInteractionListener interface shall be defined in order to allow the reception of the publish interaction results.

3.5.11.2 Receive a PUBLISH REGISTER ACK

3.5.11.2.1 A method ‘publishRegisterAckReceived’ shall be defined in order to receive the PUBLISH REGISTER ACK message defined by the IP PUBLISH-SUBSCRIBE.

3.5.11.2.2 The signature of the method ‘publishRegisterAckReceived’ shall be:

```
void publishRegisterAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const MALQoSProperties& qosProperties)
```

3.5.11.2.3 The parameters of the method ‘publishRegisterAckReceived’ shall be assigned as described in table 3-109.

Table 3-109: MALPublishInteractionListener ‘publishRegisterAckReceived’ Parameters

Parameter	Description
header	Header of the message
qosProperties	QoS properties of the message

3.5.11.2.4 The parameter ‘header’ shall not be NULL.

3.5.11.2.5 The parameter ‘qosProperties’ may be NULL.

3.5.11.2.6 If an error occurs, then a MALException may be raised.

3.5.11.3 Receive a PUBLISH REGISTER ERROR

3.5.11.3.1 A method ‘errorReceived’ shall be defined in order to receive the PUBLISH REGISTER ERROR message defined by the IP PUBLISH-SUBSCRIBE.

3.5.11.3.2 The signature of the method ‘errorReceived’ shall be:

```
void publishRegisterErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.5.11.3.3 The parameters of the method ‘publishRegisterErrorReceived’ shall be assigned as described in table 3-110.

Table 3-110: MALPublishInteractionListener 'publishRegisterErrorReceived' Parameters

Parameter	Description
header	Header of the message
body	Body of the message
qosProperties	QoS properties of the message

3.5.11.3.4 The parameters 'header' and 'body' shall not be NULL.

3.5.11.3.5 The parameter 'qosProperties' may be NULL.

3.5.11.3.6 If an error occurs, then a MALException may be raised.

3.5.11.4 Receive a PUBLISH ERROR

3.5.11.4.1 A method 'publishErrorReceived' shall be defined in order to receive the PUBLISH ERROR message defined by the IP PUBLISH-SUBSCRIBE.

3.5.11.4.2 The signature of the method 'publishErrorReceived' shall be:

```
void publishErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALErrorBody>& body,
                  const MALQoSProperties& qosProperties)
```

3.5.11.4.3 The parameters of the method 'publishErrorReceived' shall be assigned as described in table 3-111.

Table 3-111: MALPublishInteractionListener 'publishErrorReceived' Parameters

Parameter	Description
header	Header of the message
body	Body of the message
qosProperties	QoS properties of the message

3.5.11.4.4 The parameters 'header' and 'body' shall not be NULL.

3.5.11.4.5 The parameter 'qosProperties' may be NULL.

3.5.11.4.6 If an error occurs, then a MALException may be raised.

3.5.11.5 Receive a PUBLISH DEREGISTER ACK

3.5.11.5.1 A method 'publishDeregisterAckReceived' shall be defined in order to receive the PUBLISH DEREGISTER ACK message defined by the IP PUBLISH-SUBSCRIBE.

3.5.11.5.2 The signature of the method 'publishDeregisterAckReceived' shall be:

```
void publishDeregisterAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const MALQoSProperties& qosProperties)
```

3.5.11.5.3 The parameters of the method 'publishDeregisterAckReceived' shall be assigned as described in table 3-112.

Table 3-112: MALPublishInteractionListener 'publishDeregisterAckReceived' Parameters

Parameter	Description
header	Header of the message
qosProperties	QoS properties of the message

3.5.11.5.4 The parameter 'header' shall not be NULL.

3.5.11.5.5 The parameter 'qosProperties' may be NULL.

3.5.11.5.6 If an error occurs, then a MALException may be raised.

3.5.12 MALPROVIDERSET

3.5.12.1 Definition

A MALProviderSet class shall be defined in order to:

a) manage a list of MALProvider providing the same service through different protocols;

b) create instances of MALPublisherSet, register their references and update them when a MALProvider is either added or removed.

3.5.12.2 Creation

3.5.12.2.1 A MALProviderSet public constructor shall be defined.

3.5.12.2.2 The MALProviderSet constructor signature shall be:

```
MALProviderSet(shared_ptr<MALService>& service)
```

3.5.12.2.3 The MALProviderSet constructor parameter shall be assigned as described in table 3-113.

Table 3-113: MALProviderSet Constructor Parameter

Parameter	Description
service	The service to be provided by the MALProviders added in this MALProviderSet

3.5.12.3 Create a MALPublisherSet

3.5.12.3.1 A method 'createPublisherSet' shall be defined in order to create a MALPublisherSet and register its reference.

3.5.12.3.2 The signature of the method 'createPublisherSet' shall be:

```
shared_ptr<MALPublisherSet> createPublisherSet(
  const shared_ptr<MALPubSubOperation>& op,
  const IdentifierList& domain,
  const Identifier& networkZone,
  const SessionType& sessionType,
  const Identifier& sessionName,
  const QoSLevel& remotePublisherQos,
  const MALQoSProperties& remotePublisherQosProps,
  const uint32_t& remotePublisherPriority)
```

3.5.12.3.3 The parameters of the method 'createPublisherSet' shall be assigned as described in table 3-114.

Table 3-114: MALProviderSet 'createPublisherSet' Parameters

Parameter	Description
op	PUBLISH-SUBSCRIBE operation
domain	Domain of the PUBLISH messages
networkZone	Network zone of the PUBLISH messages
sessionType	Session type of the PUBLISH messages
sessionName	Session name of the PUBLISH messages
remotePublisherQos	QoS level of the PUBLISH messages
remotePublisherQosProps	QoS properties of the PUBLISH messages
remotePublisherPriority	Priority of the PUBLISH messages

3.5.12.3.4 The parameters 'remotePublisherQos', 'remotePublisherQosProps', and 'remotePublisherPriority' shall be ignored if the broker is local.

3.5.12.3.5 The reference of the returned MALPublisherSet shall be registered in this MALProviderSet.

3.5.12.4 Add a provider

3.5.12.4.1 A method 'addProvider' shall be defined in order to add a MALProvider to this MALProviderSet.

3.5.12.4.2 The signature of the method 'addProvider' shall be:

```
void addProvider(const shared_ptr<MALProvider>& provider)
```

3.5.12.4.3 The parameter of the method 'addProvider' shall be assigned as described in table 3-115.

Table 3-115: MALProviderSet 'addProvider' Parameter

Parameter	Description
provider	MALProvider to be added

3.5.12.4.4 If the added MALProvider provides a MALService which is not the same as the MALService of this MALProviderSet, then a MALException shall be raised.

3.5.12.4.5 For every registered MALPublisherSet, one MALPublisher shall be created by calling the method 'createPublisher' provided by the MALPublisherSet.

3.5.12.5 Remove a provider

3.5.12.5.1 A method ‘removeProvider’ shall be defined in order to remove a MALProvider from this MALProviderSet.

3.5.12.5.2 The signature of the method ‘removeProvider’ shall be:

```
bool removeProvider(const shared_ptr<MALProvider>& provider)
```

3.5.12.5.3 The parameter of the method ‘removeProvider’ shall be assigned as described in table 3-116.

Table 3-116: MALProviderSet ‘removeProvider’ Parameter

Parameter	Description
provider	MALProvider to be removed

3.5.12.5.4 If this MALProviderSet contains the specified provider, then the method ‘removeProvider’ shall return TRUE; otherwise it shall return FALSE.

3.5.12.5.5 For every registered MALPublisherSet, the MALPublisher created through the removed MALProvider shall be deleted by calling the method ‘deletePublisher’ provided by the MALPublisherSet.

3.5.13 MALPUBLISHERSET

3.5.13.1 Definition

A MALPublisherSet class shall be defined in order to manage a set of MALPublishers that publish updates:

a) through the same PUBLISH-SUBSCRIBE operation;

b) in the same domain, network zone and session;

c) with the same QoS level and priority.

3.5.13.2 Creation

3.5.13.2.1 A MALPublisherSet constructor shall be defined.

3.5.13.2.2 The MALPublisherSet constructor signature shall be:

```
MALPublisherSet(
  sonst shared_ptr<MALProviderSet>& providerSet,
  const shared_ptr<MALPubSubOperation>& op,
  const IdentifierList& domain,
```

```
const Identifier& networkZone,
const SessionType& sessionType,
const Identifier& sessionName,
const QoSLevel& remotePublisherQos,
const MALQoSProperties& remotePublisherQosProps,
const uint32_t& remotePublisherPriority)
```

3.5.13.2.3 The MALPublisherSet constructor parameters shall be assigned as described in table 3-117.

Table 3-117: MALPublisherSet Constructor Parameters

Parameter	Description
providerSet	MALProviderSet that owns this MALPublisherSet
op	PUBLISH-SUBSCRIBE operation
domain	Domain of the PUBLISH messages
networkZone	Network zone of the PUBLISH messages
sessionType	Session type of the PUBLISH messages
sessionName	Session name of the PUBLISH messages
remotePublisherQos	QoS level of the PUBLISH messages
remotePublisherQosProps	QoS properties of the PUBLISH messages
remotePublisherPriority	Priority of the PUBLISH messages

3.5.13.2.4 The parameters ‘remotePublisherQos’, ‘remotePublisherQosProps’ and ‘remotePublisherPriority’ may be NULL.

3.5.13.3 Create a MALPublisher

3.5.13.3.1 A method ‘createPublisher’ shall be defined in order to create a MALPublisher and add it to this MALPublisherSet.

3.5.13.3.2 The signature of the method ‘createPublisher’ shall be:

```
void createPublisher(const shared_ptr<MALProvider>& provider)
```

3.5.13.3.3 The parameter of the method ‘createPublisher’ shall be assigned as described in table 3-118.

Table 3-118: MALPublisherSet 'createPublisher' Parameter

Parameter	Description
provider	MALProvider to be used in order to create the MALPublisher

3.5.13.3.4 If a MALException occurs, then it shall be raised again.

3.5.13.4 Delete a MALPublisher

3.5.13.4.1 A method 'deletePublisher' shall be defined in order to close a MALPublisher and remove it from this MALPublisherSet.

3.5.13.4.2 The signature of the method 'deletePublisher' shall be:

```
void deletePublisher(const shared_ptr<MALProvider>& provider)
```

3.5.13.4.3 The parameter of the method 'deletePublisher' shall be assigned as described in table 3-119.

Table 3-119: MALPublisherSet 'deletePublisher' Parameter

Parameter	Description
provider	MALProvider that owns the MALPublisher to remove

3.5.13.4.4 If no MALPublisher has been created with the specified MALProvider, then the method 'deletePublisher' shall return void; i.e., no MALException shall be raised.

3.5.13.4.5 If a MALPublisher has been created with the specified MALProvider, then the MALPublisher shall be closed and removed from this MALPublisherSet.

3.5.13.4.6 If a MALException occurs, then it shall be raised again.

3.5.13.5 Publish Updates

3.5.13.5.1 A method 'publish' shall be defined in order to publish updates through all the MALPublishers of this MALPublisherSet.

3.5.13.5.2 The signature of the method 'publish' shall be:

```
shared_ptr<MALMessage> publish(
                  const shared_ptr<UpdateHeaderList>& updateHeaderList,
                  const vector<UpdateList>& updateLists)
```

3.5.13.5.3 The parameters of the method ‘publish’ shall be assigned as described in table 3-120.

Table 3-120: MALPublisherSet ‘publish’ Parameters

Parameter	Description
updateHeaderList	Published UpdateHeaders
updateLists	Lists of updates to be published

3.5.13.5.4 Each MALPublisher shall be sequentially invoked through the method ‘publish’.

3.5.13.5.5 If a MALException or MALInteractionException occurs, then it shall be immediately raised again by this method.

3.5.13.6 Register

3.5.13.6.1 A method ‘syncRegister’ shall be defined in order to synchronously register through all the MALPublishers of this MALPublisherSet.

3.5.13.6.2 The signature of the method ‘register’ shall be:

```
void syncRegister(
            const shared_ptr<EntityKeyList>& entityKeys,
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.13.6.3 The parameters of the method ‘register’ shall be assigned as described in table 3-121.

Table 3-121: MALPublisherSet ‘register’ Parameters

Parameter	Description
entityKeyList	Keys of the entities that are to be published
listener	Listener in charge of receiving the messages PUBLISH ERROR

3.5.13.6.4 Each MALPublisher shall be sequentially invoked through the method ‘register’.

3.5.13.6.5 If a MALException or MALInteractionException occurs, then it shall be immediately raised again by this method.

3.5.13.7 Deregister

3.5.13.7.1 A method ‘deregister’ shall be defined in order to synchronously deregister through all the MALPublishers of this MALPublisherSet.

3.5.13.7.2 The signature of the method ‘deregister’ shall be:

```
void deregister()
```

3.5.13.7.3 Each MALPublisher shall be sequentially invoked through the method ‘deregister’.

3.5.13.7.4 If a MALException or MALInteractionException occurs, then it shall be immediately raised again by this method.

3.5.13.8 Asynchronous Register

3.5.13.8.1 A method ‘asyncRegister’ shall be defined in order to asynchronously register through all the MALPublishers of this MALPublisherSet.

3.5.13.8.2 The signature of the method ‘asyncRegister’ shall be:

```
shared_ptr<MALMessage> asyncRegister(
            const shared_ptr<EntityKeyList>& entityKeys,
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.13.8.3 The parameters of the method ‘asyncRegister’ shall be assigned as described in table 3-122.

Table 3-122: MALPublisherSet ‘asyncRegister’ Parameters

Parameter	Description
entityKeyList	Keys of the entities that are to be published
listener	Listener in charge of receiving the messages PUBLISH REGISTER ACK, PUBLISH REGISTER ERROR and PUBLISH ERROR

3.5.13.8.4 Each MALPublisher shall be sequentially invoked through the method ‘asyncRegister’.

3.5.13.8.5 If a MALException or MALInteractionException occurs, then it shall be immediately raised again by this method.

3.5.13.9 Asynchronous Deregister

3.5.13.9.1 A method ‘asyncDeregister’ shall be defined in order to asynchronously deregister through all the MALPublishers of this MALPublisherSet.

3.5.13.9.2 The signature of the method ‘asyncDeregister’ shall be:

```
shared_ptr<MALMessage> asyncDeregister(
            const shared_ptr<MALPublishInteractionListener>& listener)
```

3.5.13.9.3 The parameter of the method ‘asyncDeregister’ shall be assigned as described in table 3-123.

Table 3-123: MALPublisherSet ‘asyncDeregister’ Parameter

Parameter	Description
listener	Listener in charge of receiving the messages PUBLISH DEREGISTER ACK

3.5.13.9.4 Each MALPublisher shall be sequentially invoked through the method ‘asyncDeregister’.

3.5.13.9.5 If a MALException or MALInteractionException occurs, then it shall be immediately raised again by this method.

3.5.13.10 Close

3.5.13.10.1 A method ‘close’ shall be defined in order to close all the MALPublishers of this MALPublisherSet.

3.5.13.10.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.5.13.10.3 Each MALPublisher shall be sequentially invoked through the method ‘close’.

3.5.13.10.4 If a MALException occurs, then it shall be immediately raised again by this method.

3.6 BROKER NAMESPACE

3.6.1 OVERVIEW

This part of the API is dedicated to the MAL clients handling PUBLISH-SUBSCRIBE interactions as brokers. The broker state diagrams are specified in reference [1].

The classes and interfaces belong to the C++ namespace:

```
mo::mal::broker
```

3.6.2 MALBROKERMANAGER

3.6.2.1 Definition

3.6.2.1.1 A MALBrokerManager interface shall be defined in order to encapsulate the resources used to enable MAL brokers to handle PUBLISH-SUBSCRIBE interactions.

3.6.2.1.2 A MALBrokerManager shall be a MALBroker factory.

3.6.2.2 MALBrokerManager Creation

3.6.2.2.1 A MALBrokerManager shall be created by calling the method ‘createBrokerManager’ provided by MALContext.

3.6.2.2.2 Several MALBrokerManager instances should be created in order to separate the resources used by some brokers.

3.6.2.3 Create a MALBroker

3.6.2.3.1 Two methods ‘createBroker’ shall be defined in order to create a shared MAL level broker:

a) using no parameter;

b) using a MALBrokerHandler parameter.

3.6.2.3.2 The signatures of the method ‘createBroker’ shall be:

```
shared_ptr<MALBroker> createBroker()

shared_ptr<MALBroker> createBroker(
            const shared_ptr<MALBrokerHandler>& handler)
```

3.6.2.3.3 The parameter of the method ‘createBroker’ shall be assigned as described in table 3-124.

Table 3-124: MALBrokerManager 'createBroker' Parameter

Parameter	Description
handler	Broker interaction handler

3.6.2.3.4 The method 'createBroker' shall not return the value NULL.

3.6.2.3.5 If an error occurs, then a MALException shall be raised.

3.6.2.3.6 If the MALBrokerManager is closed, then a MALException shall be raised.

3.6.2.4 Create a MALBrokerBinding

3.6.2.4.1 Two methods 'createBrokerBinding' shall be defined in order to bind a shared MAL level broker to a particular transport:

a) using a private MALEndpoint;

b) using a shared MALEndpoint.

3.6.2.4.2 The signatures of the method 'createBrokerBinding' shall be:

```
shared_ptr<MALBrokerBinding> createBrokerBinding(
      const shared_ptr<MALBroker>& optionalMALBroker,
      const string& localName,
      const string& protocol,
      const shared_ptr<Blob>& authenticationId,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& qosProperties)

shared_ptr<MALBrokerBinding> createBrokerBinding(
      const shared_ptr<MALBroker>& optionalMALBroker,
      const shared_ptr<MALEndpoint>& endpoint,
      const shared_ptr<Blob>& authenticationId,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& qosProperties)
```

3.6.2.4.3 The parameters of the method 'createBrokerBinding' shall be assigned as described in table 3-125.

Table 3-125: MALBrokerBinding 'createBrokerBinding' Parameters

Parameter	Description
optionalMALBroker	MAL level broker to be bound
localName	Name of the private MALEndpoint to be created and used by the broker binding
protocol	Name of the protocol used to bind the broker
endpoint	Shared MALEndpoint to be used by the broker
authenticationId	Authentication identifier that should be used by the broker
expectedQos	QoS levels the broker assumes it can rely on
priorityLevelNumber	Number of priorities the broker uses
qosProperties	Default QoS properties used by the broker to send messages

3.6.2.4.4 If the parameter 'optionalMALBroker' is NULL, then a transport level broker shall be created.

3.6.2.4.5 The parameter 'localName' may be NULL.

3.6.2.4.6 If the parameter 'localName' is not NULL, then its value shall be unique for the MALContext instance and the protocol specified by the parameter 'protocol'.

3.6.2.4.7 The parameter 'authenticationId' shall only give a hint: some transports may not use this authentication identifier and assign another authentication identifier to the broker.

3.6.2.4.8 The parameter 'qosProperties' may be NULL.

3.6.2.4.9 If the MALBrokerManager is closed, then a MALException shall be raised.

3.6.2.4.10 If the broker binding local name is not NULL and if the broker process starts again after a stop and creates the MALBrokerBinding with the same local name (directly or through a shared MALEndpoint) then the MALBrokerBinding shall recover the same URI as before the stop.

3.6.2.5 Close

3.6.2.5.1 A method 'close' shall be defined in order to release the resources owned by this MALBrokerManager.

3.6.2.5.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.6.2.5.3 The method ‘close’ shall close all the MALBrokerBindings owned by this MALBrokerManager.

3.6.2.5.4 The method ‘close’ shall return after all the MALBrokerBindings have been closed.

3.6.2.5.5 If an internal error occurs, then a MALException shall be raised.

3.6.3 MALBROKER

3.6.3.1 Definition

A MALBroker interface shall be defined in order to represent a shared MAL level broker, i.e., a shared broker implemented at the MAL level.

NOTE – A single MALBroker instance can be bound to one or several transport layers. In this way, the MALBroker can act as a bridge between several transport layers. For instance, a MALBroker can bridge two transport layers by receiving updates from publishers through the first transport layer and notifying those updates to subscribers through the second transport layer.

3.6.3.2 Get Bindings

3.6.3.2.1 A method ‘getBindings’ shall be defined in order to return the MALBrokerBindings owned by this MALBroker.

3.6.3.2.2 The signature of the method ‘getBindings’ shall be:

```
vector<shared_ptr<MALBrokerBinding>> getBindings()
```

3.6.3.3 Close

3.6.3.3.1 A method ‘close’ shall be defined in order to terminate all pending interactions.

3.6.3.3.2 The signature of the method ‘close’ shall be:

```
void close()
```

3.6.3.3.3 The method ‘close’ shall call the method ‘close’ provided by all the MALBrokerBinding owned by this MALBroker.

3.6.3.3.4 If an internal error occurs, then a MALException shall be raised.

3.6.4 MALBROKERBINDING

3.6.4.1 Definition

A MALBrokerBinding interface shall be defined in order to represent:

a) a binding of a shared MAL level broker to a transport layer;

b) a transport level broker.

3.6.4.2 Get the URI

3.6.4.2.1 A method ‘getURI’ shall be defined in order to return the URI of the broker binding.

3.6.4.2.2 The signature of the method ‘getURI’ shall be:

```
URI getURI()
```

3.6.4.3 Get the Authentication Identifier

3.6.4.3.1 A method ‘getAuthenticationId’ shall be defined in order to return the authentication identifier of the broker.

3.6.4.3.2 The signature of the method ‘getAuthenticationId’ shall be:

```
shared_ptr<Blob> getAuthenticationId()
```

3.6.4.3.3 The method ‘getAuthenticationId’ shall return the actual authentication identifier.

NOTE – The method ‘getAuthenticationId’ may not return the same value as the one assigned when creating the broker, that assignment being just a hint.

3.6.4.4 Send a NOTIFY

3.6.4.4.1 A method ‘sendNotify’ shall be defined in order to enable a MALBrokerHandler to send a NOTIFY message to a subscriber.

3.6.4.4.2 The signature of the method ‘sendNotify’ shall be:

```
shared_ptr<MALMessage> sendNotify(
      const shared_ptr<MALOperation>& operation,
      const URI& subscriber,
      const int64_t& transactionId,
      const IdentifierList& domainId,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
      const QoSLevel& notifyQos,
```

```
const MALQoSProperties& notifyQosProps,
const uint32_t& notifyPriority,
const Identifier& subscriptionId,
const shared_ptr<UpdateHeaderList>& updateHeaderList,
const vector<UpdateList>& updateLists)
```

3.6.4.4.3 The parameters of the method ‘sendNotify’ shall be assigned as described in table 3-126.

Table 3-126: MALBrokerBinding ‘sendNotify’ Parameters

Parameter	Description
operation	Operation of the NOTIFY message
subscriber	Subscriber’s URI
transactionId	Transaction identifier of the NOTIFY message
domainId	Domain of the NOTIFY message
networkZone	Network zone of the NOTIFY message
sessionType	Session type of the NOTIFY message
sessionName	Session name of the NOTIFY message
notifyQos	QoS level of the NOTIFY message
notifyQosProps	QoS properties of the NOTIFY message
notifyPriority	Priority of the NOTIFY message
subscriptionId	Subscription identifier
updateHeaderList	List of update headers
updateLists	Lists of updates

3.6.4.4.4 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.6.4.4.5 A MALException shall be thrown if a non-MAL error occurs during the message sending.

3.6.4.4.6 If this MALBrokerBinding represents a transport level broker or is linked to a MAL level broker without MALBrokerHandler, then a MALException shall be raised.

3.6.4.4.7 If the MALBrokerBinding is closed, then a MALException shall be raised.

3.6.4.4.8 The method ‘sendNotify’ shall return the MALMessage that has been sent.

3.6.4.4.9 The allowed update list types shall be:

a) a std::vector<MAL element> updates list;

b) a std::vectorMALEncodedElement> containing the encoded updates;

c) a std::vector defined by a specific C++ mapping extension.

3.6.4.5 Send a NOTIFY ERROR

3.6.4.5.1 A method 'sendNotifyError' shall be defined in order to enable a MALBrokerHandler to send a NOTIFY ERROR message to a subscriber.

3.6.4.5.2 The signature of the method 'sendNotifyError' shall be:

```
shared_ptr<MALMessage> sendNotifyError(
      const shared_ptr<MALOperation>& operation,
      const URI& subscriber,
      const in64_t& transactionId,
      const IdentifierList& domainId,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
      const QoSLevel& notifyQos,
      const MALQoSProperties& notifyQosProps,
      const uint32_t& notifyPriority,
      const shared_ptr<MALStandardError>& error)
```

3.6.4.5.3 The parameters of the method 'sendNotifyError' shall be assigned as described in table 3-127.

Table 3-127: MALBrokerBinding 'sendNotifyError' Parameters

Parameter	Description
operation	Operation of the NOTIFY ERROR message
subscriber	Subscriber's URI
transactionId	Transaction identifier of the NOTIFY ERROR message
domainId	Domain of the NOTIFY ERROR message
networkZone	Network zone of the NOTIFY ERROR message
sessionType	Session type of the NOTIFY ERROR message
sessionName	Session name of the NOTIFY ERROR message
notifyQos	QoS level of the NOTIFY ERROR message
notifyQosProps	QoS properties of the NOTIFY ERROR message
notifyPriority	Priority of the NOTIFY ERROR message
error	Body of the NOTIFY ERROR message

3.6.4.5.4 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.6.4.5.5 A MALException shall be thrown if a non-MAL error occurs during the message sending.

3.6.4.5.6 If this MALBrokerBinding represents a transport level broker or is linked to a MAL level broker without MALBrokerHandler, then a MALException shall be raised.

3.6.4.5.7 If the MALBrokerBinding is closed, then a MALException shall be raised.

3.6.4.5.8 The method ‘sendNotifyError’ shall return the MALMessage that has been sent.

3.6.4.6 Send a PUBLISH ERROR

3.6.4.6.1 A method ‘sendPublishError’ shall be defined in order to enable a MALBrokerHandler to send a PUBLISH ERROR message to a publisher.

3.6.4.6.2 The signature of the method ‘sendPublishError’ shall be:

