FIWARE Advanced Middleware KIARA Documentation

Release 0.4.0

eProsima, DFKI, ZHAW

March 29, 2016

KIARA Advanced Middleware is a Java based communication middleware for modern, efficient and secure applications.

It is an implementation of the FIWARE Advanced Middleware Generic Enabler.

This first release focuses on the basic features of RPC communication:

- Modern Interface Definition Language (IDL) with a syntax based on the Corba IDL.
- Easy to use and extensible Application Programmer Interface (API).
- IDL derived operation mode providing Stubs and Skeletons for RPC Client/Server implementations.
- Synchronous and Asynchronous function calls.

Later versions will include additional features like:

- Application derived and Mapped operation mode providing dynamic declaration of functions and data type mapping.
- Advanced security features like field encryption and authentication.
- Additional communication patterns like publish/subscribe.

KIARA Advanced Middleware is essentially a library which is incorporated into the developed applications, the requirements are rather minimal. In particular it requires no service running in the background.

- *Manuals*
- *Architecture*
- *Open Specification*
- *API Specification*

Please also check out our Github repository

CHAPTER 1

KIARA Installation and Administration Guide

Date: 18th January 2016

- Version: *0.4.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

1.1 Introduction

KIARA Advanced Middleware is a Java based communication middleware for modern, efficient and secure applications. It is an implementation of the FIWARE Advanced Middleware Generic Enabler.

This first release focuses on the basic features of RPC communication:

- Modern Interface Definition Language (IDL) with a syntax based on the Corba IDL.
- Easy to use and extensible Application Programmer Interface (API).
- IDL derived operation mode providing Stubs and Skeletons for RPC Client/Server implementations.
- Synchronous and Asynchronous function calls.

Later versions will include additional features like:

- Application derived and Mapped operation mode providing dynamic declaration of functions and data type mapping.
- Advanced security features like field encryption and authentication.
- Additional communication patterns like publish/subscribe.

KIARA Advanced Middleware is essentially a library which is incorporated into the developed applications, the requirements are rather minimal. In particular it requires no service running in the background.

1.2 System Requirements

This section describes the basic requirements of KIARA Advanced Middleware and how to install them.

1.2.1 Hardware Requirements

The hardware requirements depend on the application to be developed. Any hardware running Java JRE/JDK 7 or later is supported.

This Oracle JDK Webpage provides specific minimal hardware requirements (Disk, Memory, CPU).

1.2.2 Software Requirements

Runtime Systems:

- Any Java SE JRE 7 or later distribution (OpenJDK, Oracle Java SE or IBM Java SDK)

Development Systems:

- Any Java SE JDK 7 or later distribution (OpenJDK, Oracle Java SE or IBM Java SDK)
- Build Tools (gradle or maven) and/or IDE (Eclipse, IntelliJ IDEA, Netbeans, ...)

1.2.3 Operating System support:

Any Operating system running Java JRE/JDK 7 or later is supported.

KIARA is tested on Windows (7), Linux (Ubuntu 14.04/Fedora 19) and OS X (10.9/10.10).

1.3 Installation

1.3.1 Installation of the Java JDK

Please follow the installation instructions of the respective Java distribution:

- OpenJDK (Default on most Linux-Systems)
- Oracle Java SE JDK (Linux, Windows, OS X, Solaris)
- IBM Java SE JDK (Linux, Windows, AIX, z/OS)

To verify that the installation is correct please open a terminal/shell/command line interface (cmd.exe, sh/bash/zsh) and check, that the java command is executable:

```
$ java -version
java version 1.7.0_65
OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-0ubuntu0.14.04.1)
OpenJDK 64-Bit Server VM (build 24.65-b04, mixed mode)
```

If the java command is not found, please make sure, that the `<java_home>/bin` directory is in your `PATH` environment variable and the `JAVA_HOME` environment variable is set (see troubleshooting instructions on the Oracle Website).

1.3.2 Installation of Build Tools

On development systems developers should use a build tool to compile, test, package and deploy applications. In the Java (JVM) world the most commonly used tools are Gradle and Apache Maven. An alternative approach is to use the built in build-management of IDEs like Eclipse, IntelliJ IDEA or Netbeans.

In the following section covers the Installation of these tools.

Gradle

Gradle is the newest and most flexible build tool for Java. It provides every good and detailed documentation and tutorials to get developers started. Official installation instructions are here.

The official way to install Gradle is to download and unpack the binary ZIP file to a common directory, set the environment variable `GRADLE_HOME` and add the bin directory to your `PATH` environment variable.

Windows installation

Download newest gradle-x.x.x-all.zip from http://www.gradle.org/downloads.

Open CLI (cmd.exe)

```
unzip ~\Download\gradle-x.x.x-all.zip -d C:\Program Files
setx GRADLE_HOME C:\Program Files\gradle-x.x.x /M
setx PATH %PATH%;%GRADLE_HOME%\bin /M
```

/M sets the value on a machine level, which means for all users. The values are stored permanently and will be available in any new cmd.exe session.

Unix (Linux / OS X / Solaris / FreeBSD) manual installation

Download newest `gradle-x.x.x-all.zip` from http://www.gradle.org/downloads.

Open a shell:

```
$ sudo unzip ~/gradle-x.x.x-all.zip -d /usr/share/
$ sudo ln -s /usr/share/gradle-x.x.x /usr/share/gradle
```

Open `~/.profile` (single user) or `/etc/profile` (all users) and add the following lines:

```
export GRADLE_HOME=/usr/share/gradle
export PATH=$PATH:$GRADLE_HOME/bin
```

Unix (Linux / OS X / Solaris / FreeBSD) installation using gvm

An alternative and simpler option to install gradle for a single user is to use the Groovy enVironment Manager (gvm) to install and update Gradle. You need the commands/packages curl and unzip to be installed on your system.

Open shell:

```
$ curl -s get.gvmtool.net | bash
... follow instructions
$ gvm install gradle
```

See gvm help to get more infos about other options of gvm, like updating or switching between different versions.

Verify installation

Open a new shell or cmd.exe session and test if gradle is available:

```
$ gradle -v
------------------------------------------------------------
Gradle 2.2.1
------------------------------------------------------------

Build time:   2014-11-24 09:45:35 UTC
```

```
Build number: none
Revision:     6fcb59c06f43a4e6b1bcb401f7686a8601a1fb4a

Groovy:       2.3.6
Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM:          1.7.0_65 (Oracle Corporation 24.65-b04)
OS:           Linux 3.13.0-34-generic amd64
```

Apache Maven

Apache Maven is a very common build tool in the Java/JVM world and is very well known for its dependency management and its central artifact repository (mavencentral). Find the documentation and tutorials on the main page. Installation instructions and downloads are here.

The official way to install Maven is to download and unpack the binary ZIP file to a common directory, set the environment variable `M2_HOME` and add the bin directory to your `PATH` environment variable.

Windows installation

Follow process in the install instructions.

Unix (Linux / OS X / Solaris / FreeBSD) manual installation

Download newest apache-maven-x.x.x-bin.zip from http://maven.apache.org/download.html.

Open shell:

```
$ sudo unzip ~/apache-maven-x.x.x-bin.zip -d /usr/share/
$ sudo ln -s apache-maven-x.x.x /usr/share/maven
```

Open `~/.profile` (single user) or `/etc/profile` (all users) and add the following lines:

```
export M2_HOME=/usr/share/maven
export PATH=$PATH:$M2_HOME/bin
```

Unix (Linux / OS X / Solaris / FreeBSD) installation using package manager

An alternative option to install maven is to use the package manager of the unix system.

- on DEB based systems (Debian,Ubuntu,...) `$ sudo apt-get install maven` (this is a quite outdated version 3.0.x)
- on RPM based systems (RedHat,CentOS,Fedora,...) exists no official package (use above manual instructions).
- on OS X you can install Maven using a packet manager for OS X like Homebrew or MacPorts. Because the packages are usually compiled during installation you need to install Xcode beforehand. This is recommended especially, if you already have Xcode installed or you would like to install also other common unix packages.

Homebrew (http://brew.sh):

:: `$ brew install maven`

MacPorts (http://www.macports.org/install.php):

:: `$ port install maven2`

Verify installation Open a new shell or cmd.exe session and test if maven is available:

```
$ mvn -version
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T22:58:10+02:00)
Maven home: /usr/local/Cellar/maven/3.2.3/libexec
Java version: 1.8.0_20, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: mac os x, version: 10.10.2, arch: x86_64, family: mac
```

Integraded Development Environments (IDE)

To install your IDE please check the webpage of your prefered IDE product:

- Eclipse
- IntelliJ IDEA
- Netbeans

These IDEs typically integrate well with Gradle and Apache Maven using plugins. Alternatively you have to copy the KIARA libraries manually to the library folder of your project and add them to your classpath.

Installation of the kiaragen tool

The kiaragen tool is part of the KIARA components available on Maven Central. Depending on your build tool kiaragen can be easily integrated or it can be called with a shell/batch script.

If you are using Maven or an IDE you can download an executable jar file of kiaragen from the ga|1|g:org.fiware.kiara|KIARA Maven-Central repository, or you can find it in a standalone distribution available online.

On Windows:

- There are two ways of obtaining the kiaragen software:
- Download the file `kiaragen-x.x.x-cli.jar` from Maven Central and place it into a directory (e.g. subdirectory kiaragen).
- To make the execution simpler you can also download the kiaragen.bat script from the kiaragen project (https://github.com/FIWARE-Middleware/kiaragen) and copy it into the scripts directory (create if not not created yet).

  Please take into account that the script will look for the `kiaragen-x.x.x-cli.jar` file inside the kiaragen subdirectory.
- Now the tool can be called using: kiaragen.bat when the scripts folder in the installation dir is in the execute path or with a relative path `./scripts/kiaragen.bat` for project local installations.

On Linux / OS X:

- Download the file `kiaragen-x.x.x-cli.jar` from Maven Central
- Place it in a directory of your shells execute path (e.g. `/usr/local/bin`). Alternatively you can also add it to your project dir and call it with a relative path (./scripts/kiaragen.sh).
- To make the execution simpler you can also download the kiaragen.sh script from the kiaragen project (https://github.com/FIWARE-Middleware/kiaragen) and copy it into the scripts directory.

  Please take into account that the script will look for the `kiaragen-x.x.x-cli.jar` file inside the kiaragen subdirectory.

- Now the tool can be called using: kiaragen.sh when the scripts folder in the installation dir is in the execute path or with a relative path `./scripts/kiaragen.sh` for project local installations executable flag is lost while downloading, you can set it again using `chmod a+x kiaragen`

1.3.3 KIARA components

The KIARA components (libraries) are usually delivered together with the the developed application and do not have to be installed separately.

1.3.4 Setting up the development environment

In this section it is explained how to set up your development environment and configure your project to use KIARA Advanced Middleware. We support the most common build tools for Java projects, which are:

- Gradle
- Apache Maven

All Java Integrated Development environments like Eclipse, InteliJ IDEA, Netbeans, etc. provide support for one of these tools.

Please check the Installation Manual for instructions how to install the required plugins and import your KIARA project.

Gradle

Set up the basic project structure

If you do not yet have a project you can setup the basic structure using the gradle init plugin:

```
$ mkdir calculator
$ cd calculator
$ gradle init --type java-library
```

This will create a basic directory structure for your source and test code and create a commented `build.gradle` file for a Java application.

Additionally the gradle wrapper is set up, which allows developers to execute gradle tasks without installing the gradle tool globally.

Configure your Gradle project to use KIARA

To use KIARA in your project you have to extend your build.gradle file:

```
apply plugin: 'java'

sourceCompatibility = 1.7
version = '1.0'

// In this section you declare where to find the dependencies of your project
repositories {
  mavenCentral()
}

// In this section declare the dependencies for your production and test code
dependencies {
    compile group: 'org.fiware.kiara', name: 'KIARA', version: '0.4.0'
    compile group: 'org.slf4j', name: 'slf4j-api', version: '1.7.7'
```

```
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

The KIARA artefacts are available on the Maven Central repository. So you have to make sure, 'mavenCentral() is part of your repositories section.

To include the KIARA artefacts you have to add the `kiara` main library to the dependencies section. All the depending libraries will be added automatically to your project.

The following is a typical file structure for a gradle project using KIARA:

```
.
├── build                                       // generated files
|   ├── classes                                 // compiled classes
|   |   └── main
|   |       └── com
|   |           └── example
|   |               ├── Calculator.class
|   |               ├── CalculatorAsync.class
|   |               ├── CalculatorClient.class
|   |               ├── CalculatorProcess.class
|   |               ├── CalculatorProxy.class
|   |               ├── CalculatorServant.class
|   |               ├── CalculatorServantExample.class
|   |               ├── ClientExample.class
|   |               ├── IDLText.class
|   |               └── ServerExample.class
|   ├── generated-src                           // generated support classes
|   |   └── kiara
|   |       └── com
|   |           └── example
|   |               ├── Calculator.java
|   |               ├── CalculatorAsync.java
|   |               ├── CalculatorClient.java
|   |               ├── CalculatorProcess.java
|   |               ├── CalculatorProxy.java
|   |               └── CalculatorServant.java
|   └── libs
|       └── Calculator-1.0.jar                  // packaged application
├── build.gradle                                // gradle build file
├── gradle
|   └── wrapper                                 // gradle wrapper files
|       └── ...
├── gradlew                                     // gradle wrapper unix script
├── gradlew.bat                                 // gradle wrapper windows script
├── settings.gradle
└── src                                         // source files
    ├── main
    |   ├── idl                                 // IDL definitions for KIARA
    |   |   └── com
    |   |       └── example
    |   |           └── Calculator.idl
    |   └── java                                // application code
    |       └── com
    |           └── example
    |               ├── ClientExample.java              // client start code
    |               ├── ServerExample.java              // server start code
    |               └── CalculatorServantExample.java   // servant impl.
    └── test
        └── java
```

Some basic gradle tasks:

`./gradlew build` → builds all classes and run tests
`./gradlew jar` → creates the application jar
`./gradlew clean` → cleans up your project
`./gradlew tasks` → shows all available tasks

Apache Maven

Set up the basic project structure

If you do not yet have a maven project you can setup the basic structure using the archetype plugin:

```
$ mvn archetype:generate \
 -DgroupId=mw.kiara \
 -DartifactId=calculator \
 -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

This will create a basic directory structure for your source and test code and create a commented `pom.xml` file for a Java application.

1.4 Sanity Check Procedures

1.4.1 End to End testing

To verify your development environment you can download and run the KIARA Calculator example application.

Download the example application from Github. You can clone it using git or download the ZIP archive and unzip it to an empty directory.

```
$ git clone https://github.com/FIWARE-Middleware/Examples.git KiaraCalculator
$ cd KiaraCalculator
```

Build the application

```
$ gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL

Total time: 1.793 secs
```

Run the Server

```
$ gradle runServer
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:runServer
CalculatorServerExample
Apr 15, 2015 6:00:32 PM io.netty.util.internal.logging.Slf4JLogger info
```

```
INFO: [id: 0xbfb04d67] REGISTERED
Apr 15, 2015 6:00:32 PM io.netty.util.internal.logging.Slf4JLogger info
INFO: [id: 0xbfb04d67] BIND(/0.0.0.0:9090)
Apr 15, 2015 6:00:32 PM io.netty.util.internal.logging.Slf4JLogger info
INFO: [id: 0xbfb04d67, /0:0:0:0:0:0:0:0:9090] ACTIVE
> Building 75% > :runServer
```

Open an new terminal window and run the Client

```
$ cd KiaraCalculator
$ gradle runClient
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:runClient
CalculatorClientExample

10 + 5 = 15

Apr 15, 2015 5:54:06 PM org.fiware.kiara.Kiara shutdown
INFO: shutdown 2 services
Apr 15, 2015 5:54:06 PM org.fiware.kiara.Kiara shutdown
INFO: shutdown org.fiware.kiara.netty.NettyTransportFactory$1@880ec60
Apr 15, 2015 5:54:11 PM org.fiware.kiara.Kiara shutdown
INFO: shutdown org.fiware.kiara.transport.impl.Global$1@3f3afe78

BUILD SUCCESSFUL

Total time: 12.76 secs
```

The Client task should terminate with BUILD SUCCESSFUL and the Calculation should show the correct result.