```
shared_ptr<MALMessage> sendPublishError(
      const shared_ptr<MALOperation>& operation,
      const URI& publisher,
      const int64_t& transactionId,
      const IdentifierList& domainId,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
      const QoSLevel& qos,
      const MALQoSProperties& qosProps,
      const uint32_t& priority,
      const shared_ptr<MALStandardError>& error)
```

3.6.4.6.3 The parameters of the method ‘sendPublishError’ shall be assigned as described in table 3-128.

Table 3-128: MALBrokerBinding 'sendPublishError' Parameters

Parameter	Description
operation	Operation of the PUBLISH ERROR message
publisher	Publisher's URI
transactionId	Transaction identifier of the PUBLISH ERROR message
domainId	Domain of the PUBLISH ERROR message
networkZone	Network zone of the PUBLISH ERROR message
sessionType	Session type of the PUBLISH ERROR message
sessionName	Session name of the PUBLISH ERROR message
qos	QoS level of the PUBLISH ERROR message
notifyQoSProps	QoS properties of the PUBLISH ERROR message
notifyPriority	Priority of the PUBLISH ERROR message
error	Body of the PUBLISH ERROR message

3.6.4.6.4 A MALInteractionException shall be thrown if a MAL standard error occurs during the initiation message sending.

3.6.4.6.5 A MALException shall be thrown if a non-MAL error occurs during the message sending.

3.6.4.6.6 If this MALBrokerBinding represents a transport level broker or a MAL level broker without MALBrokerHandler, then a MALException shall be raised.

3.6.4.6.7 If the MALBrokerBinding is closed, then a MALException shall be raised.

3.6.4.6.8 The method 'sendPublishError' shall return the MALMessage that has been sent.

3.6.4.7 Set the Transmit Error Listener

3.6.4.7.1 A method 'setTransmitErrorListener' shall be defined in order to set a MALTransmitErrorListener.

3.6.4.7.2 The signature of the method 'setTransmitErrorListener' shall be:

```
void setTransmitErrorListener(
            const shared_ptr<MALTransmitErrorListener>& listener)
```

3.6.4.7.3 The parameter of the method 'setTransmitErrorListener' shall be assigned as described in table 3-129.

Table 3-129: MALBrokerBinding 'setTransmitErrorListener' Parameter

Parameter	Description
listener	Listener in charge of receiving every asynchronous TRANSMIT ERROR that cannot be returned as a MAL message

3.6.4.7.4 The parameter 'listener' may be NULL.

3.6.4.7.5 The MALTransmitErrorListener shall be called when:

a) a TRANSMIT ERROR is asynchronously returned to the broker;

b) the TRANSMIT ERROR cannot be returned as a MAL message.

3.6.4.7.6 If the MALBrokerBinding is closed, then a MALException shall be raised.

3.6.4.7.7 If an internal error occurs, then a MALException shall be raised.

3.6.4.8 Get the Transmit Error Listener

3.6.4.8.1 A method 'getTransmitErrorListener' shall be defined in order to return the MALTransmitErrorListener.

3.6.4.8.2 The signature of the method 'getTransmitErrorListener' shall be:

```
shared_ptr<MALTransmitErrorListener> getTransmitErrorListener()
```

3.6.4.8.3 If no MALTransmitErrorListener has been set, then the method 'getTransmitErrorListener' shall return NULL.

3.6.4.8.4 If the MALBrokerBinding is closed, then a MALException shall be raised.

3.6.4.8.5 If an internal error occurs, then a MALException shall be raised.

3.6.4.9 Close

3.6.4.9.1 A method 'close' shall be defined in order to terminate all pending interactions and deactivate the broker binding.

3.6.4.9.2 The signature of the method 'close' shall be:

```
void close()
```

3.6.4.9.3 A close should be called by a MAL client before the broker process is stopped for any operational reason.

3.6.4.9.4 The method 'close' shall deactivate this MALBrokerBinding.

3.6.4.9.5 If the MALBroker is being activated at the time of the close, then the closing process shall wait for the end of the broker's execution.

3.6.4.9.6 Once a MALBrokerBinding has been closed, the message delivery shall be stopped in order that the broker will not receive any message through this MALBrokerBinding.

3.6.4.9.7 If an internal error occurs, then a MALException shall be raised.

3.6.4.9.8 Pending interactions shall be finalized by returning the error DESTINATION_LOST_OR_DIED to the subscribers or publishers.

3.6.4.9.9 If a subscriber or a publisher tries to interact with a closed broker binding and if the QoS level is not QUEUED, a DELIVERY_FAILED shall be returned to the subscriber or publisher.

3.6.4.9.10 If a subscriber or a publisher tries to interact with a closed broker binding and if the QoS level is QUEUED, then the request shall be queued and delivered as soon as the broker binding message delivery is started again.

3.6.4.9.11 If the broker binding owns a private MALEndpoint, then the MALEndpoint shall be closed.

3.6.5 MALBROKERHANDLER

3.6.5.1 Definition

3.6.5.1.1 A MALBrokerHandler interface shall be defined in order to enable to handle the interactions on the broker side.

3.6.5.1.2 The MALBrokerHandler interface shall handle the following PUBLISH-SUBSCRIBE interaction stages: REGISTER, PUBLISH REGISTER, PUBLISH, DEREGISTER and PUBLISH DEREGISTER.

3.6.5.2 Initialize

3.6.5.2.1 A method 'initialize' shall be defined in order to enable a MALBrokerHandler to be initialized when the broker is activated.

3.6.5.2.2 The signature of the method 'initialize' shall be:

```
void initialize(const shared_ptr<MALBrokerBinding>& brokerBinding)
```

3.6.5.2.3 The parameter of the method 'initialize' shall be assigned as described in table 3-130.

Table 3-130: MALBrokerHandler 'initialize' Parameter

Parameter	Description
brokerBinding	Created MALBrokerBinding

3.6.5.2.4 The method 'initialize' shall be called when the method 'createBrokerBinding' provided by the interface MALBrokerManager is called.

NOTE – The method 'initialize' enables the handler to store the reference of the MALBrokerBinding in order to send NOTIFY, NOTIFY ERROR and PUBLISH ERROR messages.

3.6.5.2.5 If an instance of MALBrokerHandler is used by several MALBrokerBinding, then the method 'initialize' shall be called once for each MALBrokerBinding.

3.6.5.3 Handle the REGISTER stage

3.6.5.3.1 A method 'handleRegister' shall be defined in order to handle the REGISTER stage of a PUBLISH-SUBSCRIBE interaction.

3.6.5.3.2 The signature of the method 'handleRegister' shall be:

```
void handleRegister(
            const shared_ptr<MALInteraction>& interaction,
            const shared_ptr<MALRegisterBody>& body)
```

3.6.5.3.3 The parameters of the method 'handleRegister' shall be assigned as described in table 3-131.

Table 3-131: MALBrokerHandler 'handleRegister' Parameters

Parameter	Description
interaction	Interaction context
body	Body of the REGISTER message to handle

3.6.5.3.4 The parameters 'interaction' and 'body' shall not be NULL.

3.6.5.3.5 A REGISTER ACK message shall be sent to the subscriber when the method 'handleRegister' returns.

3.6.5.3.6 If a MAL standard error occurs, then a MALInteractionException shall be raised.

3.6.5.3.7 If a non-MAL error occurs, then a MALException shall be raised.

3.6.5.3.8 If a MALInteractionException is raised, then a REGISTER ERROR message shall be sent to the subscriber.

3.6.5.4 Handle the PUBLISH REGISTER stage

3.6.5.4.1 A method 'handlePublishRegister' shall be defined in order to handle the PUBLISH REGISTER stage of a PUBLISH-SUBSCRIBE interaction.

3.6.5.4.2 The signature of the method 'handlePublishRegister' shall be:

```
void handlePublishRegister(
            const shared_ptr<MALInteraction>& interaction,
            const shared_ptr<MALPublishRegisterBody>& body)
```

3.6.5.4.3 The parameters of the method 'handlePublishRegister' shall be assigned as described in table 3-132.

Table 3-132: MALBrokerHandler 'handlePublishRegister' Parameters

Parameter	Description
interaction	Interaction context
body	Body of the PUBLISH REGISTER message to handle

3.6.5.4.4 The parameters 'interaction' and 'body' shall not be NULL.

3.6.5.4.5 A PUBLISH REGISTER ACK message shall be sent to the subscriber when the method 'handlePublishRegister' returns.

3.6.5.4.6 If a MAL standard error occurs, then a MALInteractionException shall be raised.

3.6.5.4.7 If a non-MAL error occurs, then a MALException shall be raised.

3.6.5.4.8 If a MALInteractionException is raised, then a PUBLISH REGISTER ERROR message shall be sent to the subscriber.

3.6.5.5 Handle the PUBLISH stage

3.6.5.5.1 A method 'handlePublish' shall be defined in order to handle the PUBLISH stage of a PUBLISH-SUBSCRIBE interaction.

3.6.5.5.2 The signature of the method 'handlePublish' shall be:

```
void handlePublish(
            const shared_ptr<MALInteraction>& interaction,
            const shared_ptr<MALPublishBody>& body)
```

3.6.5.5.3 The parameters of the method ‘handlePublish’ shall be assigned as described in table 3-133.

Table 3-133: MALBrokerHandler ‘handlePublish’ Parameters

Parameter	Description
interaction	Interaction context
body	Body of the PUBLISH message to handle

3.6.5.5.4 The parameters ‘interaction’ and ‘body’ shall not be NULL.

3.6.5.5.5 If a MAL standard error occurs, then a message PUBLISH ERROR shall be sent through the MALBrokerBinding which URI is equal to the field ‘URI to’ of the PUBLISH message.

3.6.5.5.6 If a non-MAL error occurs, then a MALException shall be raised.

3.6.5.6 Handle the DEREGISTER stage

3.6.5.6.1 A method ‘handleDeregister’ shall be defined in order to handle the DEREGISTER stage of a PUBLISH-SUBSCRIBE interaction.

3.6.5.6.2 The signature of the method ‘handleDeregister’ shall be:

```
void handleDeregister(
            const shared_ptr<MALInteraction>& interaction,
            const shared_ptr<MALDeregisterBody>& body)
```

3.6.5.6.3 The parameters of the method ‘handleDeregister’ shall be assigned as described in table 3-134.

Table 3-134: MALBrokerHandler ‘handleDeregister’ Parameters

Parameter	Description
interaction	Interaction context
body	Body of the DEREGISTER message to handle

3.6.5.6.4 The parameters ‘interaction’ and ‘body’ shall not be NULL.

3.6.5.6.5 A DEREGISTER ACK message shall be sent to the subscriber when the method ‘handleDeregister’ returns.

3.6.5.6.6 If a non-MAL error occurs, then a MALException shall be raised.

3.6.5.7 Handle the PUBLISH DEREGISTER stage

3.6.5.7.1 A method ‘handlePublishDeregister’ shall be defined in order to handle the PUBLISH DEREGISTER stage of a PUBLISH-SUBSCRIBE interaction.

3.6.5.7.2 The signature of the method ‘handlePublishDeregister’ shall be:

```
public void handlePublishDeregister(
            const shared_ptr<MALInteraction>& interaction)
```

3.6.5.7.3 The parameters of the method ‘handlePublishDeregister’ shall be assigned as described in table 3-135.

Table 3-135: MALBrokerHandler ‘handlePublishDeregister’ Parameter

Parameter	Description
interaction	Interaction context

3.6.5.7.4 The parameter ‘interaction’ shall not be NULL.

3.6.5.7.5 A PUBLISH DEREGISTER ACK message shall be sent to the subscriber when the method ‘handlePublishDeregister’ returns.

3.6.5.7.6 If a non-MAL error occurs, then a MALException shall be raised.

3.6.5.8 Finalize

3.6.5.8.1 A method ‘finalize’ shall be defined in order to enable a MALBrokerHandler to be notified when a MALBrokerBinding is closed.

3.6.5.8.2 The signature of the method ‘malFinalize’ shall be:

```
void finalize(const shared_ptr<MALBrokerBinding>& brokerBinding)
```

3.6.5.8.3 The parameter of the method ‘finalize’ shall be assigned as described in table 3-136.

Table 3-136: MALBrokerHandler ‘malFinalize’ Parameter

Parameter	Description
brokerBinding	Closed MALBrokerBinding

3.6.5.8.4 The method ‘finalize’ shall be called when the method ‘close’ provided by the interface MALBrokerBinding is called.

4 SERVICE MAPPING

4.1 OVERVIEW

This section explains how to map a service specification (reference [1]) to C++. Two aspects are handled here:

a) the definition of the interfaces used by a consumer/provider to use/provide a service;

b) the way of implementing those interfaces.

This section uses C++ templates. The template variables are related to the MAL XML Schema; i.e., each variable is linked to an XPath enabling to unambiguously resolve the values to be assigned to this variable when applying the template.

4.2 DEFINITION

4.2.1 The values of the variables used in the C++ code templates shall be resolved according to table 4-1 and the following rules:

a) the default namespace shall be http://www.ccsds.org/schema/ServiceSchema;

b) the default current XPath context shall be the element ‘specification’;

c) each variable shall be assigned with the value specified by the XPath;

d) in the context of the parent of the element specified by the XPath, several values may be assigned to a variable;

e) if the last column mentions ‘Attribute value’ then the value of the specified XML attribute shall be assigned;

f) if the last column contains a value in inverted commas, then this value shall be assigned;

g) if the last column mentions ‘C++ class mapped from type’ then the name of the C++ class mapped from the specified XML element ‘type’ shall be assigned;

h) if the last column mentions ‘Type absolute short form’ then the value shall depend on the type specified by the XML element ‘type’ as follows:

   1) if the type is concrete, then the absolute short form shall be assigned;

   2) if the type is abstract, then the value NULL shall be assigned;

i) if the last column mentions ‘List type absolute short form’ then the value shall depend on the type specified by the XML element ‘type’ as follows:

   1) if the type is concrete, then the absolute short form of the type list shall be assigned;

2) if the type is abstract, then the value NULL shall be assigned;

j) if the last column mentions ‘Mapping extension’ then the class name ‘Element’ shall be assigned unless a specific C++ mapping extension is defined for the specified namespace <<extension>>;

k) if the last column mentions ‘Enumeration item index’ then the index of the enumeration item shall be assigned;

l) the index of enumeration items shall start from zero and be incremented by one in the same order as items are declared.

Table 4-1: Service Mapping Variables

Variable name	Current context	Element or attribute	Value
ack	area/service/ capabilitySet	invokeIP/ acknowledgement/ type	C++ class mapped from type
		progressIP/ acknowledgement/ type	C++ class mapped from type
		invokeIP/ acknowledgement/ <<extension>>:*	Mapping extension
		progressIP/ acknowledgement/ <<extension>>:*	Mapping extension
ack short form	area/service/ capabilitySet	invokeIP/ acknowledgement/ type	Type absolute short form
		progressIP/ acknowledgement/ type	Type absolute short form
area	.	area/@name	Attribute value
area number	.	area/@number	Attribute value
area version	.	area/@version	Attribute value
composite	.//dataType	composite/@name	Attribute value
composite parent class	.//dataType/ composite/extends	type	C++ class mapped from type
composite field name	.//dataType/ composite	field/@name	Attribute value
composite field class	.//dataType/ composite/field	type	C++ class mapped from type
enumeration	.//dataType	enumeration@name	Attribute value
enum item	.//dataType/ enumeration	item/@value	Attribute value
enum item index	.//dataType/ enumeration	item	Enumeration item index
enum item numeric value	.//dataType/ enumeration	item/@nvalue	Attribute value
error	.//errors	error/@name	Attribute value

Variable name	**Current context**	**Element or attribute**	**Value**
error number	.//errors	error/@number	Attribute value
in	area/service/ capabilitySet	sendIP/send/type	C++ class mapped from type
		submitIP/submit/type	C++ class mapped from type
		requestIP/request/type	C++ class mapped from type
		invokeIP/invoke/type	C++ class mapped from type
		progressIP/progress/type	C++ class mapped from type
		sendIP/send/ <<extension>>:*	Mapping extension
		submitIP/submit/ <<extension>>:*	Mapping extension
		requestIP/request/ <<extension>>:*	Mapping extension
		invokeIP/invoke/ <<extension>>:*	Mapping extension
		progressIP/progress/ <<extension>>:*	Mapping extension
in short form	area/service/ capabilitySet	sendIP/send/type	Type absolute short form
		submitIP/submit/type	Type absolute short form
		requestIP/request/type	Type absolute short form
		invokeIP/invoke/type	Type absolute short form
		progressIP/progress/type	Type absolute short form
list	.//dataType	list/@name	Attribute value
list element class	.//dataType/list	type	C++ class mapped from type
notify	area/service/ capabilitySet	pubsubIP/publishNotify/ type	C++ class mapped from type
notify list short form	area/service/ capabilitySet	pubsubIP/publishNotify/ type	List type absolute short form

Variable name	**Current context**	**Element or attribute**	**Value**
op	area/service/ capabilitySet	sendIP/@name	Attribute value
		submitIP/@name	Attribute value
		requestIP/@name	Attribute value
		invokeIP/@name	Attribute value
		progressIP/@name	Attribute value
		pubsubIP/@name	Attribute value
op ip	area/service/ capabilitySet	sendIP	'SEND'
		submitIP	'SUBMIT'
		requestIP	'REQUEST'
		invokeIP	'INVOKE'
		progressIP	'PROGRESS'
		pubsubIP	'PUBLISH-SUBSCRIBE'
op number	area/service/ capabilitySet	sendIP/@number	Attribute value
		submitIP/@number	Attribute value
		requestIP/@number	Attribute value
		invokeIP/@number	Attribute value
		progressIP/@number	Attribute value
		pubsubIP/@number	Attribute value
op replayable	area/service/ capabilitySet	sendIP/@supportInReplay	Attribute value
		submitIP/@supportInReplay	Attribute value
		requestIP/@supportInReplay	Attribute value
		invokeIP/@supportInReplay	Attribute value
		progressIP/@supportInReplay	Attribute value
		pubsubIP/@supportInReplay	Attribute value
res	area/service/ capabilitySet	requestIP/response/type	C++ class mapped from type
		invokeIP/response/type	C++ class mapped from type
		progressIP/response/type	C++ class mapped from type
		requestIP/response/ <<extension>>:*	Mapping extension
		invokeIP/response/ <<extension>>:*	Mapping extension

Variable name	Current context	Element or attribute	Value
		progressIP/response/ <<extension>>:*	Mapping extension
res	area/service/ capabilitySet	requestIP/response/type	Type absolute short form
		invokeIP/response/type	Type absolute short form
		progressIP/response/type	Type absolute short form
service	area	service/@name	Attribute value
service number	area	service/@number	Attribute value
short form	.//dataType	composite/@shortForm	Attribute value
		enumeration/@shortForm	Attribute value
		list/@shortForm	Attribute value
update	area/service/ capabilitySet	progressIP/update/type	C++ class mapped from type
		progressIP/update/ <<extension>>:*	Mapping extension
update short form	area/service/ capabilitySet	progressIP/update/type	Type absolute short form

4.2.2 A C++ namespace root name shall be defined for each service area.

NOTE – For example the namespace root name can be:

```
mo
```

4.2.3 The C++ namespace root name shall be unique in the C++ namespace.

4.2.4 The C++ namespace root name may contain an area version number in order to enable to launch several versions of the same service in a process.

4.2.5 The namespace root name shall be assigned to the code template variable ‘root name’.

4.2.6 The following namespaces shall be defined:

```
<<root name>>::<<!area!>>
<<root name>>::<<!area!>>::structures
<<root name>>::<<!area!>>::structures::factory
<<root name>>::<<!area!>>::<<!service!>>
<<root name>>::<<!area!>>::<<!service!>>::body
<<root name>>::<<!area!>>::<<!service!>>::consumer
<<root name>>::<<!area!>>::<<!service!>>::provider
<<root name>>::<<!area!>>::<<!service!>>::structures
<<root name>>::<<!area!>>::<<!service!>>::structures::factory
```

4.2.7 For each area, the classes indicated in table 4-2 shall be created.

Table 4-2: Area Classes

Namespace name	Content
<<root name>>::<<!area!>>	Area helper class
<<root name>>::<<!area!>>::structures	Data structures classes: enumerations, lists and other composites
<<root name>>::<<!area!>>::structures::factory	MALElementFactory classes

4.2.8 For each service, the classes and interfaces indicated in table 4-3 shall be created.

Table 4-3: Service Classes

Namespace name	Content
<<root name>>::<<!area!>>::<<!service!>>	Service helper class
<<root name>>::<<!area!>>::<<!service!>>::consumer	Stub interface Adapter class Stub implementation class
<<root name>>::<<!area!>>::<<!service!>>::provider	Handler interface Skeleton interface Publisher classes Interaction classes Skeleton implementation classes (inheritance and delegation models)
<<root name>>::<<!area!>>::<<!service!>>::structures	Data structures classes: enumerations, lists and other composite types
<<root name>>::<<!area!>>::<<!service!>>::structures::factory	MALElementFactory classes
<<root name>>::<<!area!>>::<<!service!>>::body	Multiple element body classes

4.3 CONSUMER

4.3.1 GENERAL

Figure 4-1 is a class diagram that describes the relationships between the stub classes and interfaces at the consumer side. The following classes or interfaces are defined:

a) <<Service>> is the stub interface;

b) <<Service>>Stub is the stub implementation;

c) <<Service>>Adapter is the specific adapter class.

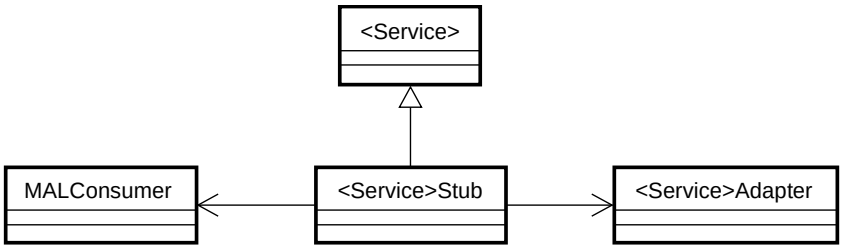

Figure 4-1: Relationships between the Stub Classes and Interfaces

4.3.2 STUB INTERFACE

4.3.2.1 Definition

4.3.2.1.1 A Stub interface shall be defined in order to initiate interaction patterns through service specific methods.

4.3.2.1.2 The name of the interface shall be: <<Service>>.

4.3.2.2 Consumer Getter

4.3.2.2.1 A method ‘getConsumer’ shall be defined in order to get the reference of the MALConsumer.

4.3.2.2.2 The signature of the method ‘getConsumer’ shall be:

```
shared_ptr<MALConsumer> getConsumer()
```

4.3.2.3 SEND Operation Invocation

4.3.2.3.1 A method ‘<<op>>’ shall be defined for each SEND operation provided by the service.

4.3.2.3.2 The signature of the method ‘<<op>>’ shall be:

```
shared_ptr<MALMessage> <<op>>(<<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>)
```

4.3.2.4 SUBMIT Operation Synchronous Invocation

4.3.2.4.1 A method '<<op>>' shall be defined for each SUBMIT operation provided by the service.

4.3.2.4.2 The signature of the method '<<op>>' shall be:

```
void <<op>>(<<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>)
```

4.3.2.5 REQUEST Operation Synchronous Invocation

4.3.2.5.1 A method '<<op>>' shall be defined for each REQUEST operation provided by the service.

4.3.2.5.2 If the RESPONSE message body is empty, then the signature of the method '<<op>>' shall be:

```
void <<op>>(<<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>)
```

4.3.2.5.3 If the RESPONSE message body contains one element, then the signature of the method '<<op>>' shall be:

```
<<Res [1]>> <<op>>(<<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>)
```

4.3.2.5.4 If the RESPONSE message body contains more than one element, then the signature of the method '<<op>>' shall be:

```
<<Op>>Response <<op>>(<<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>)
```

4.3.2.6 INVOKE Operation Synchronous Invocation

4.3.2.6.1 A method '<<op>>' shall be defined for each INVOKE operation provided by the service.

4.3.2.6.2 If the ACK message body is empty, then the signature of the method '<<op>>' shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.6.3 If the ACK message body contains one element, then the signature of the method '<<op>>' shall be:

```
<<Ack [1]>> <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
```

```
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.6.4 If the ACK message body contains more than one element, then the signature of the method '<<op>>' shall be:

```
<<Op>>Ack <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.7 PROGRESS Operation Synchronous Invocation

4.3.2.7.1 A method '<<op>>' shall be defined for each PROGRESS operation provided by the service.

4.3.2.7.2 If the ACK message body is empty, then the signature of the method '<<op>>' shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.7.3 If the ACK message body contains one element, then the signature of the method '<<op>>' shall be:

```
public <<Ack [1]>> <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.7.4 If the ACK message body contains more than one element, then the signature of the method '<<op>>' shall be:

```
public <<Op>>Ack <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.8 PUBLISH-SUBSCRIBE Operation REGISTER Synchronous Invocation

4.3.2.8.1 A method '<<op>>Register' shall be defined for each PUBLISH-SUBSCRIBE operation provided by the service.

4.3.2.8.2 The signature of the method '<<op>>Register' shall be:

```
void <<op>>Register(
            const shared_ptr<Subscription>& subscription,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.9 PUBLISH-SUBSCRIBE Operation DEREGISTER Synchronous Invocation

4.3.2.9.1 A method ‘<<op>>Deregister’ shall be defined for each PUBLISH-SUBSCRIBE operation provided by the service.

4.3.2.9.2 The signature of the method ‘<<op>>Deregister’ shall be:

```
void <<op>>Deregister(const IdentifierList& subscriptionIdList)
```

4.3.2.10 SUBMIT Operation Asynchronous Invocation

4.3.2.10.1 A method ‘async<<Op>>’ shall be defined for each SUBMIT operation provided by the service.

4.3.2.10.2 The signature of the method ‘async<<Op>>’ shall be:

```
shared_ptr<MALMessage> async<<Op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.11 REQUEST Operation Asynchronous Invocation

4.3.2.11.1 A method ‘async<<Op>>’ shall be defined for each REQUEST operation provided by the service.

4.3.2.11.2 The signature of the method ‘async<<Op>>’ shall be:

```
shared_ptr<MALMessage> async<<Op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.12 INVOKE Operation Asynchronous Invocation

4.3.2.12.1 A method ‘async<<Op>>’ shall be defined for each INVOKE operation provided by the service.

4.3.2.12.2 The signature of the method ‘async<<Op>>’ shall be:

```
shared_ptr<MALMessage> async<<Op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.13 PROGRESS Operation Asynchronous Invocation

4.3.2.13.1 A method ‘async<<Op>>’ shall be defined for each PROGRESS operation provided by the service.

4.3.2.13.2 The signature of the method ‘async<<Op>>’ shall be:

```
shared_ptr<MALMessage> async<<Op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.14 PUBLISH-SUBSCRIBE Operation REGISTER Asynchronous Invocation

4.3.2.14.1 A method ‘async<<Op>>Register’ shall be defined for each PUBLISH-SUBSCRIBE operation provided by the service.

4.3.2.14.2 The signature of the ‘method async<<Op>>Register’ shall be:

```
shared_ptr<MALMessage> async<<Op>>Register(
            const shared_ptr<Subscription>& subscription,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.15 PUBLISH-SUBSCRIBE Operation DEREGISTER Asynchronous Invocation

4.3.2.15.1 A method ‘async<<Op>>Deregister’ shall be defined for each PUBLISH-SUBSCRIBE operation provided by the service.

4.3.2.15.2 The signature of the ‘method async<<Op>>Deregister’ shall be:

```
shared_ptr<MALMessage> async<<Op>>Deregister(
                const IdentifierList& subscriptionIdList,
                const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.2.16 Continue an Interaction

4.3.2.16.1 A method ‘continue<<Op>>’ shall be defined for each SUBMIT, REQUEST, INVOKE and PROGRESS operation provided by the service.