1.4.2 List of Running Processes

KIARA Advanced Middleware itself do not install any kind of daemon or service. There are no running processes, but libraries to link to your applications.

1.4.3 Network interfaces Up & Open

The KIARA Middleware itself does not open or provide services, therefore has no open Ports or Interfaces. Applications using KIARA can open any ports or interfaces and firewalls have to be configured accordingly.

The provided TestServer is opening and listening by default on Port 9090.

1.4.4 Databases

N/A

1.5 Diagnosis Procedures

1.5.1 Resource availability

This middleware requires very few resources, any typical PC should be enough to run the regular examples.

1.5.2 Remote Service Access

N/A

1.5.3 Resource consumption

Depends on your application, it can be as low of 256 Kbytes of heap space and almost zero cpu use. The amount of RAM depends on your data types size and the different persistence options, please read the user manual for more information.

1.5.4 I/O flows

N/A

CHAPTER 2

KIARA User and Programmer Guide

Date: 18th January 2016

- Version: *0.4.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

2.1 Introduction

KIARA Advanced Middleware is a Java based communication middleware for modern, efficient and secure applications. It is an implementation of the FIWARE Advanced Middleware Generic Enabler.

This first release focuses on the basic features of RPC communication:

- Modern Interface Definition Language (IDL) with a syntax based on the Corba IDL.
- Easy to use and extensible Application Programmer Interface (API).
- IDL derived operation mode providing Stubs and Skeletons for RPC Client/Server implementations.
- Synchronous and Asynchronous function calls.

Later versions will include additional features like:

- Application derived and Mapped operation mode providing dynamic declaration of functions and data type mapping.
- Advanced security features like field encryption and authentication.
- Additional communication patterns like publish/subscribe.

KIARA Advanced Middleware is essentially a library which is incorporated into the developed applications, the requirements are rather minimal. In particular it requires no service running in the background.

2.1.1 Background and Detail

This User and Programmers Guide relates to the Advanced Middleware GE which is part of the *Interface to Networks and Devices (I2ND) chapter*. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

2.2 User guide

These products are for programmers, who will invoke the APIs programmatically and there is no user interface as such.

See the programmers guide section to browse the available documentation.

2.3 Programmers guide

2.3.1 Middleware Operation Modes

The KIARA Advanced Middleware supports multiple operation modes. From traditional IDL-based approaches like Corba, DDS, Thrift up to newer approaches which start with the application data structure and automatically create the wire format.

We therefore differentiate three operation modes.

IDL derived operation mode

The IDL derived operation mode is similar to the traditional middleware approaches.

Based on the IDL definition we generate with a precompiler stub- and skeleton-classes, which have to be used by the application to implement the server and client (or Pub/Sub) application parts.

Prerequisite: IDL definition

Generated: Stubs and Skeletons (at compile time) which have to be used by the application

Examples: Corba, DDS, Thrift, ...

Application derived operation mode

This mode is typical for some modern (e.g. RMI, WebService,...) frameworks. Based on an application specific interface definitions, the framework automatically generates Server- and Client-Proxy-Classes, which serialize the application internal data structures and send them over the wire. Using Annotations, the required serialization and transport mechanisms and type mappings can be influenced.

This mode implicitly generates an IDL definition based on the Java interfaces definition and provide this IDL through a "service registry" for remote partners.

Prerequisite: Application-Interface-Definition (has to be the same on client and server side)

Generated: Server-/Client-Proxies (generated at runtime)

Examples: RMI, JAX-RS, Spring REST, ...

Mapped operation mode

Goal of the mapped operation mode is to separate the application interfaces from the data structure used to transport the data over the wire. Therefore the middleware has to map the application internal data structure and interfaces to a common IDL definition. Advantage is, that the application interface on client and server (or publisher/subscriber) side can be different.

Prerequisite: Application-Interface-Definition (can be different on server and client side) IDL Definition

Generated: Server-/Client-Proxis (generated at runtime, which map the attributes & operations

Examples: KIARA

The current release of KIARA provides support for the traditional IDL derived operation mode, being able to handle different communication patterns such as RPC or Publish/Subscribe. Application derived and mapped operation mode will follow in a future release.

2.3.2 A quick example

In the following chapters we will use the following example application to explain the basic concepts of building an application using KIARA.

Calculator

The KIARA Calculator example application provides an API to ask for simple mathematics operations over two numbers. Is a common used example when trying to understand how an RPC framework works.

Basically the service provides two functions:

- **`float add (float n1, float n2)`:** Returns the result of adding the two numbers introduced as parameters (n1 and n2).
- **`float subtract (float n1, float n2)`:** Returns the result of subtracting the two numbers introduced as parameters (n1 and n2).

The KIARA Calculator example is provided within this distribution, so itcan be used as starting point.

Basic procedure

Before diving into the details describing the features and configure your projectfor KIARA, the following quick example should show the basic steps to create a simple client and server application in the different operation modes.

Detailed instructions on how to execute the particular steps are given in chapter *Building a KIARA RPC application*.

IDL derived application process

In the IDL derived approach, first the IDL definition has to be created:

```
service Calculator
{
    float32 add (float32 n1, float32 n2);
    float32 subtract (float32 n1, float32 n2);
};
```

The developer has to implement the functions inside the class `CalculatorServantImpl`:

```
public static class CalculatorServantImpl extends CalculatorServant
{
    @Override
    public float add (/*in*/ float n1, /*in*/ float n2) {
        return (float) n1 + n2;
    }

    @Override
    public float subtract (/*in*/ float n1, /*in*/ float n2) {
        return (float) n1 - n2;
    }
    ...
}
```

Now the server can be started:

```
Context context = Kiara.createContext();
Server server = context.createServer();
Service service = context.createService();

// Create and register an instance of the CalculatorServant implementation.
CalculatorServant Calculator_impl = new CalculatorServantImpl();
service.register(Calculator_impl);

// register the service on port 9090 using CDR serialization
server.addService(service, "tcp://0.0.0.0:9090", "cdr");

// run the server
server.run();
```

The client can connect and call the remote functions via the proxy class:

```
Context context = Kiara.createContext();

// setup the connection to the server
Connection connection = context.connect("tcp://192.168.1.18:9090?serialization=cdr");

// get the client Proxy implementation
CalculatorClient client = connection.getServiceProxy(CalculatorClient.class);

// call the remote methods
float result = client.add(3, 5);
```

Application derived application example

This example will be added, when the feature is implemented.

Mapping application example

This example will be added, when the feature is implemented.

2.3.3 Kiaragen tool

Kiaragen installation

To install kiaragen, please follow the installation instructions that can be found in the .

Generate support code manually using kiaragen

To call `kiaragen` manually it has to be installed and in your run path.

The usage syntax is:

```
$ kiaragen [options] <IDL file> [<IDL file> ...]
```

Options:

-help	Shows help information
-version	Shows the current version of KIARA/kiaragen
-package	Defines the package prefix of the generated Java classes. Default: no package
-d <path>	Specify the output diretory for the generated Java classes. Default: Current working dir

-replace	Replaces existing generated
--example <pattern>	Generates the support files (interfaces, classes, stubs, skeletons,...) for the given target communication pattern. These classes can be used by the developer to implement his application. It also creates build.gradle files. Supported values: • rpc: Creates an example application which uses RPC as a communication framework. • ps: Creates an example application which uses Publish/Subscribe as a communication pattern.
-ppDisable	Disables the preprocessor.
--ppPath <path>	Specifies the path of the preprocessor. Default: Systems C++ preprocessor
-t <path>	Specify the output temporal directory for the files generated by the preprocessor. Default: machine temp path

2.3.4 KIARA IDL

The KIARA Interface Definition Language (IDL) can be used to describe data types, namespaces, constants and even remote functions the server will offer (when using RPC pattern). In addition the KIARA IDL supports the declaration and application of Annotations to add metadata to almost any IDL element. These can be used by the code generator, when implementing the service functionality or configure some specific runtime functionality. The IDL syntax is based on the OMG IDL 3.5.

The basic structure of an IDL File is shown in the picture in the right.