4.3.2.16.2 The signature of the method ‘continue<<Op>>’ shall be:

```
public void continue<<Op>>(
            const uint8_t& lastInteractionStage,
            const Time& initiationTimestamp,
            const int64_t& transactionId,
            const shared_ptr<<<Service>>>Adapter& adapter)
```

4.3.3 STUB ADAPTER CLASS

4.3.3.1 Definition

4.3.3.1.1 A Stub adapter class shall be defined in order to specialize the MALInteractionAdapter methods according to the specific operations provided by the service.

4.3.3.1.2 The name of the class shall be: <<Service>>Adapter.

4.3.3.1.3 The Stub adapter class shall be abstract.

4.3.3.1.4 The Stub adapter class shall extend the class MALInteractionAdapter and define one specific adapter method for each operation and message type.

4.3.3.1.5 The Stub adapter class shall redefine each method provided by MALInteractionAdapter in order to call the specific adapter method according to the operation.

4.3.3.1.6 When the parameter 'body' is typed MALMessageBody, the body elements shall be extracted by calling the method 'getBodyElement'.

4.3.3.1.7 When the parameter 'body' is typed MALErrorBody, the body element shall be the MALStandardError extracted by calling the method 'getError'.

4.3.3.1.8 When getting a body element from a MALMessageBody, the Stub adapter class shall pass an instance of the element if the type is concrete, i.e., if there is no polymorphism; otherwise the Stub adapter class shall pass the value NULL.

4.3.3.1.9 The Stub adapter class shall convert the body elements typed Union into their C++ type when calling the specific adapter method.

4.3.3.1.10 If the operation specified by the header is not defined by the service, then a MALException shall be raised except if the IP is PUBLISH-SUBSCRIBE and the stage is NOTIFY.

4.3.3.2 SUBMIT ACK Adapter Method

The method 'submitAckReceived' inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method '<<op>>AckReceived' with the parameters 'header' and 'qosProperties'.

4.3.3.3 SUBMIT ERROR Adapter Method

The method 'submitErrorReceived' inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method '<<op>>ErrorReceived' with the parameters 'header', the MALStandardError, and 'qosProperties'.

4.3.3.4 REQUEST RESPONSE Adapter Method

The method 'requestResponseReceived' inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method '<<op>>ResponseReceived' with the parameters 'header', the body elements, and 'qosProperties'.

4.3.3.5 REQUEST ERROR Adapter Method

The method ‘requestErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>ErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.6 INVOKE ACK Adapter Method

The method ‘invokeAckReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>AckReceived’ with the parameters ‘header’, the body elements, and ‘qosProperties’.

4.3.3.7 INVOKE ACK ERROR Adapter Method

The method ‘invokeAckErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>AckErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.8 INVOKE RESPONSE Adapter Method

The method ‘invokeResponseReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>ResponseReceived’ with the parameters ‘header’, the body elements, and ‘qosProperties’.

4.3.3.9 INVOKE RESPONSE ERROR Adapter Method

The method ‘invokeResponseErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>ResponseErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.10 PROGRESS ACK Adapter Method

The method ‘progressAckReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>AckReceived’ with the parameters ‘header’, the body elements, and ‘qosProperties’.

4.3.3.11 PROGRESS ACK ERROR Adapter Method

The method ‘progressAckErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>AckErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.12 PROGRESS UPDATE Adapter Method

The method ‘progressUpdateReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>UpdateReceived’ with the parameters ‘header’, the body elements, and ‘qosProperties’.

4.3.3.13 PROGRESS UPDATE ERROR Adapter Method

The method ‘progressUpdateErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>UpdateErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.14 PROGRESS RESPONSE Adapter Method

The method ‘progressResponseReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>ResponseReceived’ with the parameters ‘header’, the body elements, and ‘qosProperties’.

4.3.3.15 PROGRESS RESPONSE ERROR Adapter Method

The method ‘progressResponseErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>ResponseErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.16 REGISTER ACK Adapter Method

The method ‘registerAckReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>RegisterAckReceived’ with the parameters ‘header’ and ‘qosProperties’.

4.3.3.17 REGISTER ERROR Adapter Method

The method ‘registerErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>RegisterErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.18 NOTIFY Adapter Method

4.3.3.18.1 The method ‘notifyReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>NotifyReceived’ with the parameters ‘header’, ‘subscriptionId’, ‘updateHeaderList’, the multiple update lists (<<Notify [i]>>List, ... <<Notify [N]>>List), and ‘qosProperties’.

4.3.3.18.2 If the area and service specified by the header are not the same as the area and service this adapter belongs to, then the method ‘notifyReceivedFromOtherService’ shall be called with parameters ‘header’, ‘body’, and ‘qosProperties’.

4.3.3.19 NOTIFY ERROR Adapter Method

The method ‘notifyErrorReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>NotifyErrorReceived’ with the parameters ‘header’, the MALStandardError, and ‘qosProperties’.

4.3.3.20 DEREGISTER ACK Adapter Method

The method ‘deregisterAckReceived’ inherited from MALInteractionAdapter shall be redefined by calling the specific adapter method ‘<<op>>DeregisterAckReceived’ with the parameters ‘header’ and ‘qosProperties’.

4.3.3.21 Specific SUBMIT Adapter Methods

4.3.3.21.1 For each SUBMIT operation two specific adapter methods shall be defined:

a) <<op>>AckReceived;

b) <<op>>ErrorReceived.

4.3.3.21.2 The signature of the method ‘<<op>>AcknowledgementReceived’ shall be:

```
void <<op>>AckReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

4.3.3.21.3 The signature of the method ‘<<op>>ErrorReceived’ shall be:

```
void <<op>>ErrorReceived(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& error,
            const MALQoSProperties& qosProperties)
```

4.3.3.22 Specific REQUEST Adapter Methods

4.3.3.22.1 For each REQUEST operation two specific adapter methods shall be defined:

a) <<op>>ResponseReceived;

b) <<op>>ErrorReceived.

4.3.3.22.2 The signature of the method '<<op>>ResponseReceived' shall be defined according to the RESPONSE message body:

a) if it is empty, the signature of the method '<<op>>ResponseReceived' shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise, the signature of the method '<<op>>ResponseReceived' shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            <<Res [i]>> res<<i>>, ... <<Res [N]>> res<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.22.3 The signature of the method '<<op>>ErrorReceived' shall be:

```
void <<op>>ErrorReceived(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& error,
            const MALQoSProperties& qosProperties)
```

4.3.3.23 Specific INVOKE Adapter Methods

4.3.3.23.1 For each INVOKE operation three specific adapter methods shall be defined:

a) <<op>>AckReceived;

b) <<op>>AckErrorReceived;

c) <<op>>ResponseReceived;

d) <<op>>ReponseErrorReceived.

4.3.3.23.2 The signature of the method '<<op>>AckReceived' shall be defined according to the ACK message body:

a) if it is empty the signature shall be:

```
void <<op>>AckReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise the signature shall be:

```
void <<op>>AckReceived(
            const shared_ptr<MALMessageHeader>& header,
            <<Ack [i]>> ack<<i>>, ... <<Ack [N]>> ack<<N>>,
            const MALQoSProperties& qosProperties )
```

4.3.3.23.3 The signature of the method '<<op>>AckErrorReceived' shall be:

```
void <<op>>AckErrorReceived(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& error,
            const MALQoSProperties& qosProperties)
```

4.3.3.23.4 The signature of the method '<<op>>ResponseReceived' shall be defined according to the RESPONSE message body:

a) if it is empty the signature shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise the signature shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            <<Res [i]>> res<<i>>, ... <<Res [N]>> res<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.23.5 The signature of the method '<<op>>ResponseErrorReceived' shall be:

```
void <<op>>ResponseErrorReceived(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& error,
            const MALQoSProperties& qosProperties)
```

4.3.3.24 Specific PROGRESS Adapter Methods

4.3.3.24.1 For each PROGRESS operation four specific adapter methods shall be defined:

a) <<op>>AckReceived;

b) <<op>>AckErrorReceived;

c) <<op>>UpdateReceived;

d) <<op>>UpdateErrorReceived;

e) <<op>>ResponseReceived;

f) <<op>>ResponseErrorReceived.

4.3.3.24.2 The signature of the method '<<op>>AckReceived' shall be defined according to the ACK message body:

a) if it is empty the signature shall be:

```
void <<op>>AckReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise the signature shall be:

```
void <<op>>AckReceived (
            const shared_ptr<MALMessageHeader>& header,
            Ack [i]>> ack<<i>>, ... <<Ack [N]>> ack<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.24.3 The signature of the method '<<op>>AckErrorReceived' shall be:

```
void <<op>>AckErrorReceived(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& error)
```

4.3.3.24.4 The signature of the method '<<op>>UpdateReceived' shall be defined according to the UPDATE message body:

a) if it is empty the signature shall be:

```
void <<op>>UpdateReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise the signature shall be:

```
void <<op>>UpdateReceived(
            const shared_ptr<MALMessageHeader>& header,
            <<Update [i]>> update<<i>>, ... <<Update [N]>> update<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.24.5 The signature of the method '<<op>>UpdateErrorReceived' shall be:

```
void <<op>>UpdateErrorReceived(
           const shared_ptr<MALMessageHeader>& header,
           const shared_ptr<MALStandardError>& error,
           const MALQoSProperties& qosProperties)
```

4.3.3.24.6 The signature of the method '<<op>>ResponseReceived' shall be defined according to the RESPONSE message body:

a) if it is empty the signature shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            const MALQoSProperties& qosProperties)
```

b) otherwise the signature shall be:

```
void <<op>>ResponseReceived(
            const shared_ptr<MALMessageHeader>& header,
            <<Res [i]>> res<<i>>, ... <<Res [N]>> res<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.24.7 The signature of the method '<<op>>ResponseErrorReceived' shall be:

```
void <<op>>ResponseErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALStandardError>& error,
                  const MALQoSProperties& qosProperties)
```

4.3.3.25 Specific PUBLISH-SUBSCRIBE Adapter Methods

4.3.3.25.1 For each PUBLISH-SUBSCRIBE operation six specific adapter methods shall be defined:

a) <<op>>RegisterAckReceived;

b) <<op>>RegisterErrorReceived;

c) <<op>>DeregisterAckReceived;

d) <<op>>NotifyReceived;

e) <<op>>NotifyErrorReceived.

4.3.3.25.2 The signature of the method '<<op>>RegisterAckReceived' shall be:

```
void <<op>>RegisterAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  <<Ack>> ack,
                  const MALQoSProperties& qosProperties)
```

4.3.3.25.3 The signature of the method '<<op>>RegisterErrorReceived' shall be:

```
void <<op>>RegisterErrorReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALStandardError>& error,
                  const MALQoSProperties& qosProperties )
```

4.3.3.25.4 The signature of the method '<<op>>DeregisterAckReceived' shall be:

```
void <<op>>DeregisterAckReceived(
                  const shared_ptr<MALMessageHeader>& header,
                  <<Ack>> ack,
                  const MALQoSProperties& qosProperties)
```

4.3.3.25.5 The signature of the method '<<op>>NotifyReceived' shall be defined according to the <<Notify [N]>> type:

```
void <<op>>NotifyReceived(
            const shared_ptr<MALMessageHeader>& header,
            const Identifier& subscriptionId,
            const shared_ptr<UpdateHeaderList>& updateHeaderList,
            const <<Notify [i]>>List& updateList<<i>>, ...
            const <<Notify [N]>>List& updateList<<N>>,
            const MALQoSProperties& qosProperties)
```

4.3.3.25.6 The signature of the method '<<op>>NotifyErrorReceived' shall be:

```
public void <<op>>NotifyErrorReceived(MALMessageHeader header,
  MALStandardError error, C++.util.Map qosProperties)
```

4.3.3.26 Notify from Other Service Adapter Method

4.3.3.26.1 A method 'notifyReceivedFromOtherService' shall be defined in order to enable a consumer to receive Notify messages from any services, i.e., not only from the consumed service.

4.3.3.26.2 The signature of method 'notifyReceivedFromOtherService' shall be:

```
void notifyReceivedFromOtherService(
            const shared_ptr<MALMessageHeader>& msgHeader,
            const shared_ptr<MALNotifyBody>& body,
            const MALQoSProperties& qosProperties)
```

4.3.4 STUB IMPLEMENTATION CLASS

4.3.4.1 Definition

4.3.4.1.1 A Stub class shall be defined in order to implement the Stub interface.

4.3.4.1.2 The name of the class shall be the identifier of the service suffixed with 'Stub': <<Service>>Stub.

4.3.4.1.3 The stub class shall implement the interface <<Service>>.

4.3.4.1.4 The class shall define the attribute specified in table 4-4.

Table 4-4: <<Service>>Stub Attribute

Attribute	Type
consumer	MALConsumer

4.3.4.1.5 The stub class shall provide a public constructor.

4.3.4.1.6 The constructor signature shall be:

```
<<Service>>Stub(const shared_ptr<MALConsumer>& consumer)
```

4.3.4.1.7 The constructor shall assign the attribute 'consumer' with the value of the parameter 'consumer'.

4.3.4.1.8 When calling the MALConsumer, the Stub shall convert each body element which type is mapped to a Union by creating a new Union from the body element.

4.3.4.1.9 When getting a body element from a MALMessageBody, the Stub class shall pass an instance of the element if the type is concrete, i.e., if there is no polymorphism; otherwise the Stub class shall pass the value NULL.

4.3.4.1.10 When returning a result from the MALConsumer, the Stub shall convert each body element which type is Union to its C++ type value.

4.3.4.2 SEND Operation Invocation

For each SEND operation, the method '<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'send' provided by the attribute 'consumer' with the following parameters:

a) the MALSendOperation description resolved from the <<Service>>Helper;

b) the body elements.

4.3.4.3 SUBMIT Operation Synchronous Invocation

For each SUBMIT operation, the method '<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'submit' provided by the attribute 'consumer' with the following parameters:

a) the MALSubmitOperation description resolved from the <<Service>>Helper;

b) the body elements.

4.3.4.4 REQUEST Operation Synchronous Invocation

4.3.4.4.1 For each REQUEST operation, the method '<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'request' provided by the attribute 'consumer' with the following parameters:

a) the MALRequestOperation description resolved from the <<Service>>Helper;

b) the body elements.

4.3.4.4.2 If the RESPONSE is not empty, the body elements returned by the call to the method 'request' shall be returned as the result of the method '<<op>>' in the following way:

a) if there is only one body element, then it shall be returned;

b) if there is more than one body element, then an <<Op>>Response shall be created from the body elements and returned.

4.3.4.5 INVOKE Operation Synchronous Invocation

4.3.4.5.1 For each INVOKE operation, the method '<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'invoke' provided by the attribute 'consumer' with the following parameters:

a) the MALInvokeOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.5.2 If the ACK is not empty, the body elements returned by the call to the method 'invoke' shall be returned as the result of the method '<<op>>' in the following way:

a) if there is only one body element, then it shall be returned;

b) if there is more than one body element, then a <<Op>>Ack shall be created from the body elements and returned.

4.3.4.6 PROGRESS Operation Synchronous Invocation

4.3.4.6.1 For each PROGRESS operation, the method '<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'progress' provided by the attribute 'consumer' with the following parameters:

a) the MALProgressOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.6.2 If the ACK is not empty, the Element returned by the call to the method 'progress' shall be returned as the result of the method '<<op>>' in the following way:

a) if there is only one body element, then it shall be returned;

b) if there is more than one body element, then a <<Op>>Ack shall be created from the body elements and returned.

4.3.4.7 PUBLISH-SUBSCRIBE Operation Synchronous Invocation

4.3.4.7.1 For each PUBLISH-SUBSCRIBE operation, the method 'registerFor<<op>>' inherited from the interface <<Service>> shall be implemented by calling the method 'register' provided by the attribute 'consumer' with the following parameters:

a) the MALPubSubOperation description resolved from the <<Service>>Helper;

b) the Subscription;

c) the <<Service>>Adapter.

4.3.4.7.2 For each PUBLISH-SUBSCRIBE operation, the method ‘deregisterFor<<op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘deregister’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALPubSubOperation description resolved from the <<Service>>Helper;

b) the IdentifierList.

4.3.4.8 SUBMIT Operation Asynchronous Invocation

For each SUBMIT operation, the method ‘async<<Op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncSubmit’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALSubmitOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.9 REQUEST Operation Asynchronous Invocation

For each REQUEST operation, the method ‘async<<Op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncRequest’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALRequestOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.10 INVOKE Operation Asynchronous Invocation

For each INVOKE operation, the method ‘async<<Op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncInvoke’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALInvokeOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.11 PROGRESS Operation Asynchronous Invocation

For each PROGRESS operation, the method ‘async<<Op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncProgress’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALProgressOperation description resolved from the <<Service>>Helper;

b) the <<Service>>Adapter;

c) the body elements.

4.3.4.12 PUBLISH-SUBSCRIBE Operation Asynchronous Invocation

4.3.4.12.1 For each PUBLISH-SUBSCRIBE operation, the method ‘async<<Op>>Register’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncRegister’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALPubSubOperation description resolved from the <<Service>>Helper;

b) the Subscription;

c) the <<Service>>Adapter.

4.3.4.12.2 For each PUBLISH-SUBSCRIBE operation, the method ‘async<<Op>>Deregister’ inherited from the interface <<Service>> shall be implemented by calling the method ‘asyncDeregister’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALPubSubOperation description resolved from the <<Service>>Helper;

b) the IdentifierList;

c) the <<Service>>Adapter.

4.3.4.13 Continue an Interaction

For each SUBMIT, REQUEST, INVOKE and PROGRESS operation, the method ‘continue<<Op>>’ inherited from the interface <<Service>> shall be implemented by calling the method ‘continueInteraction’ provided by the attribute ‘consumer’ with the following parameters:

a) the MALOperation description resolved from the <<Service>>Helper;

b) the parameters ‘lastInteractionStage’, ‘initiationTimestamp’ and ‘transactionId’;

c) the <<Service>>Adapter.

4.4 PROVIDER

4.4.1 OVERVIEW

Figure 4-2 is a class diagram that describes the relationships between the skeleton classes and interfaces at the provider side with the delegation pattern. The following classes or interfaces are shown:

a) <<Service>>Skeleton is the skeleton interface;

b) <<Service>>DelegationSkeleton is the skeleton implementation class according to the delegation pattern;

c) <<Service>>Handler is the service specific handler interface;

d) <<Service>>HandlerImpl is the service implementation class;

e) <<Op>>Publisher is a specific publishing interface;

f) <<Operation>>Interaction is a specific interaction interface.

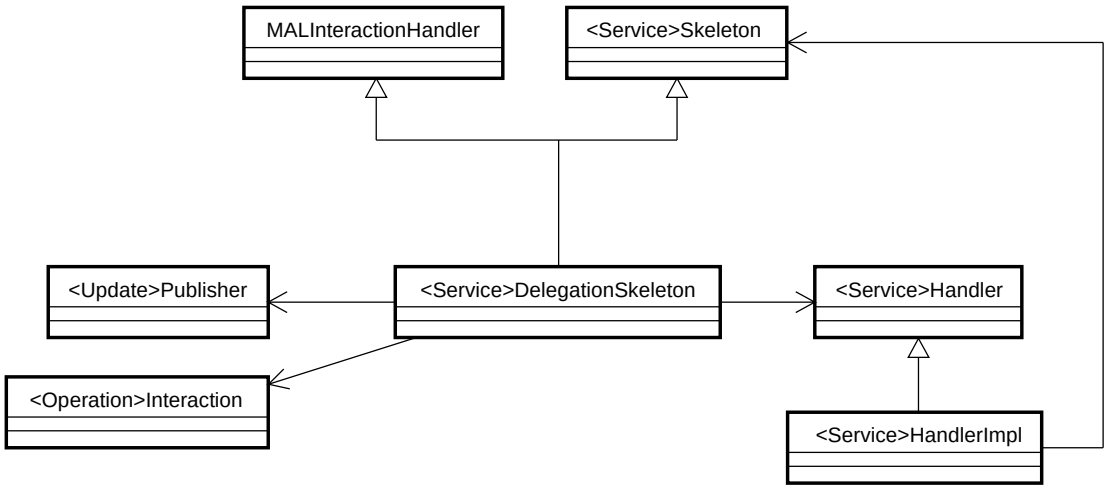

Figure 4-2: Relationships between the Skeleton Classes and Interfaces (Delegation Pattern)

With the inheritance pattern, the class <<Service>>DelegationSkeleton is replaced by the class <<Service>>InheritanceSkeleton, which is the skeleton implementation class according to the inheritance pattern. The class diagram becomes that shown in figure 4-3.

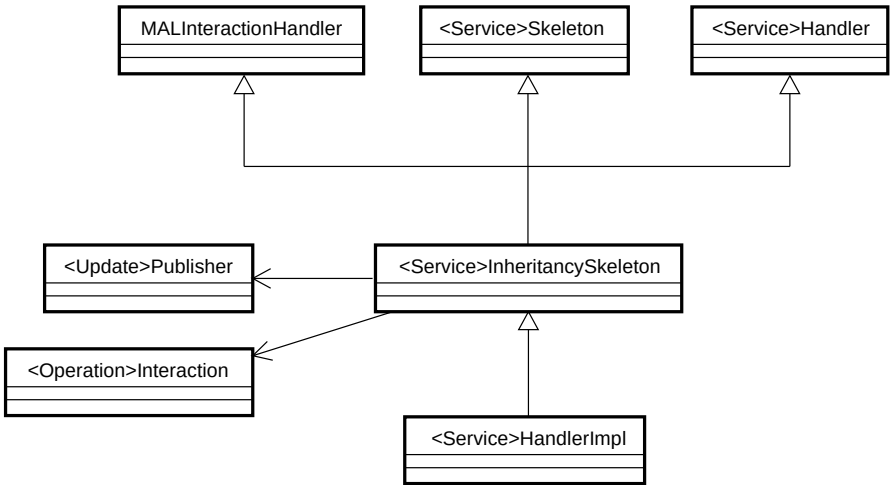

Figure 4-3: Relationships between the Skeleton Classes and Interfaces (Inheritance Pattern)

A service provider can be bound several times to one or several transport layers. Such a service provider is called a ‘multi-binding service provider’. It is represented by a single skeleton instance bound to several MALProviders as shown by figure 4-4.

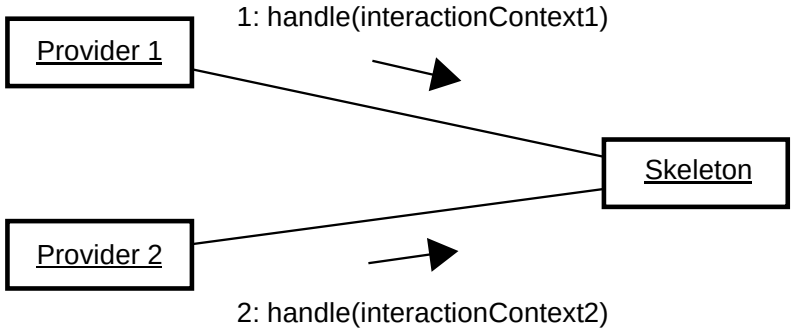

Figure 4-4: Multi-Binding Service Provider

4.4.2 HANDLER INTERFACE

4.4.2.1 Definition

4.4.2.1.1 A handler interface shall be defined in order to handle interaction patterns through service specific methods.

4.4.2.1.2 The name of the interface is the identifier of the service suffixed with ‘Handler’: <<Service>>Handler.

4.4.2.2 SEND Operation Invocation

4.4.2.2.1 A method ‘<<op>>’ shall be defined for each SEND operation provided by the service.

4.4.2.2.2 The signature of the method ‘<<op>>’ shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInteraction>& interaction)
```

4.4.2.3 SUBMIT Operation Invocation

4.4.2.3.1 A method ‘<<op>>’ shall be defined for each SUBMIT operation provided by the service.

4.4.2.3.2 The signature of the method ‘<<op>>’ shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInteraction>& interaction)
```

4.4.2.4 REQUEST Operation Invocation

4.4.2.4.1 A method ‘<<op>>’ shall be defined for each REQUEST operation provided by the service.

4.4.2.4.2 If the RESPONSE message body is empty, then the signature of the method ‘<<op>>’ shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInteraction>& interaction)
```

4.4.2.4.3 If the RESPONSE message body contains one element, then the signature of the method ‘<<op>>’ shall be:

```
<<Res [1]>> <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInteraction>& interaction)
```

4.4.2.4.4 If the RESPONSE message body contains more than one element, then the signature of the method ‘<<op>>’ shall be:

```
<<Op>>Response <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInteraction>& interaction)
```

4.4.2.5 INVOKE Operation Invocation

4.4.2.5.1 A method '<<op>>' shall be defined for each INVOKE operation provided by the service.

4.4.2.5.2 The signature of the method '<<op>>' shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALInvoke>& interaction)
```

4.4.2.6 PROGRESS Operation Invocation

4.4.2.6.1 A method '<<op>>' shall be defined for each PROGRESS operation provided by the service.

4.4.2.6.2 The signature of the method '<<op>>' shall be:

```
void <<op>>(
            <<In [i]>> p<<i>>, ... <<In [N]>> p<<N>>,
            const shared_ptr<MALProgress>& interaction)
```

4.4.2.7 Skeleton setter

4.4.2.7.1 A method 'setSkeleton' shall be defined in order to set the reference of the skeleton object.

4.4.2.7.2 The signature of the method 'setSkeleton' shall be:

```
void setSkeleton(const shared_ptr<<<Service>>Skeleton>& skeleton)
```

4.4.3 SKELETON INTERFACE

4.4.3.1 Definition

4.4.3.1.1 A skeleton interface shall be defined in order to enable the creation of publishers.

4.4.3.1.2 The name of the interface is the identifier of the service suffixed with 'Skeleton': <<Service>>Skeleton.

4.4.3.2 Create a Publisher

4.4.3.2.1 A method 'create<<Op>>Publisher' shall be defined for each PUBLISH-SUBSCRIBE operation in order to create a specific publisher.

4.4.3.2.2 The signature of the method ‘create<<Op>>Publisher’ shall be:

```
<<Op>>Publisher create<<Op>>Publisher(
      const IdentifierList& domain,
      const Identifier& networkZone,
      const SessionType& sessionType,
      const Identifier& sessionName,
      const QoSLevel& publishQos,
      const MALQoSProperties& publishProperties,
      const uint32_t& publishPriority)
```

4.4.4 PUBLISHER

4.4.4.1 Definition

4.4.4.1.1 A publisher class shall be defined in order to enable the handler to publish updates to the brokers owned by the activating providers, i.e., the MALProviders that activate the handler.

4.4.4.1.2 One publisher class shall be defined for each PUBLISH-SUBSCRIBE operation.

4.4.4.1.3 A publisher class shall belong to the ‘provider’ namespace of the service that owns the PUBLISH-SUBSCRIBE operation.

4.4.4.1.4 The name of the class shall be the identifier of the PUBLISH-SUBSCRIBE operation suffixed with ‘Publisher’: <<Op>>Publisher.

4.4.4.1.5 The class shall define the attribute specified in table 4-5.

Table 4-5: <<Op>> Publisher Attribute

Attribute	Type
publisherSet	MALPublisherSet

4.4.4.1.6 The publisher class shall provide a public constructor.

4.4.4.1.7 The constructor signature shall be:

```
<<Op>>Publisher(const shared_ptr<MALPublisherSet>& publisherSet)
```

4.4.4.1.8 The constructor parameter shall be assigned as described in table 4-6.

Table 4-6: <<Op>>Publisher Constructor Parameter

Parameter	Description
publisherSet	MALPublisherSet created through the MALProviderSet owned by the skeleton

4.4.4.2 Synchronous REGISTER Invocation

4.4.4.2.1 A method ‘syncRegister’ shall be defined in order to call the MALPublisherSet method ‘syncRegister’.

4.4.4.2.2 The signature of the method ‘register’ shall be the same as the MALPublisherSet method ‘syncRegister’.

4.4.4.3 Synchronous DEREGISTER Invocation

4.4.4.3.1 A method ‘deregister’ shall be defined in order to call the MALPublisherSet method ‘deregister’.

4.4.4.3.2 The signature of the method ‘deregister’ shall be the same as the MALPublisherSet method ‘deregister’.

4.4.4.4 Asynchronous REGISTER Invocation

4.4.4.4.1 A method ‘asyncRegister’ shall be defined in order to call the MALPublisherSet method ‘asyncRegister’.

4.4.4.4.2 The signature of the method ‘asyncRegister’ shall be the same as the MALPublisherSet method ‘asyncRegister’.

4.4.4.5 Asynchronous DEREGISTER Invocation

4.4.4.5.1 A method ‘asyncDeregister’ shall be defined in order to call the MALPublisherSet method ‘asyncDeregister’.

4.4.4.5.2 The signature of the method ‘asyncDeregister’ shall be the same as the MALPublisherSet method ‘asyncDeregister’.

4.4.4.6 PUBLISH Invocation

4.4.4.6.1 A method ‘publish’ shall be defined in order to call the MALPublisherSet method ‘publish’.

4.4.4.6.2 The signature of the method ‘publish’ shall be:

```
void publish(
            const shared_ptr<UpdateHeaderList>& updateHeaderList,
            const <<Notify [i]>>List& updateList<<i>>,
            ... const <<Notify [N]>>List& updateList<<N>>)
```

4.4.4.6.3 The parameters of the method ‘publish’ shall be assigned as described in table 4-7.

Table 4-7: <<Op>>Publisher ‘publish’ Parameters

Parameter	Description
updateHeaderList	Headers of the updates to be published
updateList<<i>>	List of updates to be published

4.4.4.7 Close

4.4.4.7.1 A method ‘close’ shall be defined in order to call the MALPublisherSet method ‘close’.

4.4.4.7.2 The signature of the method ‘close’ shall be the same as the MALPublisherSet method ‘close’.

4.4.5 INVOKE INTERACTION

4.4.5.1 Definition

4.4.5.1.1 For each INVOKE operation an <<Op>>Interaction class shall be defined.

4.4.5.1.2 The <<Op>>Interaction class shall define the attribute specified in table 4-8.

Table 4-8: Invoke <<Op>>Interaction Attribute

Attribute	Type
interaction	MALInvoke

4.4.5.1.3 The <<Op>>Interaction class shall provide a public constructor.

4.4.5.1.4 The <<Op>>Interaction constructor signature shall be:

```
<<Op>>Interaction(const shared_ptr<MALInvoke>& interaction)
```

4.4.5.1.5 When calling the MALInvoke, the <<Op>>Interaction class shall convert each body element which type is mapped to a Union by creating a new Union from the body element.

4.4.5.1.6 If the message body is empty, then the value NULL shall be passed to the MALInvoke interaction.

4.4.5.2 Interaction Getter

4.4.5.2.1 A method ‘getInteraction’ shall be defined in order to get the MALInvoke interaction.

4.4.5.2.2 The signature of the method ‘getInteraction’ shall be:

```
shared_ptr<MALInvoke> getInteraction()
```

4.4.5.3 Specific ACK Sending

4.4.5.3.1 A method ‘sendAcknowledgement’ shall be defined in order to send a specific ACK.

4.4.5.3.2 The signature of the method ‘sendAcknowledgement’ shall be:

```
shared_ptr<MALMessage> sendAcknowledgement(
            <<Ack [i]>> ack<<i>>, ... <<Ack [N]>> ack<<N>>)
```

4.4.5.3.3 The method shall be implemented in order to call the method ‘sendAcknowledgement’ provided by the MALInvoke interaction.

4.4.5.4 Specific RESPONSE Sending

4.4.5.4.1 A method ‘sendResponse’ shall be defined in order to send a specific RESPONSE.

4.4.5.4.2 The signature of the method ‘sendResponse’ shall be:

```
shared_ptr<MALMessage> sendResponse(
            <<Res [i]>> res<<i>>, ... <<Res [N]>> res<<N>>)
```

4.4.5.4.3 The method ‘sendResponse’ shall be implemented in order to call the method ‘sendResponse’ provided by the MALInvoke interaction.

4.4.5.5 Generic ERROR Sending

4.4.5.5.1 A method ‘sendError’ shall be defined in order to send an ERROR.

4.4.5.5.2 The signature of the method ‘sendError’ shall be:

```
shared_ptr<MALMessage> sendError(
            const shared_ptr<MALStandardError>& error)
```

4.4.5.5.3 The method ‘sendError’ shall be implemented in order to call the method ‘sendError’ provided by the MALInvoke interaction.

4.4.6 PROGRESS INTERACTION

4.4.6.1 Definition

4.4.6.1.1 For each PROGRESS operation an <<Op>>Interaction class shall be defined.

4.4.6.1.2 The <<Op>>Interaction class shall define the attribute specified in table 4-9.

Table 4-9: Progress <<Op>>Interaction Attribute

Attribute	Type
interaction	MALProgress

4.4.6.1.3 The <<Op>>Interaction class shall provide a public constructor.

4.4.6.1.4 The <<Op>>Interaction constructor signature shall be:

```
<<Op>>Interaction(const shared_ptr<MALProgress>& interaction)
```

4.4.6.1.5 When calling the MALProgress, the <<Op>>Interaction class shall convert each body element which type is mapped to a Union by creating a new Union from the body element.

4.4.6.1.6 If the message body is empty, then the value NULL shall be passed to the MALProgress interaction.

4.4.6.2 Interaction Getter

4.4.6.2.1 A method ‘getInteraction’ shall be defined in order to get the MALProgress interaction.

4.4.6.2.2 The signature of the method ‘getInteraction’ shall be:

```
shared_ptr<MALProgress> getInteraction()
```

4.4.6.3 Specific ACK Sending

4.4.6.3.1 A method ‘sendAcknowledgement’ shall be defined in order to send a specific ACK.

4.4.6.3.2 The signature of the method ‘sendAcknowledgement’ shall be defined according to the ACK message body:

```
shared_ptr<MALMessage> sendAcknowledgement(
            <<Ack [i]>> ack<<i>>, ... <<Ack [N]>> ack<<N>>)
```

4.4.6.3.3 The method ‘sendAcknowledgement’ shall be implemented in order to call the method ‘sendAcknowledgement’ provided by the MALProgress.

4.4.6.4 Specific UPDATE Sending

4.4.6.4.1 A method ‘sendUpdate’ shall be defined in order to send a specific UPDATE.

4.4.6.4.2 The signature of the method ‘sendUpdate’ shall be:

```
shared_ptr<MALMessage> sendUpdate(
      const shared_ptr<<<Update>>>& update)
```

4.4.6.4.3 The method ‘sendUpdate’ shall be implemented in order to call the method ‘sendUpdate’ provided by the MALProgress interaction.

4.4.6.5 Specific RESPONSE Sending

4.4.6.5.1 A method ‘sendResponse’ shall be defined in order to send a specific RESPONSE.

4.4.6.5.2 The signature of the method ‘sendResponse’ shall be:

```
shared_ptr<MALMessage> sendResponse(
            <<Res [i]>> res<<i>>, ... <<Res [N]>> res<<N>>)
```

4.4.6.5.3 The method ‘sendResponse’ shall be implemented in order to call the method ‘sendResponse’ provided by the MALProgress interaction.

4.4.6.6 Generic ERROR Sending

4.4.6.6.1 A method ‘sendError’ shall be defined in order to send an ERROR.

4.4.6.6.2 The signature of the method ‘sendError’ shall be:

```
shared_ptr<MALMessage> sendError(
            const shared_ptr<MALStandardError>& error)
```

4.4.6.6.3 The method ‘sendError’ shall be implemented in order to call the method ‘sendError’ provided by the MALProgress interaction.

4.4.7 INHERITANCE SKELETON IMPLEMENTATION CLASS

4.4.7.1 Definition

4.4.7.1.1 A Skeleton class shall be defined in order to enable to implement a handler by inheriting from this skeleton.

4.4.7.1.2 The name of the class shall be the identifier of the service suffixed with ‘InheritanceSkeleton’: <<Service>>InheritanceSkeleton.

4.4.7.1.3 The <<Service>>InheritanceSkeleton class shall be abstract.

4.4.7.1.4 The <<Service>>InheritanceSkeleton class shall implement three interfaces:

a) MALInteractionHandler;

b) <<Service>>Skeleton;

c) <<Service>>Handler.

4.4.7.1.5 The <<Service>>InheritanceSkeleton class shall define the attributes specified in table 4-10.

Table 4-10: <<Service>>InheritanceSkeleton Attribute

Attribute	Type
providerSet	MALProviderSet

4.4.7.1.6 The <<Service>>InheritanceSkeleton class shall provide a public empty constructor.

4.4.7.1.7 The <<Service>>InheritanceSkeleton constructor signature shall be:

```
<<Service>>InheritanceSkeleton()
```

4.4.7.1.8 The <<Service>>InheritanceSkeleton constructor shall assign the attribute ‘providerSet’ with a new ProviderSet taking as a parameter the MALService statically declared by the <<Service>>Helper.

4.4.7.1.9 When getting a body element from a MALMessageBody, the <<Service>>InheritanceSkeleton class shall pass an instance of the element if the type is concrete, i.e., if there is no polymorphism; otherwise the <<Service>>InheritanceSkeleton class shall pass the value NULL.

4.4.7.1.10 When calling the <<Service>>Handler, the <<Service>>InheritanceSkeleton class shall convert each body element which type is Union to its C++ type value.

4.4.7.1.11 When returning a result from the <<Service>>Handler, the <<Service>>InheritanceSkeleton class shall convert each body element which type is mapped to a Union by creating a new Union from the body element.

4.4.7.2 MAL Initialization

The method ‘malInitialize’ shall be implemented by adding the MALProvider to the MALProviderSet.

4.4.7.3 MAL Finalization

The method ‘malFinalize’ shall be implemented by removing the MALProvider from the MALProviderSet.

4.4.7.4 Creating Publishers

4.4.7.4.1 The <<Service>>InheritanceSkeleton class shall implement the methods ‘create<<Op>>Publisher’ inherited from the <<Service>>Skeleton interface.

4.4.7.4.2 The implementation shall create a new MALPublisherSet from the MALProviderSet, call the <<Op>>Publisher constructor with the created MALPublisherSet as a parameter and return the result.

4.4.7.5 Handle SEND Interactions

4.4.7.5.1 The method ‘handleSend’ inherited from the <<Service>>Handler interface shall be implemented by calling the specific handler method according to the operation.

4.4.7.5.2 If the operation is not defined by the service, then a MAL standard error MAL::UNSUPPORTED_OPERATION shall be raised as a MALInteractionException.

4.4.7.6 Handle SUBMIT Interactions

4.4.7.6.1 The method ‘handleSubmit’ inherited from the <<Service>>Handler interface shall be implemented by calling the specific handler method according to the operation.

4.4.7.6.2 Once the specific handler method ‘<<op>>’ has been called the method ‘sendAcknowledgement’ provided by the MALSubmit parameter shall be called.

4.4.7.6.3 If the operation is not defined by the service, then a MAL standard error MAL::UNSUPPORTED_OPERATION shall be raised as an ACK ERROR through the MALSubmit interaction context.

4.4.7.7 Handle REQUEST Interactions

4.4.7.7.1 The method ‘handleRequest’ inherited from the <<Service>>Handler interface shall be implemented by calling the specific handler method according to the operation.

4.4.7.7.2 Once the specific handler method ‘<<op>>’ has been called the method ‘sendResponse’ provided by the MALRequest parameter shall be called.

4.4.7.7.3 If the returned message body is empty, then no value shall be passed to the interaction.

4.4.7.7.4 If the operation is not defined by the service, then a MAL standard error MAL::UNSUPPORTED_OPERATION shall be raised as a RESPONSE ERROR through the MALRequest interaction context.

4.4.7.8 Handle INVOKE Interactions

4.4.7.8.1 The method ‘handleInvoke’ inherited from the <<Service>>Handler interface shall be implemented by calling the specific handler method according to the operation.

4.4.7.8.2 a new <<Op>>Interaction shall be instantiated with the MALInvoke parameter of the method ‘handleInvoke’ and passed as the <<Op>>Interaction parameter of the method ‘<<op>>’.

4.4.7.8.3 If the operation is not defined by the service, then a MAL standard error MAL::UNSUPPORTED_OPERATION shall be raised as an ACK ERROR through the MALInvoke interaction context.

4.4.7.9 Handle PROGRESS Interactions

4.4.7.9.1 The method ‘handleProgress’ inherited from the <<Service>>Handler interface shall be implemented by calling the specific handler method according to the operation.

4.4.7.9.2 a new <<Op>>Interaction shall be instantiated with the MALProgress parameter of the method ‘handleProgress’ and passed as the <<Op>>Interaction parameter of the method ‘<<op>>’.

4.4.7.9.3 If the operation is not defined by the service, then a MAL standard error MAL::UNSUPPORTED_OPERATION shall be raised as an ACK ERROR through the MALProgress interaction context.

4.4.8 DELEGATION SKELETON IMPLEMENTATION CLASS

4.4.8.1 Definition

4.4.8.1.1 A skeleton class shall be defined in order to enable to implement a handler by implementing the <<Service>>Handler interface.

4.4.8.1.2 The name of the class shall be the identifier of the service suffixed with 'DelegationSkeleton': <<Service>>DelegationSkeleton.

4.4.8.1.3 The <<Service>>DelegationSkeleton class shall not be abstract.

4.4.8.1.4 The <<Service>>DelegationSkeleton class shall implement two interfaces:

a) MALInteractionHandler;

b) <<Service>>Skeleton.

4.4.8.1.5 The <<Service>>DelegationSkeleton class shall define the attributes specified in table 4-11.

Table 4-11: <<Service>>DelegationSkeleton Attributes

Attribute	Type
delegate	<<Service>>Handler
providerSet	MALProviderSet

4.4.8.1.6 The <<Service>>DelegationSkeleton class shall provide a public constructor.

4.4.8.1.7 The <<Service>>DelegationSkeleton constructor signature shall be:

```
<<Service>>DelegationSkeleton(
            const shared_ptr<<<Service>>Handler>& delegate)
```

4.4.8.1.8 The <<Service>>DelegationSkeleton constructor shall assign the attribute 'delegate' with the value of the parameter 'delegate'.

4.4.8.2 MAL Initialization

The <<Service>>DelegationSkeleton class shall implement the methods 'malInitialize' in the same way as the inheritance skeleton.

4.4.8.3 MAL Finalization

The <<Service>>DelegationSkeleton class shall implement the methods ‘malFinalize’ in the same way as the inheritance skeleton.

4.4.8.4 Creating Publishers

The <<Service>>DelegationSkeleton class shall implement the methods ‘create<<Op>>Publisher’ in the same way as the inheritance skeleton.

4.4.8.5 Handle Interactions

The <<Service>>DelegationSkeleton class shall implement the method ‘handle<<Ip>>’ in the same way as the inheritance skeleton except that the specific handler method is provided by the attribute ‘delegate’.

4.5 DATA STRUCTURES

4.5.1 OVERVIEW

This subsection specifies how enumeration, list and composite data structures are mapped to classes used on both consumer and provider sides.

4.5.2 NAMESPACE

4.5.2.1 If the data structure belongs to an area, then the namespace shall be:

```
<<root name>>::<<!area!>>::structures
```

4.5.2.2 If the data structure belongs to a service, then the namespace shall be:

```
<<root name>>::<<!area!>>::<<!service!>>::structures
```

4.5.3 CLASS DEFINITION

4.5.3.1 A class shall be defined for each data type.

4.5.3.2 The class name shall be the name of the data type: <<Type name>>.

4.5.3.3 A list class shall be defined for every concrete data type.

4.5.3.4 The list class name shall be: <<List element type name>>List.

4.5.3.5 The class shall be public.

4.5.3.6 If the data type is concrete, then the class shall be final.

4.5.3.7 If the data type is abstract, then the class shall be abstract.

4.5.4 INTERFACE DEFINITION

4.5.4.1 A list marker interface shall be defined for every abstract data type.

4.5.4.2 The list marker interface name shall be: <<List element type name>>List.

4.5.5 SHORT FORM

4.5.5.1 Constants declaration

4.5.5.2 Limitation

The C++ MAL API shall support only data types whose short form part is strictly less than 2^{23} and strictly greater than -2^{23}.

4.5.5.2.1 If the data type is concrete, then its absolute short form shall be computed as follows:

a) the absolute short form shall be a long integer;

b) the area number shall be shifted to the leftmost position of the absolute short form: bits 48 to 63;

c) the service number shall be shifted to the left of the absolute short form after the area number: bits 32 to 47;

d) the area version number shall be shifted to the left of the absolute short form after the service number: bits 24 to 31;

e) the relative type shall be assigned to the rightmost position of the absolute short form: bits 0 to 23.

4.5.5.2.2 If the data type is concrete, then the following constants shall be declared:

```
static const int64_t SHORT_FORM = <<absolute short form>>

static const int32_t TYPE_SHORT_FORM = <<type short form>>
```

4.5.5.3 Get the Short Form

If the data type is concrete, then the method ‘getShortForm’ inherited from the Element interface shall be implemented by returning the constant SHORT_FORM declared by the class.

4.5.5.4 Get the Area Number

If the data type is concrete, then the method ‘getAreaNumber’ inherited from the Element interface shall be implemented by returning the number of the area where the data type is defined.

4.5.5.5 Get the Service Number

If the data type is concrete, then the method ‘getServiceNumber’ inherited from the Element interface shall be implemented as follows:

a) if the data type belongs to a service, then the number of the service shall be returned;

b) if the data type does not belong to a service, then the MALService constant NULL_SERVICE_NUMBER shall be returned.

4.5.5.6 Get the Type Short Form

If the data type is concrete, then the method ‘getShortForm’ inherited from the Element interface shall be implemented by returning the constant TYPE_SHORT_FORM declared by the class.

4.5.6 ENUMERATION

4.5.6.1 Limitation

The MAL C++ API shall support only MAL enumeration values that are less than or equal to the greatest 32-bit signed integer.

4.5.6.2 Class Definition

4.5.6.2.1 The class shall extend the class Enumeration.

4.5.6.2.2 The class shall be final.

4.5.6.3 Enumeration Items

The following static constants shall be defined for each item of the enumeration:

a) the index of the enumerated item:

```
static const int _<<ENUM ITEM>>_INDEX =
  <<enum item index>>;
```

b) the numeric value of the enumerated item:

```
static const uint32_t <<ENUM ITEM>>_NUM_VALUE =
                                <<enum item numeric value>>
```

c) the enumerated item:

```
static const <<Enumeration>> <<ENUM ITEM>> =
                  <<Enumeration>>(_<<ENUM ITEM>>_INDEX);
```

4.5.6.4 Get the Enumerated Item from its Ordinal

4.5.6.4.1 A method ‘fromOrdinal’ shall be defined in order to return an enumerated item from its ordinal value.

4.5.6.4.2 The signature of the method ‘fromOrdinal’ shall be:

```
static <<Enumeration>> fromOrdinal(int ordinal)
```

4.5.6.5 Get the Enumerated Item from its Name

4.5.6.5.1 A method ‘fromString’ shall be defined in order to return an enumerated item from its name.

4.5.6.5.2 The signature of the method ‘fromString’ shall be:

```
static <<Enumeration>> fromString(const string& itemName)
```

4.5.6.6 Get an Enumerated Item from its Numeric Value

4.5.6.6.1 A method ‘fromNumericValue’ shall be defined in order to return an enumerated item from its numeric value.

4.5.6.6.2 The method signature shall be:

```
static <<Enumeration>> fromNumericValue(const uint32_t& nvalue)
```

4.5.6.7 ToString

The method ‘toString’ shall be redefined by returning the name of the enumerated item: <<ENUM ITEM>>.

4.5.6.8 Create an Element

The method ‘createElement’ defined by the interface Element shall be implemented by returning the first item of the enumeration.