Following, a short overview of the supported KIARA IDL elements. For a detailed description please see KIARA IDL Specification chapter KIARA Interface Definition Language.

- **Import Declarations:** Definitions can be split into multiple files and/or share common elements among multiple definitions using the import statement.
- **Namespace Declarations:** Within a definition file the declarations can be grouped into modules. Modules are used to define scopes for IDL identifiers. KIARA supports the modern keyword namespace. Namespaces can be nested to support multi-level namespaces.
- **Constant Declarations:** A constant declarations allows the definition of literals, which can be used as values in other definitions (e.g. as return values, default parameters, etc.)
- **Type Declarations**
  - **Basic Types:** KIARA IDL supports the OMG IDL basic data types like float, double, (unsigned) short/int/long, char, wchar, boolean, octet, etc. Additionally it supports modern aliases like float32, float64, i16, ui16, i32, ui32, i64, ui64 and byte
  - **Constructed Types:** Constructed Types are combinations of other types like. The following constructs are supported:
    - **Structures:** Struct types are mapped as classes in Java code. These structures can contain every other data type that can be described using KIARA IDL.
    - **Unions:** Union types are mapped into Java by using special classes. These classes use a discriminator value to distinguish between the different types that form the union.
    - **Exceptions:** Exception types are mapped as classes in Java code. These exceptions can contain every other data type that can be described using KIARA IDL.
  - **Template Types:** Template types are frequently used data structures like the various forms of collections. The following Template Types are supported:
    - **Lists** Ordered collection of elements of the same type. “list” is the modern variant of the OMG IDL keyword “sequence”

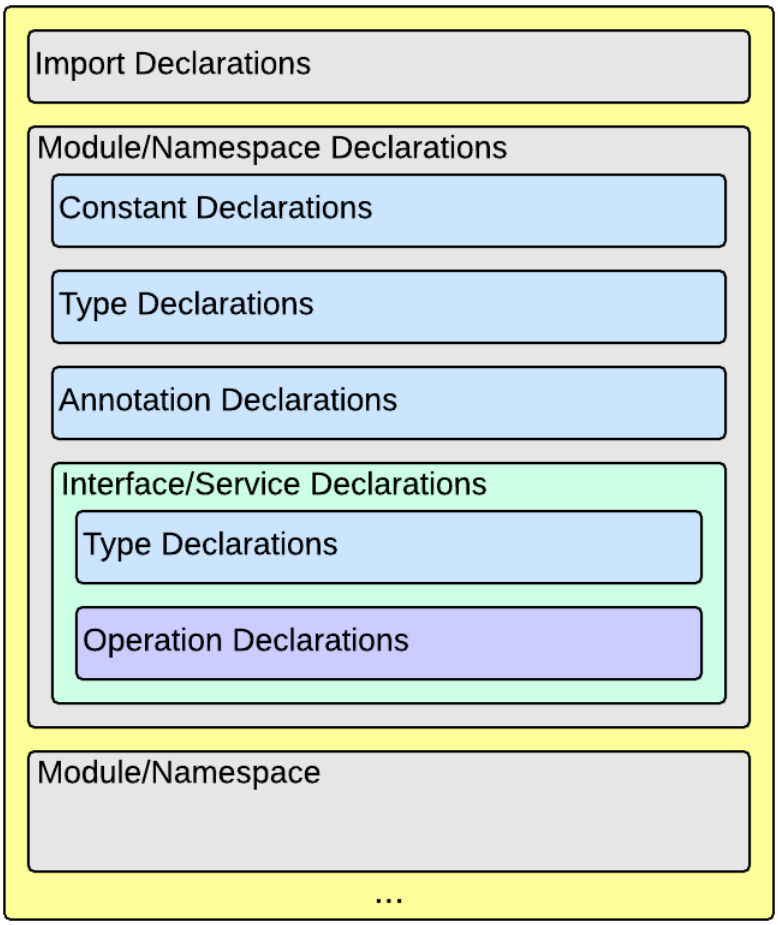

Fig. 2.1: IDL File Structure

    - **Sets** Ordered collection of different elements of the same type. "list" is the modern variant of the OMG IDL keyword "sequence"
    - **Maps** Lists of paired objects indexed by a key
    - **Strings** Collection of chars, will be mapped to the String representation of the language.
    - **Complex Declarations:** In addition to the above Type declarations, KIARA supports multidimensional Arrays using the bracket notation (e.g. `int monthlyRevenue[12][10]`)
- **Service Declarations:** KIARA supports interface and service declarations via IDL. Meaning that the user can declare different services where the operations are going to be placed.
- **Operation Declarations:** Operations can be declared within the services following the standard OMG IDL notation.

2.3.5 Using KIARA to create an RPC application

KIARA Advanced Middleware allows the developer to easily implement a distributed application using remote procedure invocations. In client/server paradigm, a server offers a set of remote procedures that the client can remotely call. How the client calls these procedures should be transparent.

For the developer, a proxy object represents the remote server, and this object offers the remote procedures implemented by the server. In the same way, how the server obtains a request from the network and how it sends the reply should also be transparent. The developer just writes the behaviour of the remote procedures.

KIARA Advanced Middleware offers this transparency and facilitates the development.

IDL derived operation mode in RPC

The general steps to build an application in IDL derived operation mode are:

1. Define a set of remote procedures: using the KIARA Interface Definition Language.
2. Generation of specific remote procedure call support code: a Client-Proxy and a Server-Skeleton.
3. Implement the servant: with the needed behaviour.
4. Implement the server: filling the server skeleton with the behaviour of the procedures.
5. Implement the client: using the client proxy to invoke the remote procedures.

This section describes the basic concepts of these four steps that a developer has to follow to implement a distributed application.

Defining a set of remote procedures using the KIARA IDL

The KIARA Interface Definition Language (IDL) can be used to define the remote procedures (operations) the server will offer. Simple and Complex Data Types | used as parameter types in these remote procedures are also defined in the IDL file. The IDL file for our example application (`calculator.idl`) shows the usage of some of the above elements.

```
service Calculator
{
    float32 add (float32 n1, float32 n2);
    float32 substract (float32 n1, float32 n2);
};
```

Generating remote procedure call support code

KIARA Advanced Middleware includes a Java application named `kiaragen`. This application parses the IDL file and generates Java code for the defined set of remote procedures.

All support classes will be generated (e.g. for structs):

- `x.y.<StructName>`: Support classes containing the definition of the data types as well as the serialization code.

Using the `-example` option (described below), kiaragen will generate the following files for each of your module/service definitions:

- **`x.y.<IDL-ServiceName>`:** Interface exposing the defined synchronous service operation calls.
- **`x.y.<IDL-ServiceName>Async`:** Interface exposing the asynchronous operation calls.
- **`x.y.<IDL-ServiceName>Client`:** Interface exposing all client side calls (sync & async).
- **`x.y.<IDL-ServiceName>Process`:** Class containing the methods that will be executed to process dynamic calls.
- **`x.y.<IDL-ServiceName>Proxy`:** This class encapsulates all the logic needed to call the remote operations. (Client side proxy → stub).
- **`x.y.<IDL-ServiceName>Servant`:** This abstract class provides all the mechanisms (transport, un/marshalling, etc.) the server requires to call the server functions.
- **`x.y.<IDL-ServiceName>ServantExample`:** This class will be extended to implement the server side functions (see *Servant Implementation*).
- **`x.y.ClientExample`:** This class contains the code needed to run a possible example of the client side application.
- **`x.y.ServerExample`:** This class contains the code needed to run a possible example of the server side application.
- **`x.y.IDLText`:** This class contains a String whose value is the content of the IDL file.

The package name `x.y.` can be declared when generating the support code using `kiaragen` (see `-package` option in `kiaragen` tool *description*).