4.5.6.9 Encode

The method ‘encode’ defined by the interface Element shall be implemented by encoding the field ‘ordinal’ through the MALEncoder as follows:

a) if the enumeration size is less than or equal to 256, then the method ‘encodeUOctet’ shall be called;

b) if the enumeration size is greater than 256 and less than or equal to 65536, then the method ‘encodeUShort’ shall be called;

c) if the enumeration size is greater than 65536, then the method ‘encodeUInteger’ shall be called.

4.5.6.10 Decode

4.5.6.10.1 The method ‘decode’ defined by the interface Element shall be implemented by decoding the field ‘ordinal’ and returning the result of the method ‘fromIndex’.

4.5.6.10.2 The ‘ordinal’ shall be decoded through the MALDecoder as follows:

a) if the enumeration size is less than or equal to 256, then the method ‘decodeUOctet’ shall be called;

b) if the enumeration size is greater than 256 and less than or equal to 65536, then the method ‘decodeUShort’ shall be called;

c) if the enumeration size is greater than 65536, then the method ‘decodeUInteger’ shall be called.

4.5.7 LIST CLASS

4.5.7.1 Class Definition

4.5.7.1.1 The list class shall inherit a class that implements C++ std::vector and assign the C++ Generic type variable with the list element mapping class.

4.5.7.1.2 If the list element type is a MAL::Attribute, then the class shall implement the interface AttributeList and assign the C++ Generic type variable with the list element mapping class.

4.5.7.1.3 If the list element type is an enumeration, then the class shall implement the interface ElementList and assign the C++ Generic type variable with the list element mapping class.

4.5.7.1.4 If the list element type is a composite, then:

a) if the element type extends MAL::Composite, then the list class shall implement the interface CompositeList and assign the C++ Generic type variable with the list element mapping class;

b) if the element type extends a type <<parent composite>>, then the list class shall implement the interface <<Parent composite>>List and assign the C++ Generic type variable with the list element mapping class.

4.5.7.2 Empty Constructor

4.5.7.2.1 An empty public constructor shall be defined.

4.5.7.2.2 The constructor signature shall be:

```
<<List class name>>()
```

4.5.7.3 Constructor with an Initial Capacity

4.5.7.3.1 A public constructor shall be defined in order to create a list with an initial capacity.

4.5.7.3.2 The specified initial capacity may not be taken into account depending on the implementation.

4.5.7.3.3 The constructor signature shall be:

```
<<List class name>>(int initialCapacity)
```

4.5.7.4 Encode

4.5.7.4.1 The method ‘encode’ inherited from the Element interface shall be implemented.

4.5.7.4.2 A MALListEncoder shall be created by calling the method ‘createListEncoder’ provided by the MALEncoder;

4.5.7.4.3 Each element of the list shall be encoded through the MALListEncoder as follows:

a) if the declared list element type is a MAL::Attribute, then the method ‘encodeNullable<<Attribute>>’ provided by the MALEncoder shall be called;

b) otherwise, the method ‘encodeNullableElement’ provided by the MALEncoder shall be called.

4.5.7.5 Decode

4.5.7.5.1 The method ‘decode’ inherited from the Element interface shall be implemented.

4.5.7.5.2 A MALListDecoder shall be created by calling the method ‘createListDecoder’ provided by the MALDecoder;

4.5.7.5.3 As long as the method ‘hasNext’ of the MALListDecoder returns TRUE, an element of the list shall be decoded through the MALListDecoder as follows:

a) if the declared List element type is a MAL::Attribute, then the method ‘decodeNullable<<Attribute>>’ provided by the MALDecoder shall be called;

b) otherwise, the method ‘decodeNullableElement’ provided by the MALDecoder shall be called with a new instance of the element as a parameter.

NOTE – The declared list element type cannot be an abstract MAL::Composite.

4.5.7.5.4 The decoded element shall be added to this list.

4.5.7.5.5 This list shall be returned as a result.

4.5.7.6 Create an Element

The method ‘createElement’ defined by the interface Element shall be implemented by calling the empty constructor and returning the result.

4.5.8 COMPOSITE

4.5.8.1 Class Definition

4.5.8.1.1 If the parent type is not MAL::Composite, then the class shall extend the class that represents the parent type: <<Composite parent class>>.

4.5.8.1.2 If the parent type is MAL::Composite, then the class shall implement the Composite interface.

4.5.8.2 Attributes Declaration

A private C++ attribute ‘<<field name>>’ typed ‘<<Field class>>’ shall be declared for each field of the Composite.

4.5.8.3 Empty Constructor

4.5.8.3.1 An empty public constructor shall be defined.

4.5.8.3.2 The constructor signature shall be:

```
<<Composite>>()
```

4.5.8.4 Constructor

4.5.8.4.1 A public constructor shall be defined with all the fields owned by this Composite and its parents.

4.5.8.4.2 The fields shall be declared in the same order as in the Composite type definition and descending the parent hierarchy.

4.5.8.5 Getters

4.5.8.5.1 A getter method ‘get<<Composite field name>>’ shall be defined for each field.

4.5.8.5.2 The signature of the method ‘get<<Composite field name>>’ shall be:

```
<<Composite field class>> get<<Composite field name>>()
```

4.5.8.6 Setters

4.5.8.6.1 A setter method ‘set<<Composite field name>>’ shall be defined for each field.

4.5.8.6.2 The signature of the method ‘set<<Composite field name>>’ shall be:

```
void set<<Composite field name>>(
  <<Composite field class>> <<field name>>)
```

4.5.8.7 Overloaded operator==

The C++ overloaded method ‘operator==’ shall be redefined as follows. The method shall return:

a) TRUE if the parameter is the same as ‘this’;

b) FALSE if the parameter is NULL;

c) FALSE if the parameter class is not compliant with the class <<Composite>>;

d) FALSE if the parent type is not MAL::Composite and the call to the super method ‘operator==’ returns FALSE;

e) FALSE if the value of a field from the Composite parameter is not equal to the value of the same field from this Composite;

f) TRUE in the other cases.

4.5.8.8 ToString

The method ‘toString’ shall be redefined by returning a String formatted as follows:

a) the first character shall be ‘(’;

b) the inherited toString method shall be called;

c) the character ‘,’ shall be appended;

d) for each field:

   1) the name of the field shall be appended;

   2) the character ‘=’ shall be appended;

   3) the toString representation of the field shall be appended;

   4) if the field is not the last one, the character ‘,’ shall be appended;

e) the last character shall be ‘)’.

4.5.8.9 Encode

The method ‘encode’ shall be implemented as follows:

a) if the parent type is not MALComposite, then the super ‘encode’ shall be called;

b) every field shall be encoded in the same order as its declaration;

c) if the field can be null, then the ‘Nullable’ encoding method shall be called;

d) the encoding shall be done according to the declared field type:

   1) if the declared field type is a MAL attribute, then the method ‘encode[Nullable]<<Attribute>>’ provided by the MALEncoder shall be called;

   2) if the declared field type is MAL::Attribute, then the method ‘encode[Nullable]Attribute’ provided by the MALEncoder shall be called;

   3) if the declared field type is a MAL enumeration, then the method ‘encode[Nullable]Element’ provided by the MALEncoder shall be called;

   4) otherwise, the declared field shall be a concrete MAL composite and the method ‘encode[Nullable]Element’ provided by the MALEncoder shall be called.

4.5.8.10 Decode

The method ‘decode’ shall be implemented as follows:

a) if the parent type is not MALComposite, then the super ‘decode’ shall be called;

b) the result of the super ‘decode’ shall not be used as it returns ‘this’;

c) every field shall be decoded in the same order as its declaration;

d) if the field can be null, then the ‘nullable’ decoding method shall be called;

e) the decoding shall be done according to the declared field type:

   1) if the declared field type is a MAL attribute, then the method ‘decode[Nullable]<<Attribute>>’ provided by the MALDecoder shall be called;

   2) if the declared field type is MAL::Attribute, then the method ‘decode[Nullable]Attribute’ provided by the MALDecoder shall be called;

   3) if the declared field type is a MAL enumeration, then the method ‘decode[Nullable]Element’ provided by the MALDecoder shall be called with the first item of the enumeration as a parameter;

   4) otherwise the declared field shall be a concrete MAL composite and the method ‘decode[Nullable]Element’ provided by the MALDecoder shall be called with a new instance of the composite as a parameter.

f) the method ‘decode’ shall return ‘this’.

4.5.8.11 Create an Element

The method ‘createElement’ defined by the interface Element shall be implemented by calling the empty constructor and returning the result.

4.5.9 LIST MARKER INTERFACE

4.5.9.1 Interface Definition

4.5.9.1.1 The list marker interface shall declare a generic type variable ‘T’ with the constraint ‘T extends <<Composite>>’.

4.5.9.1.2 If the list element type extends MAL::Composite then the list interface shall extend the CompositeList<T> interface.

4.5.9.1.3 If the list element type extends a type <<parent composite>> then the list interface shall extend the interface <<Parent composite>>List<T>.

4.6 ELEMENT FACTORY CLASSES

4.6.1 OVERVIEW

This subsection specifies how MALElementFactory classes are generated.

4.6.2 NAMESPACE

4.6.2.1 If the data structure belongs to an area, then the namespace shall be:

```
<<root name>>::<<!area!>>::structures::factory
```

4.6.2.2 If the data structure belongs to a service, then the namespace shall be:

```
<<root name>>::<<!area!>>::<<!service!>>::structures::factory
```

4.6.3 CLASS DEFINITION

4.6.3.1 A class shall be defined for each concrete data type and list type.

4.6.3.2 The class name shall be the name of the data type followed by the suffix ‘Factory’: <<Type name>>Factory.

4.6.3.3 The class shall be public and final.

4.6.3.4 The class shall implement the interface MALElementFactory by implementing the method ‘createElement’ in the following way:

a) if the element is a Composite, then the empty constructor shall be called;

b) if the element is an Enumeration, then the first enumeration item shall be returned;

c) if the element is an Attribute and is not mapped to a Union, then the empty constructor shall be called;

d) if the element is an Attribute and is mapped to a Union, then the Union constructor declaring the attribute type shall be called by passing any value.

4.7 MULTIPLE ELEMENT BODY CLASSES

4.7.1 OVERVIEW

This subsection describes the multiple element body classes used on both consumer and provider sides.

4.7.2 NAMESPACE

The namespace shall be:

```
<<root name>>::<<!area!>>::<<!service!>>::body
```

4.7.3 RESPONSE BODY

4.7.3.1 Definition

4.7.3.1.1 A class shall be defined in order to gather the body elements from the RESPONSE message.

4.7.3.1.2 The class name shall be: <<Op>>Response.

4.7.3.1.3 An <<Op>>Response shall be defined for every REQUEST operation with a RESPONSE message that takes more than one argument.

4.7.3.2 Attributes Declaration

A private C++ attribute shall be declared for each body element:

```
<<Res [i]>> bodyElement<<i>>;
...
<<Res [N]>> bodyElement<<N>>;
```

4.7.3.3 Empty Constructor

4.7.3.3.1 An empty public constructor shall be defined.

4.7.3.3.2 The constructor signature shall be:

```
<<Op>>Response()
```

4.7.3.4 Constructor

4.7.3.4.1 A public constructor shall be defined with all the fields owned by this class.

4.7.3.4.2 The constructor signature shall be:

```
<<Op>>Response(<<Res [i]>> bodyElement<<i>>, …
  <<Res [N]>> bodyElement<<N>>)
```

4.7.3.5 Getters

4.7.3.5.1 A getter method ‘getBodyElement<<i>>’ shall be defined for each field.

4.7.3.5.2 The signature of the method ‘getBodyElement<<i>>’ shall be:

```
<<Res [i]>> getBodyElement<<i>>()
```

4.7.3.6 Setters

4.7.3.6.1 A setter method ‘setBodyElement<<i>>’ shall be defined for each field.

4.7.3.6.2 The signature of the method ‘setBodyElement<<i>>’ shall be:

```
void setBodyElement<<i>>(<<Res [i]>> bodyElement)
```

4.7.4 ACK BODY

4.7.4.1 Definition

4.7.4.1.1 A class shall be defined in order to gather the body elements from the ACK message.

4.7.4.1.2 The class name shall be: <<Op>>Ack.

4.7.4.1.3 An <<Op>>Ack shall be defined for every INVOKE or PROGRESS operation with an ACK message that takes more than one argument.

4.7.4.2 Attributes Declaration

A private C++ attribute shall be declared for each body element:

```
private <<Ack [i]>> bodyElement<<i>>;
...
private <<Ack [N]>> bodyElement<<N>>;
```

4.7.4.3 Empty Constructor

4.7.4.3.1 An empty public <<Op>>Ack constructor shall be defined.

4.7.4.3.2 The <<Op>>Ack constructor signature shall be:

```
public <<Op>>Ack()
```

4.7.4.4 Constructor

4.7.4.4.1 A public <<Op>>Ack constructor shall be defined with all the fields owned by this class.

4.7.4.4.2 The <<Op>>Ack constructor signature shall be:

```
<<Op>>Ack(<<Res [i]>> bodyElement<<i>>, …
  <<Res [N]>> bodyElement<<N>>)
```

4.7.4.5 Getters

4.7.4.5.1 A getter method ‘getBodyElement<<i>>’ shall be defined for each field.

4.7.4.5.2 The signature of the method ‘getBodyElement<<i>>’ shall be:

```
<<Ack [i]>> getBodyElement<<i>>()
```

4.7.4.6 Setters

4.7.4.6.1 A setter method ‘setBodyElement<<i>>’ shall be defined for each field.

4.7.4.6.2 The signature of the method ‘setBodyElement<<i>>’ shall be:

```
void setBodyElement<<i>>(<<Ack [i]>> bodyElement)
```

4.8 HELPER AND ELEMENT FACTORY CLASSES

4.8.1 OVERVIEW

This subsection describes the helper and element factory classes used on both consumer and provider sides.

4.8.2 SERVICE HELPER

4.8.2.1 Class Definition

4.8.2.1.1 The service helper namespace shall be:

```
<<root name>>::<<!area!>>::<<!service!>>
```

4.8.2.1.2 The name of the class shall be the identifier of the service suffixed with ‘Helper’: <<Service>>Helper.

4.8.2.2 Service Declaration

4.8.2.2.1 The number of the service shall be declared as follows:

```
static const int _<<SERVICE>>_SERVICE_NUMBER = <<service number>>;
static const uint16_t <<SERVICE>>_SERVICE_NUMBER =
            _<<SERVICE>>_SERVICE_NUMBER;
```

4.8.2.2.2 The name of the service shall be declared as follows:

```
static const Identifier <<SERVICE>>_SERVICE_NAME = "<<service>>";
```

4.8.2.2.3 The MALService shall be declared as follows:

```
shared_ptr<MALService> <<SERVICE>>_SERVICE =
    make_shared<<<mal namespace>>::MALService>(
          <<SERVICE>>_SERVICE_NUMBER,
          <<SERVICE>>_SERVICE_NAME);
```

4.8.2.3 Service Level Errors Declaration

The service level errors shall be declared as follows:

```
static const int _<<ERROR>>_ERROR_NUMBER = <<error number>>;
static const uint32_t _<<ERROR>>_ERROR_NUMBER =
                                    _<<ERROR>>_ERROR_NUMBER;
```

4.8.2.4 Operations Declaration

4.8.2.4.1 The numbers of the operations provided by the service shall be declared as follows:

```
static const int _<<OP>>_OP_NUMBER = <<op number>>;
static const uint16_t <<OP>>_OP_NUMBER = _<<OP>>_OP_NUMBER;
```

4.8.2.4.2 A static MAL<<Ip>>Operation constant shall be created for each operation.

4.8.2.4.3 The last body element short forms array shall be empty.

4.8.2.4.4 The SEND operations shall be declared as follows:

```
shared_ptr<MALSendOperation> <<OP>>_OP =
    make_shared<MALSendOperation>(
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      bool(<<op replayable>>),
      uint16_t(<<capability set>>),
      MALOperationStage(
        (uint16_t) 1,
        vector<int64_t>({<<in short form [i]>>, ... <<in short form [N]>>}),
        vector<int64_t>({})));
```

4.8.2.4.5 The SUBMIT operations shall be declared as follows:

```
shared_ptr<MALSubmitOperation> <<OP>>_OP =
    make_shared<MALSubmitOperation>(
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      (bool)<<op replayable>>,
      (bool)<<capability set>>,
      MALOperationStage(
        (uint16_t) 1,
        vector<int64_t>({<<in short form [i]>>, ... <<in short form [N]>>}),
        vector<int64_t>({})));
```

4.8.2.4.6 The REQUEST operations shall be declared as follows:

```
shared_ptr<MALRequestOperation> <<OP>>_OP =
    make_shared<MALRequestOperation> (
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      (bool)<<op replayable>>,
      (uint16_t)<<capability set>>,
      MALOperationStage(
        (uint16_t) 1,
        vector<int64_t>({<<in short form [i]>>,...<<in short form [N]>>}),
        vector<int64_t>({}),
      MALOperationStage(
        (uint16_t) 2,
        vector<int64_t>({<<res short form [i]>>,...<<res short form [N]>>}),
        vector<int64_t>({})));
```

4.8.2.4.7 The INVOKE operations shall be declared as follows:

```
shared_ptr<MALInvokeOperation> <<OP>>_OP =
    make_shared<MALInvokeOperation>(
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      (bool)<<op replayable>>,
      (uint16_t)<<capability set>>,
      MALOperationStage(
        (uint16_t) 1,
        Vector<int64_t>({<<in short form [i]>>, ... <<in short form [N]>>}),
        vector<int64_t>({})),
      MALOperationStage(
        (uint16_t) 2,
        vector<int64_t>({<<ack short form [i]>>,...<<ack short form [N]>>}),
        vector<int64_t>({})),
      MALOperationStage(
        (uint16_t) 3,
        vector<int64_t>({<<res short form [i]>>,...<<res short form [N]>>}),
        vector<int64_t>({})));
```

4.8.2.4.8 The PROGRESS operations shall be declared as follows:

```
shared_ptr<MALProgressOperation> <<OP>>_OP =
    make_shared<MALProgressOperation> (
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      (bool)<<op replayable>>,
```

```
      (uint16_t)<<capability set>>,
      MALOperationStage(
        (uint16_t) 1,
        vector<int64_t>({<<in short form [i]>>, ... <<in short form [N]>>}),
        vector<int64_t>({})),
      MALOperationStage(
        (uint16_t) 2,
        vector<int64_t>{<<ack short form [i]>>, ... <<ack short form [N]>>},
        vector<int64_t>{}),
      MALOperationStage(
        (uint16_t) 3,
        vector<int64_t>{
          <<update short form [i]>>, ... << update short form [N]>>},
        vector<int64_t>{}),
      MALOperationStage(
        (uint16_t) 4,
        vector<int64_t>{<<res short form [i]>>, ... <<res short form [N]>>},
        vector<int64_t>{}));
```

4.8.2.4.9 The PUBLISH-SUBSCRIBE operations shall be declared as follows:

```
public static final MALPubSubOperation <<OP>>_OP =
    make_shared<MALPubSubOperation>(
      <<OP>>_OP_NUMBER,
      Identifier("<<op>>"),
      (bool)<<op replayable>>,
      (uint16_t)<<capability set>>,
      vector<int64_t>({
        <<notify list short form [i]>>,
        ... << notify list short form [N]>>},
      vector<int64_t>({})) );
```

4.8.2.5 Operation Level Errors Declaration

4.8.2.5.1 For all the operations of the service, a public static inner class called <<Op>>OperationHelper shall be declared.

4.8.2.5.2 All the errors defined at the level of this operation shall be declared as follows:

```
static const int32_t _<<ERROR>>_ERROR_NUMBER = <<error number>>;
static const uint32_t <<ERROR>>_ERROR_NUMBER = _<<ERROR>>_ERROR_NUMBER;
```

4.8.2.6 Initialization

4.8.2.6.1 A method ‘init’ shall be defined in order to initialize the resources required by the usage of this service.

4.8.2.6.2 The signature of the method ‘init’ shall be:

```
static void init(
      const shared_ptr<MALElementFactoryRegistry>& elementFactoryRegistry)
```

4.8.2.6.3 The method ‘init’ shall be idempotent.

4.8.2.6.4 The method ‘init’ shall be called by a MAL client before using the service.

4.8.2.6.5 All the MALOperation constants declared in the helper class shall be added to the MALService by calling the method ‘addOperation’.

4.8.2.6.6 The MALService shall be added to its MALArea by calling the method ‘addService’.

4.8.2.6.7 The MALElementFactoryRegistry shall be assigned to the MALService by calling the method ‘setElementFactoryRegistry’.

4.8.2.6.8 Every MALElementFactory defined in this service shall be registered into the MALElementFactoryRegistry passed as a parameter of the method ‘init’.

4.8.2.6.9 Every error defined in this service shall be registered into the MALContextFactory.

4.8.2.7 Deep Initialization

4.8.2.7.1 A method ‘deepInit’ shall be defined in order to initialize the resources required by the usage of this service and all the areas and services it depends on.

4.8.2.7.2 The signature of the method ‘deepInit’ shall be:

```
static void deepInit(
      const shared_ptr<MALElementFactoryRegistry>& elementFactoryRegistry)
```

4.8.2.7.3 The method ‘deepInit’ shall call the method ‘init’ provided by this helper.

4.8.2.7.4 The method ‘deepInit’ shall call the method ‘deepInit’ provided by the helper of every area and service this service depends on.

4.8.3 AREA HELPER

4.8.3.1 Class Definition

4.8.3.1.1 The area helper namespace shall be:

```
<<root name>>::<<!area!>>
```

4.8.3.1.2 The name of the class shall be the identifier of the area suffixed with ‘Helper’: <<Area>>Helper.

4.8.3.2 Area Declaration

4.8.3.2.1 The number of the area shall be declared as follows:

```
static const int32_t _<<AREA>>_AREA_NUMBER = <<area number>>;
static const uint16_t <<AREA>>_AREA_NUMBER = _<<AREA>>_AREA_NUMBER);
static const uint8_t <<AREA>>_AREA_VERSION = (uint8_t) <<area version>>;
```

4.8.3.2.2 The name of the area shall be declared as follows:

```
static const Identifier <<AREA>>_AREA_NAME = "<<area>>";
```

4.8.3.2.3 The MALArea shall be declared as follows:

```
shared_ptr<<<mal namespace>>::MALArea> <<AREA>>_AREA =
      make_shared<MALArea>(
                  <<AREA>>_AREA_NUMBER,
                  <<AREA>>_AREA_NAME,
                  <<AREA>>_AREA_VERSION);
```

4.8.3.3 Error Declaration

The area level errors shall be declared as follows:

```
static const int32_t _<<ERROR>>_ERROR_NUMBER = <<error number>>;
static const uint32_t _<<ERROR>>_ERROR_NUMBER = _<<ERROR>>_ERROR_NUMBER;
```

4.8.3.4 Initialization

4.8.3.4.1 A method ‘init’ shall be defined in order to initialize the resources required by the usage of this area.

4.8.3.4.2 The signature of the method ‘init’ shall be:

```
static void init(
      const shared_ptr<MALElementFactoryRegistry>& elementFactoryRegistry)
```

4.8.3.4.3 The method ‘init’ shall be idempotent.

4.8.3.4.4 The method ‘init’ shall be called by a MAL client before using the area.

4.8.3.4.5 The MALArea shall be registered in the MALContextFactory by calling the method ‘registerArea’.

4.8.3.4.6 Every MALElementFactory defined in this area shall be registered into the MALElementFactoryRegistry passed as a parameter of the method ‘init’.

4.8.3.4.7 Every error defined in this area shall be registered into the MALContextFactory.

4.8.3.5 Deep Initialization

4.8.3.5.1 A method ‘deepInit’ shall be defined in order to initialize the resources required by the usage of this area and all the areas and services it depends on.

4.8.3.5.2 The signature of the method ‘deepInit’ shall be:

```
static void deepInit(
      const shared_ptr<MALElementFactoryRegistry>& elementFactoryRegistry)
```

4.8.3.5.3 The method ‘deepInit’ shall call the method ‘init’ provided by this helper.

4.8.3.5.4 The method ‘deepInit’ shall call the method ‘deepInit’ provided by the helper of every area and service this area depends on.

5 TRANSPORT API

5.1 GENERAL

The transport API defines the interfaces and classes required by the MAL transport interface defined in reference [1].

The transport API shall be used by the MAL layer in order that any transport module that complies with this API can be plugged into the MAL layer.

5.2 CLASSES AND INTERFACES

5.2.1 GENERAL

The classes and interfaces defined by this API are contained in the namespace:

```
mo::mal::transport
```

5.2.2 MALTRANSPORTFACTORY

5.2.2.1 Definition

5.2.2.1.1 A MALTransportFactory class shall be defined in order to enable the MAL layer to instantiate and configure a MALTransport.

5.2.2.1.2 The MALTransportFactory class shall be abstract.

5.2.2.1.3 The MALTransportFactory class shall be extended by every specific factory class.

5.2.2.1.4 The MALTransportFactory class shall define the attributes specified in table 5-1.

Table 5-1: MALTransportFactory Attribute

Attribute	Type
protocol	String

5.2.2.1.5 The MALTransportFactory class shall provide a static factory class repository that maps the MALTransportFactory implementation classes to their class names.

5.2.2.2 Factory Class Registration

5.2.2.2.1 The MALTransportFactory class shall provide a static method ‘registerFactoryClass’ in order to register the class of a specific MALTransportFactory.