For our example the call could be:

```
$ kiaragen -example rpc -package com.example src/main/idl/calculator.idl
Loading templates...
org.fiware.kiara.generator.kiaragen
org.fiware.kiara.generator.idl.grammar.Context
Processing the file calculator.idl...
Creating destination source directory... OK
Generating Type support classes...
Generating application main entry files for interface Calculator... OK
Generating specific server side files for interface Calculator... OK
Generating specific client side files for interface Calculator... OK
Generating common server side files... OK
Generating common client side files... OK
```

This would generate the following files:

```
.
└── src                                                  // source files
    ├── main
    |   ├── idl                                          // IDL definitions for kiaragen
    |   |   └── calculator.idl
    |   └── java                                         // Generated support files
    |       └── com.example
    |            |                                       // Generated using --example
    |            ├── Calculator.java                     // Interface of service
```

```
    |             ├-- CalculatorAsync.java              // Interface of async calls
    |             ├-- CalculatorProcess.java            // Process methods for dynamic operations
    |             ├-- CalculatorClient.java             // Interface client side
    |             ├-- CalculatorProxy.java              // Client side implementation
    |             ├-- CalculatorServant.java            // Abstract server side skeleton
    |             ├-- CalculatorServantExample.java     // Dummmy servant impl.
    |             ├-- ClientExample.java                // Example client code
    |             ├-- ServerExample.java                // Example server code
    |             └-- IDLText.java                      // IDL File contents
    └-- build.gradle                                    // File with targets to compile the example
```

Servant implementation

Please note that the code inside the file `x.y.<IDL-ServiceName>ServantExample.java` (which in this case is `CalculatorServantExample.java`) has to be modified in order to specify the behaviour of each declared function.

```
class CalculatorServantExample extends CalculatorServant {

  public float add (/*in*/ float n1, /*in*/ float n2) {
        return (float) n2 + n2;
    }

    public float substract (/*in*/ float n1, /*in*/ float n2) {
        return (float) n1 - n2;
    }

}
```

Implementing the server

The source code generated using kiaragen tool (by using the `-example rpc` option) contains a simple implementation of a server. This implementation can obviously be extended as far as the user wants, this is just a very simple server capable of executing remote procedures.

The class containing the mentioned code is named ServerExample, and its code is shown below:

```
public class ServerExample {

    public static void main (String [] args) throws Exception {

        System.out.println("CalculatorServerExample");

        Context context = Kiara.createContext();
        Server server = context.createServer();

        CalculatorServant Calculator_impl = new CalculatorServantExample();

        Service service = context.createService();

        service.register(Calculator_impl);

        //Add service waiting on TCP using CDR serialization
        server.addService(service, "tcp://0.0.0.0:9090", "cdr");

        server.run();

    }

}
```

Creating a secure TCP server (SSL)

In order to create a secure TCP server, the URL specified to listen into must be different. In this case, we would use `tcps` as a network protocol instead of `tpc`. The only change that has to be done in the code is to change the service address.

This is shown in the following snippet:

```
//Add service waiting on SSL TCP using CDR serialization
server.addService(service, "tcps://0.0.0.0:9090", "cdr");
```

Implementing the client

The source code generated using kiaragen tool (by using the `-example rpc` option) contains a simple implementation of a client. This implementation must be extended in order to show the output received from the server.

In the KIARA Calculator example, as we have defined first the add function in the IDL file, this will be the one used by default in the generated code. The code for doing this is shown in the following snippet:

```
public class ClientExample {
    public static void main (String [] args) throws Exception {
        System.out.println("CalculatorClientExample");

    float n1 = (float) 3.0;
    float n2 = (float) 5.0;

        float ret = (float) 0.0;

        Context context = Kiara.createContext();

                    //Connect to server listening in 127.0.0.1:9090 (TCP)
        Connection connection =
                     context.connect("tcp://127.0.0.1:9090?serialization=cdr");
        Calculator client = connection.getServiceProxy(CalculatorClient.class);

    try {
            ret = client.add(n1, n2);
            System.out.println("Result: " + ret);
        } catch (Exception ex) {
            System.out.println("Exception: " + ex.getMessage());
            return;
        }
    }

    Kiara.shutdown();
}
```

The previous code has been shown exactly the way it is generated, with only two differences:

- Parameter initialization: Both of the parameters n1 and n2 have been initialized to random values (in this case 3 and 5).
- Result printing: To have feedback of the response sent by the server when the remote procedure is executed.

Creating a secure TCP client (SSL)

In order to create a secure TCP client, the URI to connect to must be that of the server (who must also be using SSL TCP for a full secure communication).

This is shown in the following snippet:

```
//Connect to server listening in 127.0.0.1:9090 (SSL)
Connection connection =
             context.connect("tcps://127.0.0.1:9090?serialization=cdr");
```

Compiling the client and the server

For the client and server examples to compile, some jar files are needed. These files are located under the lib directory provided with this distribution, and they must be placed in the root working directory, under the lib folder:

```
.
├-- src                       // source files
├-- lib                       // generated support files
└-- build.gradle              // Gradle compilation script
```

To compile the client using gradle, the call would be the next one (change target clientJar to serverJar to compile the server):

```
$ gradle clientJar
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:clientJar

BUILD SUCCESSFUL

Total time: 3.426 secs
```

After compiling both of them the following files will be generated:

```
.
├-- src                       // source files
├-- build                     // generated by gradle
|   ├-- classes               // Compiled .class files
|   ├-- dependency-cache      // Inner gradle files
|   ├-- libs                  // Executable jar files
|   └-- tmp                   // Temporal files used by gradle
├-- lib
└-- build.gradle              //  Gradle compilation script
```

In order to execute the examples, just cd where they are placed (build/libs directory), and execute them using the command `java -jar file_to_execute.jar`.

2.3.6 Using KIARA to create an RPC application (using the dynamic API)

The "KIARA RPC Dynamic API" allows the developers to easily execute calls in an RPC framework without having to statically generate code to support them. In the following sections, the different concepts of this feature will be explained.

Using the dynamic API we still need the IDL file, which declares the "contract" between server and client by defining the data types and services (operations) the server offers.

For the dynamic API the IDL format is identical to the one used for the static/compile time version. For example the IDL file for our demo application (`calculator.idl`) is identical to the static use-case:

```
service Calculator
{
    float32 add (float32 n1, float32 n2);
    float32 substract (float32 n1, float32 n2);
};
```

Declaring the remote calls and data types at runtime

In the dynamic approach, the comple time `kiaragen` code-generator will not be required anymore. Instead, the middleware provides a function to load the IDL definition from a String object. The generation of the IDL

String has to be done by the developer. For example it can be loaded from a File, from a URL or generated by an algorithm.

The process to declare the dynamic part is as follows:

- The server loads the IDL String (e.g. from a file).
- The IDL definition will then be provided to the clients connecting with the server.
- On the server the developer has to provide objects to act as servants and execute code depending on the function the client has requested.

Loading the IDL definition

On the server side, in order to provide the user with a definition of the functions that the server offers, the first thing to be done is to load the IDL definition into the application.

Therefore, the `Service` class provides a public function that can be used to load the IDL information from a String object. It is the developers responsibility to load the String from the source (e.g. from a file).

The following snippet shows an example on how to do this:

```
// Load IDL content string from file
String idlString = new String(Files.readAllBytes(Paths.get("calculator.idl")));
/* This is just one way to do it. Developer decides how to do it */

// Load service information dynamically from IDL
Service service = context.createService();
service.loadServiceIDLFromString(idlString);
```

Implementing the service functionality

Unlike in the static approach, in the dynamic version exists no Servant class to code the behaviour of the functions. To deal with this, KIARA provides a functional interface `DynamicFunctionHandler` that acts as a servant implementation. This class must be used to implement the function and register it with the service, which means to map the business logic of each function with its registered name.

```
// Create type descriptor and dynamic builder
final TypeDescriptorBuilder tdbuilder = Kiara.getTypeDescriptorBuilder();
final DynamicValueBuilder dvbuilder = Kiara.getDynamicValueBuilder();
// Create type descriptor int (used for the return value)
final PrimitiveTypeDescriptor intType =
                        tdbuilder.createPrimitiveType(TypeKind.INT_32_TYPE);

// Implement the functional interface for the add function
DynamicFunctionHandler addHandler = new DynamicFunctionHandler() {
     @Override
     public void process(
          DynamicFunctionRequest request,
          DynamicFunctionResponse response
     ) {
          // read the parameters
          int a = (Integer)((DynamicPrimitive)request.getParameterAt(0)).get();
          int b = (Integer)((DynamicPrimitive)request.getParameterAt(1)).get();
          // create the return value
          final DynamicPrimitive intValue =
                              (DynamicPrimitive)dvbuilder.createData(intType);
          intValue.set(a+b);    // implmement the function
          response.setReturnValue(intValue);
     }
}
```

```
// Register function and map handler (do this for every function)
service.register("Calculator.add", addHandler);
```

Implementing the server

Because the server functionality is not encapsuled in generated Servant classes, the server implmentation is a bit more extensive. It still follows the same pattern as in the static API, but the implementation and registration of the dynamic functions has to be done completely by the developer.

The following ServerExample class shows, how this would look like:

```
public class ServerExample {
    public static void main (String [] args) throws Exception {
        System.out.println("CalculatorServerExample");

        Context context = Kiara.createContext();
        Server server = context.createServer();

        // Enable negotiation with clients
        server.enableNegotiationService("0.0.0.0", 8080, "/service");

        Service service = context.createService();
        String idlContent =
        new String(Files.readAllBytes(Paths.get("calculator.idl")))
        service.loadServiceIDLFromString(idlContent);

        // Create descriptor and dynamic builder
        final TypeDescriptorBuilder tdbuilder = Kiara.getTypeDescriptorBuilder();
        final DynamicValueBuilder dvbuilder = Kiara.getDynamicValueBuilder();

        // Declare handlers
        DynamicFunctionHandler addHandler;
        DynamicFunctionHandler substractHandler;
        addHandler = /* Implement handler for the add function */;
        substractHandler = /* Implement handler for the substract function */;

        // Register services
        service.register("Calculator.add", addHandler);
        service.register("Calculator.substract", substractHandler);

        //Add service waiting on TCP with CDR serialization
        server.addService(service, "tcp://0.0.0.0:9090", "cdr");

        server.run();
    }
}
```

Creating a secure TCP server (SSL)

In order to create a secure TCP server, the URL specified to listen into must be different. In this case, we would use `tcps` as a network protocol instead of `tpc`. The only change that has to be done in the code is to change the service address.

This is shown in the following snippet:

```
            // Enable negotiation with clients
server.enableNegotiationService("0.0.0.0", 8080, "/service");

            ...

//Add service waiting on SSL TCP using CDR serialization
server.addService(service, "tcps://0.0.0.0:9090", "cdr");
```

Please note that the negotiation service has to be enabled first, otherwise the client will not be able to retrieve the connection information from the server.

Implementing the client

On the client side the key point is the negotiation with the server to download the IDL it provides. After downloading, it will automatically parse the content and generate the necessary information to create the dynamic objects.

When the `DynamicProxy` is created the functions provided by the server can be executed by using `DynamicFunctionRequest` objects. The parameters of the functions have to be set in the request using `DynamicData` objects. The call of the request function `execute()` will finally perform the call to the server and return the result in a `DynamicFunctionResponse` object.

The following code shows the client implementation:

```
public class ClientExample {
    public static void main (String [] args) throws Exception {
        System.out.println("CalculatorClientExample");

        Context context = Kiara.createContext();

        // Create connection indicating the negotiation service
        Connection connection =
                    context.connect("kiara://127.0.0.1:9090/service");

        // Create client by using the proxy's name
        DynamicProxy client = connection.getDynamicProxy("Calculator");

        // Create request object
        DynamicFunctionRequest request = client.createFunctionRequest("add");
        ((DynamicPrimitive) request.getParameterAt(0)).set(8);
        ((DynamicPrimitive) request.getParameterAt(1)).set(5);

        // Create response object and execute RPC
        DynamicFunctionResponse response = request.execute();
        if (response.isException()) {
            DynamicData result = response.getReturnValue();
            System.out.println("Exception = " + (DynamicException) result);
        } else {
            DynamicData result = response.getReturnValue();
            System.out.println("Result = " + (DynamicPrimitive) result);
        }
    // shutdown the client
        Kiara.shutdown();
    }
}
```

Creating a secure TCP client (SSL)

In order to create a secure TCP client, the URI to connect to must be that of the server's negotiation endpoint. When using the dynamic API, KIARA will automatically match the type of connection the server is using, whether it is TCP or TCPS (if the networking card of the computer supports it)

For this, the code for the client is exactly the same (note this in the following snippet):

```
// Create connection indicating the negotiation service
Connection connection =
             context.connect("kiara://127.0.0.1:9090/service");
```

2.3.7 Using KIARA to create a Pub/Sub application

KIARA Advanced Middleware allows the developer to easily implement a distributed application using a Publish/Subscribe pattern. In software architecture, publish/subscribe is a messaging pattern when messages of a specific data type (topic) are sent by entities called publishers, and received by entities who are subscribed to that same data type, called subscribers.

From the point of view of the developer, all he knows is that he has a certain data type in his application and he wants it to be sent. How the publisher publishes this data in the network and how the subscriber gets it must be transparent.

KIARA Advanced Middleware offers this transparency and facilitates the development.

IDL derived operation mode using Pub/Sub

The general steps to build an application in IDL derived operation mode are:

1. Define the application data types using KIARA IDL: using the KIARA Interface Definition Language.
2. Generation of specific support code: those classes representing the types defined using IDL.
3. Generate the Pub/Sub example: using the kiaragen tool.
4. Implementing the Publisher side: using the Publisher entity and the generated type support classes.
5. Implementing the Subscriber side: using the Subscriber entity and the generated type support classes.

This section describes the basic concepts of these steps that a developer has to follow to implement a distributed application.

Defining the application data types using KIARA IDL

The KIARA Interface Definition Language (IDL) can be used to define the application data types to be published. Simple and Complex Data Types inside the structures can also be defined in the IDL file, but take into account that only structures will count as Topic types.

The IDL file for our RPC example application shows the definition of a temperature sensor whose value is going to be published over the wire when changed.

```
struct TSensor
{
    float32 temperature;
};
```

Generate Pub/Sub code using kiaragen

KIARA Advanced Middleware includes a Java application named `kiaragen`. By using this application, the type support code for the structure defined in the IDL file can be generated. The files that will result as the output of the kiaragen execution are the following:

- x.y.: Support classes containing the definition of the data types as well as the serialization code.
- x.y.Type: Topic class for the data type. This class will be the one used to register the data types in a specific topic.

Using ps as -example option, kiaragen will generate the following files for the data type definitions:

- x.y.SubscriberExample: This class contains the code needed to run a simple application with a Subscriber.
- x.y.PublisherExample: This class contains the code needed to run a simple application with a Publisher.

The package name x.y. can be declared when generating the support code using kiaragen (see `-package` option below).

For our example the call could be:

```
$ kiaragen -example ps -package com.example src/main/idl/calculator.idl
Loading templates...
org.fiware.kiara.generator.kiaragen
org.fiware.kiara.generator.idl.grammar.Context
Processing the file calculator.idl...
Creating destination source directory... OK
Generating Type support classes...
Generating Type support class for structure TSensor... OK
Generating Topic class for structure TSensor... OK
Generating Publisher example main code for Topic TSensor... OK
Generating Subscriber example main code for Topic TSensor... OK

Generating GRADLE compilation script... OK
```

This would generate the following files:

```
.
└── src                                                 // source files
    ├── main
    |   ├── idl                                         // IDL definitions for kiaragen
    |   |   └── sensor.idl
    |   └── java                                        // Generated support files
    |       └── com.example
    |            |                                      // Generated using --example ps
    |            ├── TSensor.java                       // User data type
    |            ├── TSensorType.java                   // Topic class for user data type
    |            ├── TSensorPublisherExample.java       // Publisher example code
    |            └── TSensorSubscriberExample.java      // Subscriber example code
    └── build.gradle                                    // File with targets to compile the example
```

Static Endpoint Discovery (SED) using XML files

In this version of the Publish/Subscribe pattern implemented in KIARA, the discovery of endpoints is done statically by loding the information of those endpoints from an XML file. It supports loading such information from a String variable with the contents of the XML discovery file as well.

The discovery information than can be represented into the XML file includes the participant (with its name), and the endpoints this participant might have (readers or writers). it also supports adding multiple participant entities as well as multiple reader or writer configurations.

The XML tags supported by KIARA are described below, grouped into different categories according to the entity they belong to.

staticdiscovery

This tag is used to define that the XML file is going to contain information about the RTPS Endpoint Discovery protocol.