5.2.2.2.2 The signature of the method ‘registerFactoryClass’ shall be:

```
static void registerFactoryClass(
                  const string& protocol,
                  const shared_ptr<MALTransportFactory>& factoryClass)
```

5.2.2.2.3 The parameter of the method ‘registerFactoryClass’ shall be assigned as described in table 5-2.

Table 5-2: MALTransportFactory ‘registerFactoryClass’ Parameters

Parameter	Description
protocol	Name of the protocol to be handled by the instantiated MALTransportFactory
factoryClass	Class that inherits from MALTransportFactory

5.2.2.2.4 The method ‘registerFactoryClass’ shall store the class in the factory class repository.

5.2.2.3 Factory Class Deregistration

5.2.2.3.1 The MALTransportFactory class shall provide a static method ‘deregisterFactoryClass’ in order to deregister the class of a specific MALTransportFactory.

5.2.2.3.2 The signature of the method ‘deregisterFactoryClass’ shall be:

```
static void deregisterFactoryClass(const string& protocol)
```

5.2.2.3.3 The parameter of the method ‘deregisterFactoryClass’ shall be assigned as described in table 5-3.

Table 5-3: MALTransportFactory ‘deregisterFactoryClass’ Parameter

Parameter	Description
protocol	Name of the protocol to be handled by the instantiated MALTransportFactory

5.2.2.3.4 The method ‘deregisterFactoryClass’ shall remove the class from the factory class repository.

5.2.2.3.5 The method ‘deregisterFactoryClass’ shall do nothing if the class is not found in the factory class repository.

5.2.2.4 Constructor

5.2.2.4.1 The MALTransportFactory constructor signature shall be:

```
MALTransportFactory(const string& protocol)
```

5.2.2.4.2 The MALTransportFactory constructor parameter shall be assigned as described in table 5-4.

Table 5-4: MALTransportFactory Constructor Parameter

Parameter	Description
protocol	Name of the protocol to be handled by the instantiated MALTransportFactory

5.2.2.4.3 The attribute ‘protocol’ shall be assigned with the value of the parameter ‘protocol’.

5.2.2.5 Protocol Getter

5.2.2.5.1 A getter method ‘getProtocol’ shall be defined for the attribute ‘protocol’.

5.2.2.5.2 The signature of the method ‘getProtocol’ shall be:

```
string getProtocol()
```

5.2.2.6 MALTransportFactory Creation

5.2.2.6.1 A static method ‘newFactory’ shall be defined in order to create a factory instance from a protocol name.

5.2.2.6.2 The signature of the method ‘newFactory’ shall be:

```
template<typename FactoryType>
shared_ptr<MALTransportFactory> newFactory(const string& protocol)
```

5.2.2.6.3 The parameter of the method ‘newFactory’ shall be assigned as described in table 5-5.

Table 5-5: MALTransportFactory 'newFactory' Parameter

Parameter	Description
protocol	Name of the protocol to be handled by the instantiated MALTransportFactory

5.2.2.6.4 The method 'newFactory' shall resolve the specific MALTransportFactory class name through a 'properties file' that will define the specific factory to handle the transport needed. For example, a protocol class of the form

```
mo::mal::transport::protocol::<<protocol name>>

would have a transport factory defined in a properties file as shown in the
following examples:
```

NOTE – Each of those protocol properties is assigned with the class name of the transport factory. It should be noted that two different protocol properties can share the same transport factory class name. Below is an example of a configuration of protocol properties (the C++ namespace is omitted in the class names):

```
# DDS transport with BINARY encoding
mo.mal.transport.protocol.ddsbin  DdsTransportFactory
# Web Service transport with SOAP encoding
mo.mal.transport.protocol.wssoap   WsTransportFactory
# AMS transport with BINARY encoding
mo.mal.transport.protocol.amsbin   AmsTransportFactory
```

5.2.2.6.5 The method 'newFactory' shall lookup the MALTransportFactory implementation class from the factory class repository.

5.2.2.6.6 If the MALTransportFactory class is not found in the factory class repository, then it shall be created.

5.2.2.6.7 The specific MALTransportFactory class shall be instantiated by calling its constructor that declares a String parameter and passing the protocol name.

5.2.2.6.8 The method 'newFactory' shall not return the value NULL.

5.2.2.6.9 If no MALTransportFactory can be returned, then a MALException shall be raised.

5.2.2.7 MALTransport Instantiation

5.2.2.7.1 The factory class shall provide an abstract public method 'createTransport' to instantiate a MALTransport.

5.2.2.7.2 The signature of the pure virtual method 'createTransport' shall be:

```
shared_ptr<MALTransport> createTransport(
                              const shared_ptr<MALContext>& malContext,
                              const MALQoSProperties& properties) = 0;
```

5.2.2.7.3 The parameter of the method ‘createTransport’ shall be assigned as described in table 5-6.

Table 5-6: MALTransportFactory ‘createTransport’ Parameters

Parameter	Description
malContext	MALContext that owns the MALTransport to create
properties	Configuration properties

5.2.2.7.4 The parameters ‘malContext’ and ‘properties’ may be NULL.

5.2.2.7.5 The method ‘createTransport’ shall not return the value NULL.

5.2.2.7.6 If no MALTransport can be returned, then a MALException shall be raised.

5.2.2.7.7 The method ‘createTransport’ shall be implemented by every specific factory class.

5.2.3 MALTRANSPORT

5.2.3.1 Definition

5.2.3.1.1 A MALTransport interface shall be defined in order to enable the MAL layer to send and receive MAL messages through a single protocol.

5.2.3.1.2 If several protocols are used by the MAL layer, then one MALTransport instance shall be created for each of them.

5.2.3.1.3 Messages shall be sent and received through communication ports represented by the interface MALEndpoint.

5.2.3.1.4 A MALTransport shall be a factory of MALEndpoint.

5.2.3.1.5 One MALEndpoint shall be created for each MALConsumer and MALProvider.

NOTE – If a MALProvider owns a private broker, the MAL layer can create either two different MALEndpoints, one for the service provider and the other for the broker, or a single one for both. In the first case, the service provider and its broker have two different URIs whereas in the second case they share the same one.

5.2.3.2 Create an Endpoint

5.2.3.2.1 A method ‘createEndpoint’ shall be defined in order to instantiate a new MALEndpoint object from two parameters.

5.2.3.2.2 The signature of the method ‘createEndpoint’ shall be:

```
shared_ptr<MALEndpoint> createEndpoint(
                  const string& localName,
                  const MALQoSProperties& qosProperties)
```

5.2.3.2.3 The parameters of the method ‘createEndpoint’ shall be assigned as described in table 5-7.

Table 5-7: MALTransport ‘createEndpoint’ Parameters

Parameter	Description
localName	Name of the endpoint
qosProperties	QoS properties to be used when creating the MALEndpoint

5.2.3.2.4 The parameters ‘localName’ and ‘qosProperties’ may be NULL.

5.2.3.2.5 If the parameter ‘localName’ is not NULL, then its value shall be unique for a given transport.

5.2.3.2.6 If the MALTransport is closed, then a MALException shall be raised.

5.2.3.2.7 If the endpoint local name is not NULL and if the process starts again after a stop and creates the MALEndpoint with the same local name, then the MALEndpoint shall recover the same URI as before the stop.

NOTE – If the name is NULL, then the endpoint URI cannot be recovered.

5.2.3.3 Get an Endpoint

5.2.3.3.1 Two methods ‘getEndpoint’ shall be defined in order to get the reference of a MALEndpoint from:

a) its local name;

b) its URI.

5.2.3.3.2 The signatures of the method ‘getEndpoint’ shall be:

```
shared_ptr<MALEndpoint> getEndpoint(const string& localName)

shared_ptr<MALEndpoint> getEndpoint(const URI& uri)
```

5.2.3.3.3 The parameters of the method ‘getEndpoint’ shall be assigned as described in table 5-8.

Table 5-8: MALTransport ‘getEndpoint’ Parameters

Parameter	Description
localName	Name of the MALEndpoint to get
uri	URI of the MALEndpoint to get

5.2.3.3.4 If the MALTransport is closed, then a MALException shall be raised.

5.2.3.4 Delete an Endpoint

5.2.3.4.1 A method ‘deleteEndpoint’ shall be defined in order to delete an endpoint designated by its local name.

5.2.3.4.2 The signature of the method ‘deleteEndpoint’ shall be:

```
void deleteEndpoint(const string& localName)
```

5.2.3.4.3 The parameter of the method ‘deleteEndpoint’ shall be assigned as described in table 5-9.

Table 5-9: MALTransport ‘deleteEndpoint’ Parameter

Parameter	Description
localName	Name of the MALEndpoint to delete

5.2.3.4.4 If the MALTransport is closed, then a MALException shall be raised.

5.2.3.4.5 After deletion, the endpoint’s URI shall be invalidated.

5.2.3.4.6 If a consumer initiates an interaction with a deleted endpoint, then the error MAL::DESTINATION_UNKNOWN shall be returned to the consumer if it is allowed by the IP.

5.2.3.5 Check QoS Support

5.2.3.5.1 A method ‘isSupportedQoSLevel’ shall be defined in order to indicate whether a QoS level is supported or not.

5.2.3.5.2 The signature of the method ‘isSupportedQoSLevel’ shall be:

```
bool isSupportedQoSLevel(const QoSLevel& qos)
```

5.2.3.5.3 The parameter of the method ‘isSupportedQoSLevel’ shall be assigned as described in table 5-10.

Table 5-10: MALTransport ‘isSupportedQoSLevel’ Parameter

Parameter	Description
qos	QoSLevel which support is to be tested

5.2.3.5.4 The method ‘isSupportedQoSLevel’ shall return TRUE if the specified QoSLevel is supported by the MALTransport; otherwise it shall return FALSE.

5.2.3.5.5 The MAL layer shall use the method ‘isSupportedQoSLevel’ in order to test if a QoS level is supported or not by the MALTransport.

5.2.3.6 Check IP Support

5.2.3.6.1 A method ‘isSupportedInteractionType’ shall be defined in order to indicate whether an IP is supported or not.

5.2.3.6.2 The signature of the method ‘isSupportedInteractionType’ shall be:

```
bool isSupportedInteractionType(const InteractionType& type)
```

5.2.3.6.3 The parameter of the method ‘isSupportedInteractionType’ shall be assigned as described in table 5-11.

Table 5-11: MALTransport ‘isSupportedInteractionType’ Parameter

Parameter	Description
type	The InteractionType which support is to be tested.

5.2.3.6.4 The method ‘isSupportedInteractionType’ shall return TRUE if the specified InteractionType is supported by the MALTransport; otherwise it shall return FALSE.

5.2.3.6.5 The MAL layer shall use the method ‘isSupportedInteractionType’ in order to test if an IP is supported or not by the MALTransport.

5.2.3.7 Create a Transport Level Broker

5.2.3.7.1 Two methods ‘createBroker’ shall be defined in order to allow the creation of a transport level broker:

a) using a private MALEndpoint;

b) using a shared MALEndpoint.

5.2.3.7.2 The signatures of the method ‘createBroker’ shall be:

```
shared_ptr<MALBrokerBinding> createBroker(
      const string& localName,
      const shared_ptr<Blob>& authenticationId,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& defaultQosProperties)

shared_ptr<MALBrokerBinding> createBroker(
      const shared_ptr<MALEndpoint>& endpoint,
      const shared_ptr<Blob>& authenticationId,
      const vector<QoSLevel>& expectedQos,
      const uint32_t& priorityLevelNumber,
      const MALQoSProperties& defaultQosProperties)
```

5.2.3.7.3 The parameters of the method ‘createBroker’ shall be assigned as described in table 5-12.

Table 5-12: MALTransport ‘createBroker’ Parameters

Parameter	Description
localName	Name of the private MALEndpoint to be created and used by the broker
endpoint	Shared MALEndpoint to be used by the broker
authenticationId	Authentication identifier that should be used by the broker
expectedQoS	QoS levels the broker assumes it can rely on
priorityLevelNumber	Number of priorities the broker uses
defaultQoSProperties	Default QoS properties used by the broker to send messages

5.2.3.7.4 The parameters ‘localName’, ‘authenticationId’ and ‘defaultQoSProperties’ may be NULL.

5.2.3.7.5 If the parameter ‘localName’ is not NULL, then its value shall be unique for a given transport.

5.2.3.7.6 If the parameter ‘service’ is NULL, then the broker shall accept to interact through PUBLISH-SUBSCRIBE operations from any services.

5.2.3.7.7 Use of the parameter ‘authenticationId’ is optional.

NOTE – The parameter ‘authenticationId’ is just a hint: some transports might not use this authentication identifier and assign another identifier to the broker.

5.2.3.7.8 The method ‘createBroker’ shall return NULL if no broker can be created by this MALTransport.

5.2.3.7.9 A transport level broker shall be handled by the MALTransport.

5.2.3.7.10 The PUBLISH-SUBSCRIBE interaction shall not be handled at the MAL level: no MALMessage shall be received by the MAL layer for such a broker.

5.2.3.7.11 The transport level broker shall be represented as a MALBrokerBinding.

5.2.3.7.12 If the MALTransport is closed, then a MALException shall be raised.

5.2.3.7.13 If the local name is not NULL and if the process starts again after a stop and creates the MALBrokerBinding with the same local name, then the MALBrokerBinding shall recover the same URI as before the stop.

5.2.3.7.14 If a private MALEndpoint is used, then the message delivery to the broker shall be started as soon as the method returns.

5.2.3.8 Close

5.2.3.8.1 A method ‘close’ shall be defined in order to release all the resources allocated by the MALTransport.

5.2.3.8.2 The signature of the method ‘close’ shall be:

```
void close()
```

5.2.3.8.3 If an internal error occurs, then a MALException shall be raised.

5.2.4 MALENDPOINT

5.2.4.1 Definition

5.2.4.1.1 A MALEndpoint interface shall be defined in order to send and receive MALMessages.

5.2.4.1.2 A new MALEndpoint shall be created when the MAL layer needs to allocate a new URI. This happens in the following cases:

a) creation of a MALConsumer;

b) creation of a MALProvider;

c) creation of a MALBrokerBinding owned by a MALBroker.

5.2.4.1.3 In those cases, the URI of the MALConsumer, the MALProvider, or the MALBrokerBinding shall be the one owned by its MALEndpoint.

5.2.4.1.4 A MALEndpoint shall be a MALMessage factory.

5.2.4.1.5 The MALMessages sent by a MALConsumer, a MALProvider, or a MALBrokerBinding shall be sent through their MALEndpoint: the field 'URIfrom' belonging to the header of the MALMessages shall be assigned with the MALEndpoint URI.

5.2.4.1.6 A MALEndpoint shall be able to send messages as soon as it has been created.

5.2.4.1.7 A method 'startMessageDelivery' shall be defined in order to explicitly start the message delivery.

5.2.4.1.8 If a message is received by a MALEndpoint which message's delivery is not started,

a) and if the QoS level is QUEUED, then the message shall be queued and delivered as soon as the delivery is started;

b) otherwise, if the interaction allows to return an error message, then a DELIVERY_FAILED error shall be returned to the sending endpoint.

5.2.4.2 Get the URI

5.2.4.2.1 A getter method 'getURI' shall be defined in order to return the URI of the MALEndpoint.

5.2.4.2.2 The signature of the method 'getURI' shall be:

```
URI getURI()
```

5.2.4.3 Get the Local Name

5.2.4.3.1 A getter method 'getLocalName' shall be defined in order to return the local name of the MALEndpoint.

5.2.4.3.2 The signature of the method 'getLocalName' shall be:

```
string getLocalName()
```

5.2.4.4 Create a Message

5.2.4.4.1 Four methods ‘createMessage’ shall be defined in order to instantiate a new MALMessage object:

a) declaring parameters for the interaction type, the area, the area version, the service, the operation and the body elements;

b) declaring parameters for the interaction type, the area, the area version, the service, the operation and the encoded body;

c) declaring a MALOperation parameter and parameters for the body elements;

d) declaring a MALOperation parameter and a parameter for the encoded body.

5.2.4.4.2 The signatures of the method ‘createMessage’ shall be:

```
shared_ptr<MALMessage> createMessage(
  const shared_ptr<Blob>& authenticationId,
  const URI& uriTo,
  const Time& timestamp,
  const QoSLevel& qosLevel,
  const uint32_t& priority,
  const IdentifierList& domain,
  const Identifier& networkZone,
  const SessionType& session,
  const Identifier& sessionName,
  const InteractionType& interactionType,
  const uint8_t& interactionStage,
  const int64_t& transactionId,
  const uint16_t& serviceAreaNumber,
  const uint16_t& serviceNumber,
  const uint16_t& operationNumber,
  const uint8_t& areaVersion,
  const bool& isErrorMessage,
  const MALQoSProperties& qosProperties,
  const vector<shared_ptr<MALMessageBody>>& body)
```

```
shared_ptr<MALMessage> createMessage(
  const shared_ptr<Blob>& authenticationId,
  const URI& uriTo,
  const Time& timestamp,
  const QoSLevel& qosLevel,
  const uint32_t& priority,
  const IdentifierList& domain,
  const Identifier& networkZone,
  const SessionType& session,
  const Identifier& sessionName,
  const InteractionType& interactionType,
  const uint8_t& interactionStage,
  const int64_t& transactionId,
  const uint16_t& serviceAreaNumber,
  const uint16_t& serviceNumber,
  const uint16_t& operationNumber,
  const uint8_t& areaVersion,
  const bool& isErrorMessage,
  const MALQoSProperties& qosProperties,
  const shared_ptr<MALEncodedBody>& encodedBody)

shared_ptr<MALMessage> createMessage(
  const shared_ptr<Blob>& authenticationId,
  const URI& uriTo,
  const Time& timestamp,
  const QoSLevel& qosLevel,
  const uint32_t& priority,
  const IdentifierList& domain,
  const Identifier& networkZone,
  const SessionType& session,
  const Identifier& sessionName,
  const int64_t& transactionId,
  const bool& isErrorMessage,
  const shared_ptr<MALOperation>& operation,
  const uint8_t& interactionStage,
  const MALQoSProperties& qosProperties,
  const vector<shared_ptr<MALMessageBody>>& body)

shared_ptr<MALMessage> createMessage(
  const shared_ptr<Blob>& authenticationId,
  const URI& uriTo,
  const Time& timestamp,
  const QoSLevel& qosLevel,
  const uint32_t& priority,
  const IdentifierList& domain,
  const Identifier& networkZone,
  const SessionType& session,
  const Identifier& sessionName,
  const int64_t& transactionId,
  const bool& isErrorMessage,
  const shared_ptr<MALOperation>& operation,
  const uint8_t& interactionStage,
  const MALQoSProperties& qosProperties,
  const shared_ptr<MALEncodedBody>& encodedBody)
```

5.2.4.4.3 The parameters of the method ‘createMessage’ shall be assigned as described in table 5-13.

Table 5-13: MALEndpoint 'createMessage' Parameters

Parameter	Description
authenticationId	Authentication identifier of the message
uriTo	URI of the message destination
timestamp	Timestamp of the message
qosLevel	QoS level of the message
priority	Priority of the message
domain	Domain of the service provider
networkZone	Network zone of the service provider
session	Session of the service provider
sessionName	Session name of the service provider
interactionType	Interaction type of the operation
interactionStage	Interaction stage of the interaction
transactionId	Transaction identifier of the interaction
serviceAreaNumber	Area number of the service
serviceNumber	Service number
operationNumber	Operation number
areaVersion	Area version number
operation	Operation represented as a MALOperation
isErrorMessage	Flag indicating if the message conveys an error
qosProperties	QoS properties of the message
body	Message body elements
encodedBody	Encoded body of the message

5.2.4.4.4 The parameters 'body', 'encodedBody', 'transactionId', and 'qosProperties' may be NULL.

5.2.4.4.5 The body encoding shall be performed by the transport layer.

5.2.4.4.6 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.5 Send a Message

5.2.4.5.1 A method ‘sendMessage’ shall be provided in order to send a MALMessage.

5.2.4.5.2 The signature of the method ‘sendMessage’ shall be:

```
void sendMessage(const shared_ptr<MALMessage>& msg)
```

5.2.4.5.3 The parameter of the method ‘sendMessage’ shall be assigned as described in table 5-14.

Table 5-14: MALEndpoint ‘sendMessage’ Parameter

Parameter	Description
msg	Message to be sent

5.2.4.5.4 If a TRANSMIT ERROR occurs, then a MALTransmitErrorException shall be raised.

5.2.4.5.5 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.6 Send Multiple Messages

5.2.4.6.1 A method ‘sendMessages’ shall be defined in order to send a list of MALMessages.

5.2.4.6.2 The signature of the method ‘sendMessages’ shall be:

```
void sendMessages(const vector<shared_ptr<MALMessage>>& msgList)
```

5.2.4.6.3 The parameter of the method ‘sendMessages’ shall be assigned as described in table 5-15.

Table 5-15: MALEndpoint ‘sendMessages’ Parameter

Parameter	Description
msgList	List of messages to be sent

5.2.4.6.4 If a MULTIPLETRANSMIT ERROR occurs, then a MALTransmitErrorException shall be raised.

5.2.4.6.5 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.7 Listen to Delivered Messages

5.2.4.7.1 A method ‘setMessageListener’ shall be defined in order to listen to delivered messages.

5.2.4.7.2 The signature of the method ‘setMessageListener’ shall be:

```
void setMessageListener(const shared_ptr<MALMessageListener>& listener)
```

5.2.4.7.3 The parameter of the method ‘setMessageListener’ shall be assigned as described in table 5-16.

Table 5-16: MALEndpoint ‘setMessageListener’ Parameter

Parameter	Description
listener	MALMessageListener in charge of receiving the MALMessages

5.2.4.7.4 The parameter ‘listener’ shall not be NULL.

5.2.4.7.5 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.8 Start Message Delivery

5.2.4.8.1 A method ‘startMessageDelivery’ shall be defined in order to start the message delivery.

5.2.4.8.2 The signature of the method ‘startMessageDelivery’ shall be:

```
void startMessageDelivery()
```

5.2.4.8.3 As soon as the method ‘startMessageDelivery’ has returned, the messages shall be delivered through the MALMessageListener.

5.2.4.8.4 If no MALMessageListener has been set, then a DELIVERY_FAILED error shall be returned to the sending endpoint.

5.2.4.8.5 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.9 Stop Message Delivery

5.2.4.9.1 A method ‘stopMessageDelivery’ shall be defined in order to stop the message delivery.

5.2.4.9.2 The signature of the method ‘stopMessageDelivery’ shall be:

```
void stopMessageDelivery()
```

5.2.4.9.3 As soon as the method ‘stopMessageDelivery’ has returned, no message shall be delivered through the MALMessageListener.

5.2.4.9.4 The method ‘stopMessageDelivery’ shall synchronously stop the message delivery.

5.2.4.9.5 If the MALMessageListener is handling a message at the time of the deactivation, then the method ‘stopMessageDelivery’ shall return after the end of the message handling.

5.2.4.9.6 If the MALEndpoint is closed, then a MALException shall be raised.

5.2.4.10 Close

5.2.4.10.1 A method ‘close’ shall be defined in order to release the resources owned by a MALEndpoint.

5.2.4.10.2 The signature of the method ‘close’ shall be:

```
void close()
```

5.2.4.10.3 The method ‘close’ shall deactivate the MALEndpoint message delivery.

5.2.4.10.4 The method ‘close’ shall be called by the MAL layer when a MALConsumer or a MALProvider is closed.

5.2.4.10.5 A MALEndpoint which local name is NULL shall be deleted.

NOTE – If the MALEndpoint owns a non-null name, then it can be recovered after a close; i.e., it can be created again with the same name. The new MALEndpoint instance will be designated with the same URI.

5.2.4.10.6 If an internal error occurs, then a MALException shall be raised.

5.2.5 MALMESSAGEHEADER

5.2.5.1 Definition

A MALMessageHeader interface shall be defined in order to give a generic access to the MAL message header.

5.2.5.2 Field Getter and Setter

5.2.5.2.1 A getter and a setter shall be defined for each field of the MAL message header as defined in reference [1].

5.2.5.2.2 The name of each field shall be built from the name given in reference [1] by removing the spaces.

5.2.5.2.3 The getter of ‘Is Error Message’ shall be called ‘getIsErrorMessage’.

5.2.6 MALMESSAGEBODY

5.2.6.1 Definition

A MALMessageBody interface shall be defined in order to give a generic access to the MAL message body.

5.2.6.2 Get the Number of Body Elements

5.2.6.2.1 A method ‘getElementCount’ shall be defined in order to return the number of elements contained in the MAL message body.

5.2.6.2.2 The signature of the method ‘getElementCount’ shall be:

```
int getElementCount()
```

5.2.6.3 Get a Body Element

5.2.6.3.1 A method ‘getBodyElement’ shall be defined in order to get a body element.

5.2.6.3.2 The signature of the method ‘getBodyElement’ shall be:

```
shared_ptr<Element> getBodyElement(
      int index,
      const shared_ptr<Element>& element)
```

5.2.6.3.3 The parameters of the method ‘getBodyElement’ shall be assigned as described in table 5-17.

Table 5-17: MALMessageBody ‘getBodyElement’ Parameters

Parameter	Description
index	Index of the element in the body
element	Element to be decoded

5.2.6.3.4 If an error occurs, then a MALException shall be raised.

5.2.6.3.5 The index values shall start from 0, which designates the first element of the body.

5.2.6.3.6 The returned instance may not be the same as the parameter ‘element’.

5.2.6.4 Get an Encoded Body Element

5.2.6.4.1 A method ‘getEncodedBodyElement’ shall be defined in order to get an encoded body element.

5.2.6.4.2 The signature of the method ‘getEncodedBodyElement’ shall be:

```
shared_ptr<MALEncodedElement> getEncodedBodyElement(int index)
```

5.2.6.4.3 The parameters of the method ‘getEncodedBodyElement’ shall be assigned as described in table 5-18.

Table 5-18: MALMessageBody ‘getEncodedBodyElement’ Parameter

Parameter	Description
index	Index of the element in the body

5.2.6.4.4 If an error occurs, then a MALException shall be raised.

5.2.6.4.5 The index values shall start from 0, which designates the first element of the body.

5.2.6.5 Get the Encoded Body

5.2.6.5.1 A method ‘getEncodedBody’ shall be defined in order to get the whole encoded body of the message.

5.2.6.5.2 The signature of the method ‘getEncodedBody’ shall be:

```
shared_ptr<MALEncodedBody> getEncodedBody()
```

5.2.6.5.3 If an error occurs, then a MALException shall be raised.

5.2.7 MALERRORBODY

5.2.7.1 Definition

5.2.7.1.1 A MALErrorBody interface shall be defined in order to give access to the body of an ERROR message, i.e., a MAL message which header field ‘isError’ is TRUE.