The available tags inside `staticdiscovery` are the following:

Tag	Type	Description
`<participant>`	complexType	Participant entity.

participant

The participant tag is the one used to define a grouping entity for readers and writers. It allows to add as many endpoints as the user wants, as well as to configure the participant name.

The available tags inside `participant` are the following:

Tag	Type	Description
`<name>`	element	Name of the Participant entity
`<writer>`	complexType	Writer entity
`<reader>`	complexType	Reader entity

writer

The writer tag is the use used to describe all the characteristics of the reader endpoint. There can be multiple writers, as long as their values do not interfere one another.

The available tags inside `writer` are the following:

Tag	Type	Description
<userId>	element	Integer defining the user ID for this endpoint.
<entityId>	element	Integer defining the specific ID of the endpoint.
<topicName>	element	Indicates the name of the Topic used by the endpoint.
<topicDataName>	element	Indicates the name of the data type that can be sent by the endpoint.
<topicKind>	element	Indicates whether the endpoint uses keyed topics or not. Supported values: • WITH_KEY • NO_KEY
<reliabilityQos>	element	Indicates which kind of reliability is used by the endpoint. Supported values: • RELIABLE_RELIABILITY_QOS • BEST_EFFORT_RELIABILITY_QOS
<unicastLocator>	complexType	List of unicastLocator types indicating the unicast IP adresses of this endpoint. Attributes: **address** IP address of the endpoint. **port** Integer indicating the port for communication.
<multicastLocator>	complexType	List of unicastLocator types indicating the multicast IP adresses of this endpoint. Attributes: **address** IP address of the endpoint. **port** Integer indicating the port for communication.
<topic>	complexType	Entity inticading the name, data type and kind of the topic this endpoint is related to. Attributes: **name** Name of the topic. **dataType** Name of the dataType related to this topic. **kind** Indicates whether it is a keyed topic or not. Supported values: • WITH_KEY • NO_KEY
<durabilityQos>	element	String element indicating the durability of the data send by the endpoint. Supported values: • TRANSIENT_LOCAL_DURABILITY_QOS • VOLATILE_DURABILITY_QOS

reader

The reader tag is the use used to describe all the characteristics of the reader endpoint. There can be multiple readers, as long as their values do not interfere one another.

The available tags inside `reader` are the following:

Tag	Type	Description
<userId>	element	Integer defining the user ID for this endpoint.
<entityId>	element	Integer defining the specific ID of the endpoint.
<topicName>	element	Indicates the name of the Topic used by the endpoint.
<topicDataName>	element	Indicates the name of the data type that can be received by the endpoint.
<expectsInlineQos>	element	Boolean value inticating whether the reader endpoint expects to receive inline QoS in the RTPS messages or not.
<topicKind>	element	Indicates whether the endpoint uses keyed topics or not. Supported values: • WITH_KEY • NO_KEY
<reliabilityQos>	element	Indicates which kind of reliability is used by the endpoint. Supported values: • RELIABLE_RELIABILITY_QOS • BEST_EFFORT_RELIABILITY_QOS
<unicastLocator>	complexType*	List of unicastLocator types indicating the unicast IP adresses of this endpoint. Attributes: **address** IP address of the endpoint. **port** Integer indicating the port for communication.
<multicastLocator>	complexType*	List of unicastLocator types indicating the multicast IP adresses of this endpoint. Attributes: **address** IP address of the endpoint. **port** Integer indicating the port for communication.
<topic>	complexType	Entity inticading the name, data type and kind of the topic this endpoint is related to. Attributes: **name** Name of the topic. **dataType** Name of the dataType related to this topic. **kind** Indicates whether it is a keyed topic or not. Supported values: • WITH_KEY • NO_KEY
<durabilityQos>	element	String element indicating the durability of the data send by the endpoint. Supported values:

Implementing the Publisher

The PubliserExample class is the one containing the main entry point for creating an application capable of publishing the user's data types over the wire. This class is automatically generated by using the `kiaragen` tool, and it contains a basic initialization of QoS (Qualities of Service), a participant, and one simple Publisher entity.

The following PublisherExample class shows how this would look like:

```
public class TSensorPublisherExample {

    private static final TSensorType type = new TSensorType();

    public static void main (String [] args) throws InterruptedException {
```

The generated class has a static final variable named type, and it will be used to register the user's data type.

The predefined arguments this example will handle are:

- domainId: This parameter is a number indicating the domain identifier for the RTPS communication. If not specified, the default value is 0.
- sampleCount: Number of samples the publisher will send. If not specified, the publisher will send examples without stopping.

```
int domainId = 0;
if (args.length >= 1) {
     domainId = Integer.parseInt(args[0]);
}

int sampleCount = 0;
if (args.length >= 2) {
     sampleCount = Integer.parseInt(args[1]);
}
```

In the following lines, the data itself is created by using the generated Topic class. The developer can now edit the created object before sending it over the network.

```
TSensor instance = type.createData();

// Initialize your data here
```

Now, the participant's attributes are initialized. Note that the domainId introduces as a parameter will be used here, and also that the attributes specify the participant to activate the static discovery protocol.

To use the static discovery, either an XML file or a String variable with the XML contents can be used. In the generated example, the chosen approach is to load the XML discovery information by using a single String variable. In this String, the known endpoints have to be defined. In this case, a participant containing a BEST_EFFORT reader.

```
ParticipantAttributes pAtt = new ParticipantAttributes();
pAtt.rtps.builtinAtt.domainID = domainId;
pAtt.rtps.builtinAtt.useStaticEDP = true;

final String edpXml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
        + "<staticdiscovery>"
        + "    <participant>"
        + "        <name>SubscriberParticipant</name>"
        + "        <reader>"
        + "            <userId>1</userId>"
        + "            <topic name=\"TSensorTopic\" dataType=\"TSensor\" kind=\"NO_KEY\"></topic>
        + "            <expectsInlineQos>false</expectsInlineQos>"
        + "            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>"
        + "        </reader>"
        + "    </participant>"
        + "</staticdiscovery>";
```

```
pAtt.rtps.builtinAtt.setStaticEndpointXML(edpXml);

pAtt.rtps.setName("PublisherParticipant");
```

At this point, the only thing remaining to be done before creating the Publisher is to finally create the Participant and register the user's data type. To do so, the generated Topic class must be used **after** the participant has been correctly initialized.

```
Participant participant = Domain.createParticipant(pAtt, null /* LISTENER */);
if (participant == null) {
     throw new RuntimeException("createParticipant");
}

Domain.registerType(participant, type);
```

The Publisher's attributes must specify the topic name and the name of the data type, and this information has to be the same in the other endpoints so that they can communicate with each other. In this generated example, the topic data name will be the same of the defined structure. Note that the example uses by default a BEST_EFFORT configuration for the Publisher.

```
// Create publisher
PublisherAttributes pubAtt = new PublisherAttributes();
pubAtt.setUserDefinedID((short) 1);
pubAtt.topic.topicDataTypeName = "TSensor";
pubAtt.topic.topicName = "TSensorTopic";
pubAtt.qos.reliability.kind = ReliabilityQosPolicyKind.BEST_EFFORT_RELIABILITY_QOS;

org.fiware.kiara.ps.publisher.Publisher<TSensor> publisher = Domain.createPublisher(participant,

if (publisher == null) {
    Domain.removeParticipant(participant);
    throw new RuntimeException("createPublisher");
}
```

Finally, the examples are sent according to the number of samples specified via parameter (without stopping if this number is not set).

```
int sendPeriod = 4000; // milliseconds
for (int count=0; (sampleCount == 0) || (count < sampleCount); ++count) {
     System.out.println("Writing TSensor, count: " + count);
     publisher.write(instance);
     Thread.sleep(sendPeriod);
}
```

In order for the Participant to stop succesfully, it must be removed from the Domain (all the associated endpoints will be stopped as well), and then the method named shutdown belonging to the Kiara class will be the one to stop all running services.

```
        Domain.removeParticipant(participant);

        Kiara.shutdown();

        System.out.println("Publisher finished");

    }

}
```

Implementing the Subscriber

The SubscriberExample class is the one containing the main entry point for creating an application capable of subscribing to a topic representing the user's data types. This class is automatically generated by using the `kiaragen`

tool, and it contains a basic initialization of QoS (Qualities of Service), a participant, and one simple Subscriber entity.