5.2.7.1.2 The MALErrorBody interface shall extend MALMessageBody.

5.2.7.2 Get the Error

5.2.7.2.1 A method ‘getError’ shall be defined in order to return the error number from the ERROR message.

5.2.7.2.2 The signature of the method ‘getError’ shall be:

```
shared_ptr<MALStandardError> getError()
```

5.2.7.2.3 If an error occurs, then a MALException shall be raised.

5.2.8 MALREGISTERBODY

5.2.8.1 Definition

5.2.8.1.1 A MALRegisterBody interface shall be defined in order to give access to the body of the REGISTER message defined by the IP PUBLISH-SUBSCRIBE.

5.2.8.1.2 The MALRegisterBody interface shall extend MALMessageBody.

5.2.8.2 Get the Subscription

5.2.8.2.1 A method ‘getSubscription’ shall be defined in order to return the Subscription from the REGISTER message.

5.2.8.2.2 The signature of the method ‘getSubscription’ shall be:

```
shared_ptr<Subscription> getSubscription()
```

5.2.8.2.3 If an error occurs, then a MALException shall be raised.

5.2.9 MALPUBLISHREGISTERBODY

5.2.9.1 Definition

5.2.9.1.1 A MALPublishRegisterBody interface shall be defined in order to give access to the body of the PUBLISH REGISTER message defined by the IP PUBLISH-SUBSCRIBE.

5.2.9.1.2 The MALPublishRegisterBody interface shall extend MALMessageBody.

5.2.9.2 Get the EntityKeyList

5.2.9.2.1 A method ‘getEntityKeyList’ shall be defined in order to return the EntityKeyList from the PUBLISH REGISTER message.

5.2.9.2.2 The signature of the method ‘getEntityKeyList’ shall be:

```
shared_ptr<EntityKeyList> getEntityKeyList()
```

5.2.9.2.3 If an error occurs, then a MALException shall be raised.

5.2.10 MALPUBLISHBODY

5.2.10.1 Definition

5.2.10.1.1 A MALPublishBody interface shall be defined in order to give access to the body of the PUBLISH message defined by the PUBLISH-SUBSCRIBE interaction.

5.2.10.1.2 The MALPublishBody interface shall extend MALMessageBody.

5.2.10.2 Get the List of UpdateHeader

5.2.10.2.1 A method ‘getUpdateHeaderList’ shall be defined in order to return the list of UpdateHeaders from the message.

5.2.10.2.2 The signature of the method ‘getUpdateHeaderList’ shall be:

```
shared_ptr<UpdateHeaderList> getUpdateHeaderList()
```

5.2.10.2.3 If an error occurs, then a MALException shall be raised.

5.2.10.3 Get the Update Lists

5.2.10.3.1 A method ‘getUpdateLists’ shall be defined in order to return the update lists from the message.

5.2.10.3.2 The signature of the method ‘getUpdateLists’ shall be:

```
UpdateLists getUpdateLists(const UpdateLists& updateLists)
```

5.2.10.3.3 The parameter of the method ‘getUpdateLists’ shall be assigned as described in table 5-19.

Table 5-19: MALPublishBody ‘getUpdateLists’ Parameter

Parameter	Description
updateLists	Update lists to decode

5.2.10.3.4 If an error occurs, then a MALException shall be raised.

5.2.10.4 Get an Update List

5.2.10.4.1 A method ‘getUpdateList’ shall be defined in order to return an update list of from the message.

5.2.10.4.2 The signature of the method ‘getUpdateList’ shall be:

```
UpdateList getUpdateList(int listIndex, const UpdateLists& updateLists)
```

5.2.10.4.3 The parameter of the method ‘getUpdateList’ shall be assigned as described in table 5-20.

Table 5-20: MALPublishBody ‘getUpdateList’ Parameters

Parameter	Description
listIndex	Index of the update list
updateLists	Update lists to decode

5.2.10.4.4 The parameter ‘listIndex’ values shall start from 0, which designates the first list of the body.

5.2.10.4.5 If an error occurs, then a MALException shall be raised.

5.2.10.5 Get the Number of Updates

5.2.10.5.1 A method ‘getUpdateCount’ shall be defined in order to return the number of UpdateHeader elements from the message.

5.2.10.5.2 The signature of the method ‘getUpdateCount’ shall be:

```
int getUpdateCount()
```

5.2.10.5.3 If an error occurs, then a MALException shall be raised.

5.2.10.6 Get an Update

5.2.10.6.1 A method ‘getUpdate’ shall be defined in order to return an <<Update>> from the message.

5.2.10.6.2 The signature of the method ‘getUpdate’ shall be:

```
shared_ptr<Element> getUpdate(int listIndex, int updateIndex)
```

5.2.10.6.3 The parameters of the method ‘getUpdate’ shall be assigned as described in table 5-21.

Table 5-21: MALPublishBody 'getUpdate' Parameters

Parameter	Description
listIndex	Index of the update list
updateIndex	Index of the update

5.2.10.6.4 The parameter 'listIndex' values shall start from 0, which designates the first list of the body.

5.2.10.6.5 The parameter 'updateIndex' values shall start from 0, which designates the first update of the list.

5.2.10.6.6 If an error occurs, then a MALException shall be raised.

5.2.10.7 Get an Encoded Update

5.2.10.7.1 A method 'getEncodedUpdate' shall be defined in order to return an encoded <<Update>> from the message.

5.2.10.7.2 The signature of the method 'getEncodedUpdate' shall be:

```
shared_ptr<MALEncodedElement> getEncodedUpdate(
                                    int listIndex, int updateIndex)
```

5.2.10.7.3 The parameters of the method 'getEncodedUpdate' shall be assigned as described in table 5-22.

Table 5-22: MALPublishBody 'getEncodedUpdate' Parameters

Parameter	Description
listIndex	Index of the update list
updateIndex	Index of the update

5.2.10.7.4 The parameter 'listIndex' values shall start from 0, which designates the first list of the body.

5.2.10.7.5 The parameter 'updateIndex' values shall start from 0, which designates the first update of the list.

5.2.10.7.6 If an error occurs, then a MALException shall be raised.

5.2.10.7.7 The method 'getEncodedUpdate' may raise a MALException if the transport encoding format does not enable separate decoding of the updates.

5.2.11 MALNOTIFYBODY

5.2.11.1 Definition

5.2.11.1.1 A MALNotifyBody interface shall be defined in order to give access to the body of the NOTIFY message defined by the PUBLISH-SUBSCRIBE interaction.

5.2.11.1.2 The MALNotifyBody interface shall extend MALPublishBody.

5.2.11.2 Get the Subscription Identifier

5.2.11.2.1 A method ‘getSubscriptionId’ shall be defined in order to return the identifier of the subscription from this NOTIFY message.

5.2.11.2.2 The signature of the method ‘getSubscriptionId’ shall be:

```
Identifier getSubscriptionId()
```

5.2.11.2.3 If an error occurs, then a MALException shall be raised.

5.2.12 MALDEREGISTERBODY

5.2.12.1 Definition

5.2.12.1.1 A MALDeregisterBody interface shall be defined in order to give access to the body of the DEREGISTER message defined by the IP PUBLISH-SUBSCRIBE.

5.2.12.1.2 The MALDeregisterBody interface shall extend MALMessageBody.

5.2.12.2 Get the IdentifierList

5.2.12.2.1 A method ‘getIdentifierList’ shall be defined in order to return the IdentifierList from the DEREGISTER message.

5.2.12.2.2 The signature of the method ‘getIdentifierList’ shall be:

```
IdentifierList getIdentifierList()
```

5.2.12.2.3 If an error occurs, then a MALException shall be raised.

5.2.13 MALENCODEDELEMENT

5.2.13.1 Definition

5.2.13.1.1 A MALEncodedElement class shall be defined in order to represent an encoded element.

5.2.13.1.2 The MALEncodedElement class shall only be used to represent:

a) an encoded body element;

b) an encoded element contained in a body element typed as a list.

5.2.13.1.3 The type MALEncodedElementList shall be used when representing a body element typed as a list and containing encoded elements.

5.2.13.2 Creation

5.2.13.2.1 A MALEncodedElement public constructor shall be defined with a Blob parameter.

5.2.13.2.2 The MALEncodedElement constructor signature shall be:

```
MALEncodedElement(const shared_ptr<Blob>& encodedElement)
```

5.2.13.2.3 The constructor parameter shall be assigned as described in table 5-23.

Table 5-23: MALEncodedElement Constructor Parameter

Parameter	Description
encodedElement	Encoded element

5.2.13.3 Getter

5.2.13.3.1 A method ‘getEncodedElement’ shall be defined in order to get the encoded element.

5.2.13.3.2 The signature of the method ‘getEncodedElement’ shall be:

```
shared_ptr<Blob> getEncodedElement()
```

5.2.14 MALMESSAGE

5.2.14.1 Definition

A MALMessage interface shall be defined in order to give a generic access to the transport specific messages.

5.2.14.2 Accessors

5.2.14.2.1 Getter methods ‘getHeader’, ‘getBody’, and ‘getQoSProperties’ shall be defined in order to give access to the header, the body and the QoS properties of the message.

5.2.14.2.2 The signatures of methods ‘getHeader’, ‘getBody’, and ‘getQoSProperties’ shall be:

```
shared_ptr<MALMessageHeader> getHeader()
shared_ptr<MALMessageBody> getBody()
MALQoSProperties getQoSProperties()
```

5.2.14.2.3 If header field ‘isError’ is TRUE, then the message ‘getBody’ shall return a MALErrorBody.

5.2.14.2.4 If the interaction type is PUBLISH-SUBSCRIBE and the stage is REGISTER, then the message ‘getBody’ shall return a MALRegisterBody.

5.2.14.2.5 If the interaction type is PUBLISH-SUBSCRIBE and the stage is PUBLISH REGISTER, then the message ‘getBody’ shall return a MALPublishRegisterBody.

5.2.14.2.6 If the interaction type is PUBLISH-SUBSCRIBE and the stage is PUBLISH, then the message ‘getBody’ shall return a MALPublishBody.

5.2.14.2.7 If the interaction type is PUBLISH-SUBSCRIBE and the stage is NOTIFY, then the message ‘getBody’ shall return a MALNotifyBody.

5.2.14.2.8 If the interaction type is PUBLISH-SUBSCRIBE and the stage is DEREGISTER, then the message ‘getBody’ shall return a MALDeregisterBody.

5.2.14.3 Free

5.2.14.3.1 A method ‘free’ shall be defined in order to enable the transport to free resources owned by the MALMessage.

5.2.14.3.2 The signature of the method ‘free’ shall be:

```
virtual void free()
```

5.2.14.3.3 The method ‘free’ shall be called by the MAL layer as soon as the MALMessage is no longer used.

5.2.14.3.4 If an internal error occurs, then a MALException shall be raised.

5.2.15 MALMESSAGELISTENER

5.2.15.1 Definition

5.2.15.1.1 A MALMessageListener interface shall be defined in order to enable the MAL layer to be notified by the transport module when:

a) a MALMessage has been received by a MALEndpoint;

b) a TRANSMIT ERROR has been asynchronously raised by the transport layer;

c) an asynchronous internal error has been raised by the transport layer as a consequence of a severe failure making the transport unable to work.

5.2.15.1.2 The interface MALMessageListener shall extend the interface MALTransmitErrorListener.

5.2.15.2 Receive a Message

5.2.15.2.1 A method ‘onMessage’ shall be defined in order to receive a message.

5.2.15.2.2 The signature of the method ‘onMessage’ shall be:

```
void onMessage(
            const shared_ptr<MALEndpoint>& callingEndpoint,
            const shared_ptr<MALMessage>& msg)
```

5.2.15.2.3 The parameters of the method ‘onMessage’ shall be assigned as described in table 5-24.

Table 5-24: MALMessageListener ‘onMessage’ Parameters

Parameter	Description
callingEndpoint	MALEndpoint calling the MALMessageListener
msg	Message received by the listener

5.2.15.3 Receive Multiple Messages

5.2.15.3.1 A method ‘onMessages’ shall be defined in order to receive multiple messages.

5.2.15.3.2 The signature of the method ‘onMessages’ shall be:

```
void onMessages(
            const shared_ptr<MALEndpoint>& callingEndpoint,
            const vector<shared_ptr<MALMessage>>& msgList)
```

5.2.15.3.3 The parameters of the method ‘onMessages’ shall be assigned as described in table 5-25.

Table 5-25: MALMessageListener ‘onMessages’ Parameters

Parameter	Description
callingEndpoint	MALEndpoint calling the MALMessageListener
msgList	List of the messages received by the listener

5.2.15.4 Severe Internal Error

5.2.15.4.1 A method ‘onInternalError’ shall be defined in order to receive an internal error.

5.2.15.4.2 The signature of the method ‘onInternalError’ shall be:

```
void onInternalError(
            const shared_ptr<MALEndpoint>& callingEndpoint,
            const std::exception& error)
```

5.2.15.4.3 The parameters of the method ‘onInternalError’ shall be assigned as described in table 5-26.

Table 5-26: MALMessageListener ‘onInternalError’ Parameters

Parameter	Description
callingEndpoint	MALEndpoint calling the MALMessageListener
error	Error exception to be received by the listener

5.2.16 MALTRANSMITERROREXCEPTION

5.2.16.1 Definition

5.2.16.1.1 A MALTransmitErrorException class shall be defined in order to raise a TRANSMIT ERROR as an exception.

5.2.16.1.2 The MALTransmitErrorException class shall extend the MALInteractionException class.

5.2.16.2 Constructor

5.2.16.2.1 A MALTransmitErrorException constructor shall be defined.

5.2.16.2.2 The MALTransmitErrorException constructor signature shall be:

```
MALTransmitErrorException(
            const shared_ptr<MALMessageHeader>& header,
            const shared_ptr<MALStandardError>& standardError,
            const MALQoSProperties& qosProperties)
```

5.2.16.2.3 The MALTransmitErrorException constructor parameters shall be assigned as described in table 5-27.

Table 5-27: MALTransmitErrorException Constructor Parameters

Parameter	Description
header	Header of the MALMessage that cannot be transmitted
standardError	Error preventing the message to be transmitted
qosProperties	QoS properties of the MALMessage which cannot be transmitted

5.2.16.2.4 The MALTransmitErrorException constructor shall call the MALInteractionException constructor and pass the MALStandardError parameter.

5.2.16.3 Get the Header

5.2.16.3.1 A getter method ‘getHeader’ shall be defined in order to return the header of the MALMessage which cannot be transmitted.

5.2.16.3.2 The signature of the method ‘getHeader’ shall be:

```
shared_ptr<MALMessageHeader> getHeader()
```

5.2.16.4 Get the QoS Properties

5.2.16.4.1 A getter method ‘getQosProperties’ shall be defined in order to return the QoS properties of the MALMessage which cannot be transmitted.

5.2.16.4.2 The signature of the method ‘getQosProperties’ shall be:

```
MALQoSProperties getQosProperties()
```

5.2.17 MALTRANSMITMULTIPLEERROREXCEPTION

5.2.17.1 Definition

5.2.17.1.1 A MALTransmitMultipleErrorException class shall be defined in order to raise a TRANSMITMULTIPLE ERROR as an exception.

5.2.17.1.2 The MALTransmitMultipleErrorException class shall extend the class MALException.

5.2.17.2 Constructor

5.2.17.2.1 A MALTransmitMultipleErrorException constructor shall be defined.

5.2.17.2.2 The MALTransmitMultipleErrorException constructor signature shall be:

```
MALTransmitMultipleErrorException(
  const vector<shared_ptr<MALTransmitErrorException>>& transmitExceptions)
```

5.2.17.2.3 The MALTransmitMultipleErrorException constructor parameter shall be assigned as described in table 5-28.

Table 5-28: MALTransmitMultipleErrorException Constructor Parameter

Parameter	Description
transmitExceptions	Transmit errors preventing messages to be transmitted

5.2.17.3 Get the Transmit Errors

5.2.17.3.1 A getter method ‘getTransmitExceptions’ shall be defined in order to return the Transmit errors raised by the MALMessages which Transmit failed.

5.2.17.3.2 The signature of the method ‘getTransmitExceptions’ shall be:

```
vector<shared_ptr<MALTransmitErrorException>> getTransmitExceptions()
```

5.2.18 MALENCODEDELEMENTLIST

5.2.18.1 Definition

5.2.18.1.1 A MALEncodedElementList class shall be defined in order to represent a list of encoded elements.

5.2.18.1.2 The MALEncodedElementList class shall inherit from the C++ std::vector class with the template type MALEncodedElement.

5.2.18.2 Creation

5.2.18.2.1 A MALEncodedElementList public constructor shall be defined.

5.2.18.2.2 The MALEncodedElementList constructor signature shall be:

```
MALEncodedElementList(long long shortForm, int initialCapacity)
```

5.2.18.2.3 The constructor parameters shall be assigned as described in table 5-29.

Table 5-29: MALEncodedElementList Constructor Parameters

Parameter	Description
shortForm	Short form of the list type
initialCapacity	Initial capacity of the list

5.2.18.3 Get the Short Form

5.2.18.3.1 A method ‘getShortForm’ shall be defined in order to get the short form of the list.

5.2.18.3.2 The signature of the method ‘getShortForm’ shall be:

```
long long getShortForm()
```

5.2.19 MALENCODEDBODY

5.2.19.1 Definition

5.2.19.1.1 A MALEncodedBody class shall be defined in order to represent the encoded body of a message.

5.2.19.2 Creation

5.2.19.2.1 A MALEncodedBody public constructor shall be defined with a Blob parameter.

5.2.19.2.2 The MALEncodedBody constructor signature shall be:

```
MALEncodedBody(const shared_ptr<Blob>& encodedBody)
```

5.2.19.2.3 The constructor parameter shall be assigned as described in table 5-30.

Table 5-30: MALEncodedBody Constructor Parameter

Parameter	Description
encodedBody	Encoded body of a message

5.2.19.3 Getter

5.2.19.3.1 A method ‘getEncodedBody’ shall be defined in order to get the encoded body.

5.2.19.3.2 The signature of the method ‘getEncodedBody’ shall be:

```
shared_ptr<Blob> getEncodedBody()
```

5.2.20 MALTRANSMITERRORLISTENER

5.2.20.1 Definition

A MALTransmitErrorListener interface shall be defined in order to be notified when a TRANSMIT ERROR has been asynchronously raised and cannot be returned as a MAL message.

5.2.20.2 Asynchronous TRANSMIT ERROR

5.2.20.2.1 A method ‘onTransmitError’ shall be defined in order to receive an asynchronous TRANSMIT ERROR.

5.2.20.2.2 The signature of the method ‘onTransmitError’ shall be:

```
void onTransmitError(
                  const shared_ptr<MALEndpoint>& callingEndpoint,
                  const shared_ptr<MALMessageHeader>& header,
                  const shared_ptr<MALStandardError>& standardError,
                  const MALQoSProperties& qosProperties)
```

5.2.20.2.3 The parameters of the method ‘onTransmitError’ shall be assigned as described in table 5-31.

Table 5-31: MALTransmitErrorListener 'onTransmitError' Parameters

Parameter	Description
callingEndpoint	MALEndpoint that sent the message that cannot be transmitted
header	Header of the MALMessage that cannot be transmitted
standardError	Error preventing the message to be transmitted
qosProperties	QoS properties of the MALMessage that cannot be transmitted

6 ACCESS CONTROL API

6.1 GENERAL

The access control API defines the interfaces and classes required by the MAL access control interface defined in reference [1].

The access control API shall be used by the MAL layer in order that any access control module that complies with this API can be plugged into the MAL layer.

6.2 CLASSES AND INTERFACES

6.2.1 GENERAL

The classes and interfaces of the access control API are contained in the namespace:

```
mo::mal::accesscontrol
```

6.2.2 MALACCESSCONTROLFACTORY

6.2.2.1 Definition

6.2.2.1.1 A MALAccessControlFactory class shall be defined in order to enable the MAL layer to create and configure MALAccessControl instances.

6.2.2.1.2 The MALAccessControlFactory class shall be abstract.

6.2.2.1.3 The MALAccessControlFactory class shall be extended by every specific factory class.

6.2.2.1.4 The MALAccessControlFactory class shall provide a static factory class repository that maps the MALAccessControlFactory implementation classes to their class names.

6.2.2.2 Factory Class Registration

6.2.2.2.1 The MALAccessControlFactory class shall provide a static method ‘registerFactoryClass’ in order to register the class of a specific MALAccessControlFactory.

6.2.2.2.2 The signature of the method ‘registerFactoryClass’ shall be:

```
static void registerFactoryClass(
            const shared_ptr<MALAccessControlFactory>& factoryClass)
```

6.2.2.2.3 The parameters of the method ‘registerFactoryClass’ shall be assigned as described in table 6-1.

Table 6-1: MALAccessControlFactory 'registerFactoryClass' Parameter

Parameter	Description
factoryClass	Class extending MALAccessControlFactory

6.2.2.2.4 The method 'registerFactoryClass' shall store the class in the factory class repository.

6.2.2.3 Factory Class Deregistration

6.2.2.3.1 The MALAccessControlFactory class shall provide a static method 'deregisterFactoryClass' in order to deregister the class of a specific MALAccessControlFactory.

6.2.2.3.2 The signature of the method 'deregisterFactoryClass' shall be:

```
static void deregisterFactoryClass(
            const shared_ptr<MALAccessControlFactory>& factoryClass)
```

6.2.2.3.3 The parameters of the method 'deregisterFactoryClass' shall be assigned as described in table 6-2.

Table 6-2: MALAccessControlFactory 'deregisterFactoryClass' Parameter

Parameter	Description
factoryClass	Class extending MALAccessControlFactory

6.2.2.3.4 The method 'deregisterFactoryClass' shall remove the class from the factory class repository.

6.2.2.3.5 The method 'deregisterFactoryClass' shall do nothing if the class is not found in the factory class repository.

6.2.2.4 MALAccessControlFactory Creation

6.2.2.4.1 A static method 'newFactory' shall be defined in order to return a MALAccessControlFactory instance.

6.2.2.4.2 The signature of the method 'newFactory' shall be:

```
template<typename FactoryType>
static shared_ptr<MALAccessControlFactory> newFactory()
```

6.2.2.4.3 The method ‘newFactory’ shall resolve the specific MALAccessControlFactory class name using the C++ to ‘typeid(FactoryType).name()’, where FactoryType is the Class specified in the template of the method ‘newFactory’.

6.2.2.4.4 The method ‘newFactory’ shall lookup the MALAccessControlFactory implementation class from the factory class repository.

6.2.2.4.5 If the MALAccessControlFactory class is not found in the factory class repository, then it shall be loaded from the current class loader.

6.2.2.4.6 If the MALAccessControlFactory class is not found in the current class loader, then a MALException shall be raised with the ClassNotFoundException as the cause.

6.2.2.4.7 The method ‘newFactory’ shall not return the value NULL.

6.2.2.4.8 If no MALAccessControlFactory can be returned, then a MALException shall be raised.

6.2.2.5 MALAccessControl Instantiation

6.2.2.5.1 The factory class shall provide an abstract public method ‘createAccessControl’ to instantiate a MALAccessControl.

6.2.2.5.2 The signature of the method ‘createAccessControl’ shall be:

```
shared_ptr<MALAccessControl> createAccessControl(
                              const MALQoSProperties& properties) = 0;
```

6.2.2.5.3 The parameter of the method ‘createAccessControl’ shall be assigned as described in table 6-3.

Table 6-3: MALAccessControlFactory ‘createAccessControl’ Parameter

Parameter	Description
properties	Configuration properties

6.2.2.5.4 The parameter ‘properties’ may be NULL.

6.2.2.5.5 The method ‘createAccessControl’ shall not return the value NULL.

6.2.2.5.6 If no MALAccessControl can be returned, then a MALException shall be raised.

6.2.2.5.7 The method ‘createAccessControl’ shall be implemented by every specific factory class.

6.2.3 MALACCESSCONTROL

6.2.3.1 Definition

A MALAccessControl interface shall be defined in order to enable the MAL layer to check if a message is allowed to be transmitted or received.

6.2.3.2 Check

6.2.3.2.1 A method ‘check’ shall be defined in order to intercept and check:

a) outgoing messages before they are transmitted to the transport layer;

b) incoming messages before they are delivered to the MAL client.

6.2.3.2.2 The signature of the method ‘check’ shall be:

```
shared_ptr<MALMessage> check(const shared_ptr<MALMessage>& msg) = 0;
```

6.2.3.2.3 The parameter of the method ‘check’ shall be assigned as described in table 6-4.

Table 6-4: MALAccessControl ‘check’ Parameter

Parameter	Description
msg	MALMessage to be checked

6.2.3.2.4 The method ‘check’ shall be invoked by the MAL layer:

a) before a MALMessage is sent through a MALEndpoint;

b) after a MALMessage has been received by a MALMessageListener.

6.2.3.2.5 The method ‘check’ shall return the MALMessage to be handled by the MAL layer.

6.2.3.2.6 A MALCheckErrorException shall be raised if the message is not allowed to be sent or delivered.

6.2.3.2.7 If a MALCheckErrorException is raised, the standard error shall be transmitted to the MALMessage sender.

6.2.4 MALCHECKERROREXCEPTION

6.2.4.1 Definition

6.2.4.1.1 A MALCheckErrorException class shall be defined in order to raise a CHECK ERROR as an exception.

6.2.4.1.2 The MALCheckErrorException class shall extend the MALInteractionException class.

6.2.4.2 Constructor

6.2.4.2.1 A MALCheckErrorException constructor shall be defined.

6.2.4.2.2 The MALCheckErrorException constructor signature shall be:

```
MALCheckErrorException(
                  const shared_ptr<MALStandardError>& standardError,
                  const MALQoSProperties& qosProperties)
```

6.2.4.2.3 The MALCheckErrorException constructor parameters shall be assigned as described in table 6-5.

Table 6-5: MALCheckErrorException Constructor Parameters

Parameter	Description
standardError	Error preventing the message to be transmitted
qosProperties	QoS properties of the MALMessage which cannot be transmitted

6.2.4.2.4 The MALCheckErrorException constructor shall call the MALInteractionException constructor and pass the MALStandardError parameter.

6.2.4.3 Get the QoS Properties

6.2.4.3.1 A getter method ‘getQosProperties’ shall be defined in order to return the QoS properties of the MALMessage which cannot be transmitted.

6.2.4.3.2 The signature of the method ‘getQosProperties’ shall be:

```
MALQoSProperties getQosProperties()
```

7 ENCODING API

7.1 OVERVIEW

The encoding API should be used by the transport modules that need to externalize the encoding behaviour of a body element. The use of this encoding API is optional. The encoding API enables the transport layer to share and reuse any encoding module that complies with this API. However, a transport layer can implement the encoding behaviour in an internal way without this API.

NOTE – Two types of encoding module can be implemented:

a) A generic encoding module implements the MALEncoder and MALDecoder interfaces in order to encode and decode body elements in a generic way, i.e., by calling the generic methods ‘encode’ and ‘decode’ implemented by each specific structures.

b) A specific encoding module does not implement the MALEncoder and MALDecoder interfaces. It encodes and decodes body elements in a specific way, e.g., by calling the getter and setter methods provided by the specific structures.

7.2 CLASSES AND INTERFACES

7.2.1 GENERAL

The classes and interfaces of the encoding API are contained in the namespace:

```
mo::mal::encoding
```

7.2.2 MALELEMENTSTREAMFACTORY

7.2.2.1 Definition

7.2.2.1.1 A MALElementStreamFactory class shall be defined in order to create and configure:

a) MALElementInputStream instances;

b) MALElementOutputStream instances.

7.2.2.1.2 This MALElementStreamFactory class shall be abstract.

7.2.2.1.3 The MALElementStreamFactory class shall be extended by every specific factory class.

7.2.2.2 Factory Class Registration

7.2.2.2.1 The MALElementStreamFactory class shall provide a static method ‘registerFactoryClass’ in order to register the class of a specific MALElementStreamFactory.

NOTE – The method ‘registerFactoryClass’ may be useful in environments where several class loaders are involved.

7.2.2.2.2 The signature of the method ‘registerFactoryClass’ shall be:

```
static void registerFactoryClass(
            const string& protocol,
            const shared_ptr<MALElementStreamFactory>& factoryClass)
```

7.2.2.2.3 The parameters of the method ‘registerFactoryClass’ shall be assigned as described in table 7-1.

Table 7-1: MALElementStreamFactory ‘registerFactoryClass’ Parameter

Parameter	Description
factoryClass	Class extending MALElementStreamFactory

7.2.2.2.4 The method ‘registerFactoryClass’ shall store the class in the factory class repository.

7.2.2.3 Factory Class Deregistration

7.2.2.3.1 The MALElementStreamFactory class shall provide a static method ‘deregisterFactoryClass’ in order to deregister the class of a specific MALElementStreamFactory.

7.2.2.3.2 The signature of the method ‘deregisterFactoryClass’ shall be:

```
static void deregisterFactoryClass(
                  const shared_ptr<MALElementStreamFactory>& factoryClass)
```

7.2.2.3.3 The parameters of the method ‘deregisterFactoryClass’ shall be assigned as described in table 7-2.

Table 7-2: MALElementStreamFactory 'deregisterFactoryClass' Parameter

Parameter	Description
factoryClass	Class extending MALElementStreamFactory

7.2.2.3.4 The method 'deregisterFactoryClass' shall remove the class from the factory class repository.

7.2.2.3.5 The method 'deregisterFactoryClass' shall do nothing if the class is not found in the factory class repository.

7.2.2.4 MALElementStreamFactory Creation

7.2.2.4.1 A static method 'newFactory' shall be defined in order to create a factory instance from a protocol name.

7.2.2.4.2 The signature of the method 'newFactory' shall be:

```
template<typename FactoryType>
static shared_ptr<MALElementStreamFactory> newFactory(
            const string& protocol,
            const MALQoSProperties& properties)
```

7.2.2.4.3 The parameters of the method 'newFactory' shall be assigned as described in table 7-3.

Table 7-3: MALElementStreamFactory Creation Parameters

Parameter	Description
protocol	Name of the protocol to be handled by the instantiated MALElementStreamFactory
properties	Configuration properties

7.2.2.4.4 The parameter 'protocol' shall give the name of the protocol handled by the transport.

7.2.2.4.5 The parameter 'properties' shall give the properties of the MALTransport.

7.2.2.4.6 The method 'newFactory' shall resolve the specific MALElementStreamFactory class name using the C++ to 'typeid (FactoryType).name ()', where FactoryType is the Class specified in the template of the method 'newFactory'. The encoding to use with a specific transport protocol class can be defined in a properties file. The examples are entries in a properties file and define the factory to use for creating the encoding class associated with a particular protocol:

NOTE – Each of those protocol properties is assigned with the class name of the MALElementStreamFactory. It should be noted that two different protocol properties can share the same MALElementStreamFactory class name. Below is an example of a configuration of protocol properties (the C++ namespace is omitted in the class names):

```
# AMS transport with BINARY encoding
mo.mal.encoding.protocol.amsbin  BinaryElementStreamFactory
# DDS transport with BINARY encoding
mo.mal.encoding.protocol.ddsbin  BinaryElementStreamFactory
# Web Service transport with SOAP encoding
mo.mal.encoding.protocol.wssoap   SoapElementStreamFactory
```

7.2.2.4.7 The method ‘newFactory’ shall lookup the MALElementStreamFactory implementation class from the factory class repository.

7.2.2.4.8 If the MALElementStreamFactory class is not found in the factory class repository, then it shall be loaded from the current class loader.

7.2.2.4.9 If the MALElementStreamFactory class is not found in the current class loader, then a MALException shall be raised.

7.2.2.4.10 The method ‘newFactory’ shall create an instance of MALElementStreamFactory by calling the empty constructor provided by the implementation class.

7.2.2.4.11 The method ‘newFactory’ shall not return the value NULL.

7.2.2.4.12 If no MALElementStreamFactory can be returned, then a MALException shall be raised.

7.2.2.4.13 Once the instance is created, the method ‘init’ provided by MALElementStreamFactory shall be called.

7.2.2.5 MALElementStreamFactory Initialization

7.2.2.5.1 A method ‘init’ shall be defined in order to enable the specific implementation class to initialize the encoding module.

7.2.2.5.2 The signature of the method ‘init’ shall be:

```
void init(const string& protocol, const MALQoSProperties& properties)
```

7.2.2.5.3 The parameters of the method ‘init’ shall be assigned as described in table 7-4.

Table 7-4: MALElementStreamFactory 'init' Parameters

Parameter	Description
protocol	Name of the protocol passed through the instantiation method
properties	Properties passed through the instantiation method

7.2.2.5.4 The parameter 'properties' may be empty.

7.2.2.5.5 If an internal error occurs, then a MALException shall be raised.

7.2.2.6 MALElementInputStream Instantiation

7.2.2.6.1 Two methods 'createInputStream' shall be defined in order to enable the transport layer to create a MALElementInputStream:

a) the first one shall declare an input stream parameter;

b) the second one shall declare a byte array parameter.

7.2.2.6.2 The signature of the method 'createInputStream' shall be:

```
shared_ptr<MALElementInputStream> createInputStream(
                              const std::istream& is) = 0;

shared_ptr<MALElementInputStream> createInputStream(
            unsigned char *bytes, int offset) = 0;
```

7.2.2.6.3 The parameter of the method 'createInputStream' shall be assigned as described in table 7-5.

Table 7-5: MALElementStreamFactory 'createInputStream' Parameter

Parameter	Description
is	Input stream used to decode elements
bytes	Bytes to be decoded
offset	Index of the first byte to decode

7.2.2.6.4 If a MALElementInputStream cannot be created then a MALException shall be raised.

7.2.2.7 MALElementOutputStream Instantiation

7.2.2.7.1 A method ‘createOutputStream’ shall be defined in order to enable the transport layer to create a MALElementOutputStream.

7.2.2.7.2 The signature of the method ‘createOutputStream’ shall be:

```
shared_ptr<MALElementOutputStream> createOutputStream(
                                     const std::ostream& os)
```

7.2.2.7.3 The parameter of the method ‘createOutputStream’ shall be assigned as described in table 7-6.

Table 7-6: MALElementStreamFactory ‘createOutputStream’ Parameter

Parameter	Description
os	Output stream used to encode elements

7.2.2.7.4 If a MALElementOutputStream cannot be created then a MALException shall be raised.

7.2.2.8 Encode Elements

7.2.2.8.1 A method ‘encode’ shall be defined in order to encode an element array and return the encoding result as a byte array.

7.2.2.8.2 The signature of the method ‘encode’ shall be:

```
shared_ptr<Blob> encode(
                        const vector<Element>& elements,
                        const shared_ptr<MALEncodingContext>& ctx)
```

7.2.2.8.3 The parameters of the method ‘encode’ shall be assigned as described in table 7-7.

Table 7-7: MALElementStreamFactory ‘encode’ Parameters

Parameter	Description
elements	Elements to encode
ctx	MALEncodingContext to be used in order to encode the elements

7.2.2.8.4 The allowed types for each encoded element shall be:

a) MAL element types;

b) MALEncodedElement;

c) List<MALEncodedElement>;

d) C++ types defined by a specific C++ mapping extension.

7.2.2.8.5 If an encoding error occurs then a MALException shall be raised.

7.2.3 MALELEMENTINPUTSTREAM

7.2.3.1 Definition

A MALElementInputStream interface shall be defined in order to decode Elements pursuant to the protocol handled by the MALElementStreamFactory.

7.2.3.2 Read an Element

7.2.3.2.1 A method ‘readElement’ shall be defined in order to decode an Element.

7.2.3.2.2 The signature of the method ‘readElement’ shall be:

```
shared_ptr<Element> readElement(
                        const shared_ptr<Element>& element,
                        const shared_ptr<MALEncodingContext>& ctx)
```

7.2.3.2.3 The parameters of the method ‘readElement’ shall be assigned as described in table 7-8.

Table 7-8: MALElementInputStream ‘readElement’ Parameters

Parameter	Description
element	Element to decode
ctx	MALEncodingContext to be used in order to decode an Element

7.2.3.2.4 The parameter ‘element’ may be NULL.

7.2.3.2.5 The allowed types for the parameter ‘element’ shall be:

a) MAL element types;

b) C++ types defined by a specific C++ mapping extension.

7.2.3.2.6 The parameter ‘ctx’ may be NULL.

7.2.3.2.7 If the MALElementInputStream is closed, then a MALException shall be raised.

7.2.3.2.8 The returned element may be not the same instance as the parameter ‘element’.

7.2.3.2.9 The allowed returned element types shall be:

a) MAL element types;

b) MALEncodedElement;

c) List<MALEncodedElement>;

d) C++ types defined by a specific C++ mapping extension.

7.2.3.3 Close

7.2.3.3.1 A method ‘close’ shall be defined in order to close the stream.

7.2.3.3.2 The signature of the method ‘close’ is:

```
public void close()
```

7.2.3.3.3 The method ‘close’ shall call the method ‘close’ provided by the InputStream (C++ istream) owned by this MALElementInputStream.

7.2.3.3.4 If an internal error occurs, then a MALException shall be raised.

7.2.4 MALELEMENTOUTPUTSTREAM

7.2.4.1 Definition

A MALElementOutputStream interface shall be defined in order to encode Elements pursuant to the protocol handled by the MALElementStreamFactory.

7.2.4.2 Write an Element

7.2.4.2.1 A method ‘writeElement’ shall be defined in order to encode an Element.

7.2.4.2.2 The signature of the method ‘writeElement’ shall be:

```
void writeElement(
            const shared_ptr<Element>& element,
            const shared_ptr<MALEncodingContext>& ctx)
```

7.2.4.2.3 The parameters of the method ‘writeElement’ shall be assigned as described in table 7-9.

Table 7-9: MALElementOutputStream 'writeElement' Parameters

Parameter	Description
element	Element to encode
ctx	MALEncodingContext to be used in order to encode the Element

7.2.4.2.4 The parameter 'element' may be NULL.

7.2.4.2.5 The allowed types for the parameter 'element' shall be:

a) MAL element types;

b) MALEncodedElement;

c) List<MALEncodedElement>;

d) C++ types defined by a specific C++ mapping extension.

7.2.4.2.6 The parameter 'ctx' may be NULL.

7.2.4.2.7 If the MALElementOutputStream is closed, then a MALException shall be raised.

7.2.4.3 Flush the Stream

7.2.4.3.1 A method 'flush' shall be defined in order to flush the stream.

7.2.4.3.2 The signature of the method 'flush' shall be:

```
void flush()
```

7.2.4.3.3 The method 'flush' shall write all the buffered data owned by this MALElementOutputStream.

7.2.4.3.4 The method 'flush' shall call the method 'flush' provided by the OutputStream (C++ ostream) owned by this MALElementOutputStream.

7.2.4.3.5 If an internal error occurs, then a MALException shall be raised.

7.2.4.3.6 If the MALElementOutputStream is closed, then a MALException shall be raised.

7.2.4.4 Close

7.2.4.4.1 A method 'close' shall be defined in order to close the stream.

7.2.4.4.2 The signature of the method 'close' shall be:

```
void close()
```

7.2.4.4.3 The implementation shall call the method ‘close’ provided by the OutputStream (C++ ostream) owned by this MALElementOutputStream.

7.2.4.4.4 If an internal error occurs, then a MALException shall be raised.

7.2.5 MALENCODINGCONTEXT

7.2.5.1 Definition

A class MALEncodingContext shall be defined in order to give access to:

a) the header of the MALMessage that contains the Element to encode or decode;

b) the description of the operation that has been called;

c) the index of the body element to encode or decode;

d) the list of the QoS properties owned by the MALEndpoint that sends or receives the Element;

e) the list of the QoS properties owned by the MALMessage that contains the Element to encode or decode.

7.2.5.2 Getters and Setters

Getter and setter methods shall be defined in order to give access to the attributes listed in table 7-10.

Table 7-10: MALEncodingContext Attributes

Attributes	**Type**
header	MALMessageHeader
operation	MALOperation
bodyElementIndex	int
endpointQosProperties	MALQoSProperties
messageQosProperties	MALQoSProperties

ANNEX A

DEFINITION OF ACRONYMS

(INFORMATIVE)

API	application programming interface
BLOB	binary large object
IP	interaction pattern
MAL	Message Abstraction Layer
MO	Mission Operations
QoS	quality of service
SANA	Space Assigned Numbers Authority
SM&C	Spacecraft Monitor & Control
URI	uniform resource identifier
URL	Uniform Resource Locator

ANNEX B

INFORMATIVE REFERENCES

(INFORMATIVE)

[B1] *Mission Operations Services Concept*. Issue 3. Report Concerning Space Data System Standards (Green Book), CCSDS 520.0-G-3. Washington, D.C.: CCSDS, December 2010.

[B2] *Mission Operations Reference Model*. Issue 1. Recommendation for Space Data System Practices (Magenta Book), CCSDS 520.1-M-1. Washington, D.C.: CCSDS, July 2010.

[B3] *Mission Operations Message Abstraction Layer—JAVA API*. Issue 1. Recommendation for Space Data System Practices (Magenta Book), CCSDS 523.1-M-1. Washington, D.C.: CCSDS, April 2013.

[B4] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. Reston, Virginia: ISOC, August 1998.

[B5] R. Hinden, B. Carpenter, and L. Masinter. *Format for Literal IPv6 Addresses in URL's*. RFC 2732. Reston, Virginia: ISOC, December 1999.

NOTE – Normative references are listed in 1.8.

ANNEX C

SECURITY, SANA, AND PATENT CONSIDERATIONS

(INFORMATIVE)

C1 SECURITY CONSIDERATIONS

C1.1 OVERVIEW

This annex subsection discusses various aspects of security with respect to the MAL C++ API.

C1.2 SECURITY BACKGROUND

The MAL supports a generic security and authorization concept that allows the appropriate mechanism to be used. This concept is similar to that of the MAL's hiding the transport protocol used. The MAL also does not impose any specific set of access restrictions regarding what operations and services may be accessed by whom but provides a framework of messages and patterns that can be restricted using an appropriate access control policy for a particular domain, implementation, or agency.

The MAL C++ API is responsible for providing interfaces and methods that allow the implementation of the MAL generic security and authorization concept. The following APIs are provided:

a) an authentication API for MAL consumers and providers;

b) an access control API to perform authorization checks when messages are sent or received through the MAL layer;

c) a transport API to send and receive authenticated and non encrypted messages.

C1.3 SECURITY CONCERNS

C1.3.1 Consumer and Provider Authentication

The authentication of the consumers and providers is done above the MAL layer through an authentication service that provides an operation to get an authentication identifier. The meaning of that authentication identifier is dependent on the security system used for the deployment. This identifier allows the MAL Access Control implementation to perform a lookup for authorization purposes.

The MAL C++ API specifies how to generate an authentication API by mapping the authentication service to C++. An authentication identifier is obtained by invoking an authentication service provider through this generated API.

C1.3.2 Authorization Checks

Authorization is done by the MAL Access Control that performs any required authorization checks and converts the authentication identifier into technology dependent security credentials.

The MAL C++ API defines an access control API providing a method to check and modify the messages that are transmitted through the MAL layer.

C1.3.3 Message Authentication and Confidentiality

The message authentication and confidentiality are provided by the MAL transport layer and are transparent to the MAL layer and above. As a consequence once a message rises above the MAL transport layer, the message has been authenticated and all encryption has been removed.

The MAL C++ API defines a transport API that provides methods to send and receive non encrypted messages assigned with technology dependent security credentials. The message authentication and encryption are managed by the module implementing the transport API.

Moreover the MAL C++ API enables the transport layer to obtain the reference of the Access Control if needed for messages authentication and encryption.

C1.3.4 Message Integrity

The message integrity is ensured by the MAL transport layer.

The MAL C++ API does not provide any interface to handle this. The integrity is internally verified by the transport layer.

C1.4 POTENTIAL THREATS AND ATTACK SCENARIOS

Two types of threats are identified:

a) the threats affecting the transport protocol and the security algorithms ensuring authentication, confidentiality and integrity;

b) the threats affecting the C++ language.

C1.5 CONSEQUENCES OF NOT APPLYING SECURITY

Four security aspects may not be applied:

a) authentication;

b) authorization;

c) confidentiality;

d) integrity.

If authentication is not applied then anyone can perform any operation. The system can only log operations performed but not by whom.

If authorization is not applied then clients must log in but once in they can perform any supported operation. The system can log who performed what.

So authentication and authorization should be applied in order that everyone must log in with different levels of access. The system can then restrict who performs what.

If confidentiality is not applied then anyone can read the messages exchanged between a consumer and a provider.

If integrity is not applied then the messages exchanged between a consumer and a provider can be altered.

C2 SANA CONSIDERATIONS

The recommendations of this document request SANA to create the registry defined as follows:

a) the registry named C++BindingNamespace consists of a table of parameters:

    1) C++ Namespace Name: a string of text naming the C++ namespace;

    2) Reference: a string of text referencing the CCSDS document that has created the C++ namespace;

b) the initial registry should be filled with the values in table C-1;

c) the registration rule for new values of this registry may require an engineering review, and the request must come from the official representative of a space agency, member of the CCSDS.

Table C-1: C++BindingNamespace Initial Values

C++ Namespace Name	Reference
mo::mal	CCSDS 523.2-R-1

C3 PATENT CONSIDERATIONS

No patents are known to apply to this Recommended Standard.

ANNEX D

CODE EXAMPLE

(INFORMATIVE)

D1 OVERVIEW

This annex contains a code example for launching a consumer and a provider of the service ExampleArea::ExampleService.

D2 HANDLER IMPLEMENTATION CODE

The following class is a code template for the handler implementation using the inheritance model:

```
class ExampleServiceHandlerImpl:
            public ExampleServiceInheritanceSkeleton
{
      public:
            void exampleSubmitOperation(
                  const ExampleParameter& parameter,
                  const std::shared_ptr<MALInteraction>& interaction)
            {
                  // …
            }
}
```

D3 PROVIDER LAUNCHING CODE

The following is the main method that launches an instance of ExampleService provider:

```
int main()
{
    MALProperties properties;

    shared_ptr<MALContextFactory> malContextFactory =
            MALContextFactory::newFactory<MALContextFactoryImpl>();

    MALContext malContext =
      malContextFactory->createMALContext(properties);

    ExampleAreaHelper::init(
      MALContextFactory::getElementFactoryRegistry());

    ExampleServiceHelper::init(
      MALContextFactory::getElementFactoryRegistry());

    shared_ptr<MALProviderManager> providerMgr =
                        malContext->createProviderManager();

    shared_ptr<ExampleServiceHandlerImpl> handler =
                               make_shared<ExampleServiceHandlerImpl>();

    URI brokerURI = ""; // the broker is private

    vector<QoSLevel> qosLevels { QoSLevel.ASSURED };

    MALQoSProperties qosProperties;

    bool isPublisher = true;

    shared_ptr<MALProvider> provider = providerMgr->createProvider(
                "<<Provider name>>",
                "<<Protocol name>>",
                ExampleServiceHelper::EXAMPLESERVICE_SERVICE,
                make_shared<Blob>(),
                handler,
                qosLevels,
                1,
                qosProperties,
                isPublisher,
                brokerURI);

    cout << "ExampleService URI: " << provider->getURI() << endl;
    cout << "ExampleService broker URI: "
                  << provider->getBrokerURI() << endl;
}
```

D4 CONSUMER LAUNCHING CODE

The following is the main method that launches an instance of ExampleService consumer:

```
int main(
{
    MALProperties properties;
    MALQoSProperties qosProperties;
    shared_ptr<Blob> authenticationID = make_shared<Blob>();
    IdentifierList domain;
    Identifier networkZone = "networkZone";
    Identifier sessionName = "LIVE";
    Unteger priority = 0;

    shared_ptr<MALContextFactory malContextFactory =
                  MALContextFactory::newFactory<MALContextFactoryImpl>();

    MALContext malContext =
            malContextFactory->createMALContext(properties);

    ExampleAreaHelper::init(
      MALContextFactory::getElementFactoryRegistry());

    ExampleServiceHelper::init(
      MALContextFactory::getElementFactoryRegistry());

    shared_ptr<MALConsumerManager> consumerMgr =
            malContext->createConsumerManager();

    shared_ptr<MALConsumer> consumer =
                  consumerMgr->createConsumer(
              "<<consumer name>>",
              URI("<<provider URI>>"),
              URI("<<broker URI>>"),
              ExampleServiceHelper::EXAMPLESERVICE_SERVICE,
              authenticationID,
              domain(),
              networkZone,
              SessionType.LIVE,
              sessionName,
              QoSLevel.ASSURED,
              qosProperties,
              priority);

    ExampleServiceStub stub = ExampleServiceStub(consumer);

}
```

D5 SHARED BROKER LAUNCHING CODE

If a shared broker is needed, then the following main method should also be executed:

```
int main()
{
    MALProperties properties;
    MALQoSProperties qosProperties;
    shared_ptr<Blob> authenticationID = make_shared<Blob>();

    shared_ptr<MALContextFactory malContextFactory =
                  MALContextFactory::newFactory<MALContextFactoryImpl>();

    shared_ptr<MALContext malContext =
                  malContextFactory->createMALContext(properties());

    ExampleAreaHelper.init(
                  MALContextFactory::getElementFactoryRegistry());

    ExampleServiceHelper.init(
                  MALContextFactory::getElementFactoryRegistry());

    shared_ptr<MALBrokerManager> brokerManager =
                  malContext->createBrokerManager();

    shared_ptr<MALBroker> sharedBroker = brokerManager.createBroker(
                  ExampleServiceHelper::EXAMPLESERVICE_SERVICE);

    MALBrokerBinding sharedBrokerBinding =
                        brokerManager->createBrokerBinding(
                              sharedBroker,
                              "<<Broker name>>",
                              "<<Protocol name>>",
                              authenticationID,
                              QoSLevel.ASSURED,
                              1,
                              qosProperties);

    cout << "Shared broker URI: " << sharedBrokerBinding.getURI() << endl;

}
```