The following PublisherExample class shows how this would look like:

```
public class TSensorSubscriberExample {

    private static final TSensorType type = new TSensorType();

    public static void main (String [] args) throws InterruptedException {
```

as it happened with the PublisherExample, the generated class has a static final variable named type, and it will be used to register the user's data type.

The predefined arguments this example will handle are:

- domainId: This parameter is a number indicating the domain identifier for the RTPS communication. If not specified, the default value is 0.
- sampleCount: Number of samples the subscriber expects to receive. If not specified, the will run without stopping.

```
int domainId = 0;
if (args.length >= 1) {
     domainId = Integer.parseInt(args[0]);
}

int sampleCount = 0;
if (args.length >= 2) {
     sampleCount = Integer.parseInt(args[1]);
}
```

Now, the participant's attributes are initialized. Note that the domainId introduces as a parameter will be used here, and also that the attributes specify the participant to activate the static discovery protocol.

To use the static discovery, either an XML file or a String variable with the XML contents can be used. In the generated example, the chosen approach is to load the XML discovery information by using a single String variable. In this String, the known endpoints have to be defined. In this case, a participant containing a BEST_EFFORT writer.

```
ParticipantAttributes pAtt = new ParticipantAttributes();
pAtt.rtps.builtinAtt.domainID = domainId;
pAtt.rtps.builtinAtt.useStaticEDP = true;

final String edpXml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
        + "<staticdiscovery>"
        + "    <participant>"
        + "        <name>PublisherParticipant</name>"
        + "        <writer>"
        + "            <userId>1</userId>"
        + "            <topicName>TSensorTopic</topicName>"
        + "            <topicDataType>TSensor</topicDataType>"
        + "            <topicKind>NO_KEY</topicKind>"
        + "            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>"
        + "            <livelinessQos kind=\"AUTOMATIC_LIVELINESS_QOS\" leaseDuration_ms=\"100\">
        + "        </writer>"
        + "     </participant>"
        + "    </staticdiscovery>";

pAtt.rtps.builtinAtt.setStaticEndpointXML(edpXml);

pAtt.rtps.setName("SubscriberParticipant");
```

At this point, the only thing remaining to be done before creating the Subscriber is to finally create the Participant and register the user's data type. To do so, the generated Topic class must be used **after** the participant has been

correctly initialized.

```
Participant participant = Domain.createParticipant(pAtt, null /* LISTENER */);
if (participant == null) {
     throw new RuntimeException("createParticipant");
}

Domain.registerType(participant, type);
```

The Publisher's attributes must specify the topic name and the name of the data type, and this information has to be the same in the other endpoints so that they can communicate with each other. In this generated example, the topic data name will be the same of the defined structure. Note that the example uses by default a BEST_EFFORT configuration for the Subscriber.

```
// Create publisher
SubscriberAttributes satt = new SubscriberAttributes();
satt.setUserDefinedID((short) 1);
satt.topic.topicDataTypeName = "TSensor";
satt.topic.topicName = "TSensorTopic";
satt.qos.reliability.kind = ReliabilityQosPolicyKind.BEST_EFFORT_RELIABILITY_QOS;


// CountDown object to store the number of received samples
final CountDownLatch doneSignal = new CountDownLatch(sampleCount);
```

For this Subscriber, a SubscriberListener object is implemented below. It will print out when a new saple has been received by the Subscriber, and it will also take care of the total number of samples that have already been received.

```
org.fiware.kiara.ps.subscriber.Subscriber<TSensor> subscriber = Domain.createSubscriber(participar

    @Override
    public void onNewDataMessage(Subscriber<?> sub) {
        TSensor type = (TSensor) sub.takeNextData(null /* SampleInfo */);
        while (type != null) {
            System.out.println("Message received");
            type = (TSensor) sub.takeNextData(null);
            doneSignal.countDown();
        }
    }

    @Override
    public void onSubscriptionMatched(Subscriber<?> sub, MatchingInfo info) {
        // Write here you handling code
    }

});

if (subscriber == null) {
    Domain.removeParticipant(participant);
    throw new RuntimeException("createSubscriber");
}


int receivePeriod = 4000; // milliseconds
while ((sampleCount == 0) || (doneSignal.getCount() != 0)) {
    System.out.println("$ctx.currentSt.name$ Subscriber sleeping for " + receivePeriod/1000 + " se
    Thread.sleep(receivePeriod);
}
```

In order for the Participant to stop succesfully, it must be removed from the Domain (all the associated endpoints will be stopped as well), and then the method named shutdown belonging to the Kiara class will be the one to stop all running services.

```
        Domain.removeParticipant(participant);

        Kiara.shutdown();

        System.out.println("Publisher finished");

    }

}
```

2.3.8 Concerns

Connection compatibilities when using SSL over TCP

A secure connection is made by using TLS v1.2 (Transport Layer Security), an updated version of the SSL v3.1(Secure Sockets Layer). The use of this security layer carries a procedure to establish a connection between two endpoints, more than the classical Three-Way Handshake used in standard TCP. When using SSL, after the Three-Way Handshake, the client sends a `Client Hello` package that the server must answer with a `Server Hello`, and then the connection can be negotiated and established.

When the connection protocol specified is TCPS (SSL), there are some minor compatibility concerns that must be taken into account. The following table shows how the secure connections work depending on the side (client or server).

Server Protocol	TCP Client	TCPS Client
TCP	OK	ERROR*
TCPS	ERROR	OK

- The problem when only the client is TCPS is that the TCP connection is succesfully established, but the server will not answer the `Hello` package from the client, so this last one will never detect the connection as literally finished.

CHAPTER 3

Advanced Middleware Architecture

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

3.1 Copyright

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

3.2 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

3.3 Overview

This specification describes the Advanced Middleware GE, which enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FIWARE GEs.

Middleware in general provides a software layer between the application and the communication network and allows application to abstract from the intricacies of how to send a piece of data to a service offered by a another application and possibly return results. The middleware offers functionality to find and establish a connection to a service, negotiate the best wire and transport protocols, access the applications data structures and encode the necessary data in a format suitable for the chosen protocol, and finally send that data and possibly receive results in return. In a similar way an application can use the middleware also to offer services to other applications by registering suitable service functionality and interfaces, which can then be used as targets of communication.

3.4 Basic Concepts

In this section several basic concepts of the Advanced Communication Middleware are explained. We assume that the reader is familiar with the basic functionality of communication middleware like CORBA or WebServices.

Fig. 3.1: Publish/Subscribe Pattern

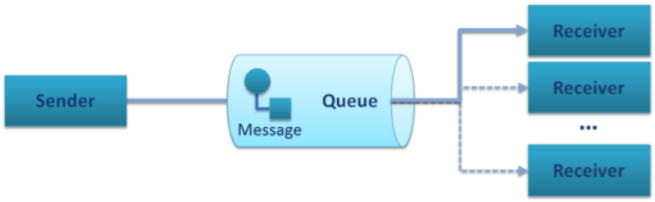

Fig. 3.2: Point-To-Point Pattern

3.4.1 Communication Patterns

We can distinguish between three main different messaging patterns, Publish/Subscribe, Point-to-Point, and Request/Reply, shown schematically below:

All available middleware technologies implement one or more of these messaging patterns and may incorporate more advanced patterns on top of them. Most RPC middleware is based on the Request/Reply pattern and more recently, is extended towards support of Publish/Subscribe and/or the Point-To-Point pattern.

W3C Web Service standards define a Request/Reply and a Publish/Subscribe pattern which is built on top on that (WS-Notification). CORBA, in a similar way, build its Publish/Subscribe pattern (Notification Service) on top of a Request/Reply infrastructure. In either case the adopted architecture is largely ruled by historical artifacts instead of performance or functional efficiency. The adopted approach is to emulate the Publish/Subscribe pattern on top of the more complex pattern thus inevitably leading to poor performance and complex implementations.

The approach of the Advanced Middleware takes the other direction. It provides native Publish/Subscribe and implements the Request/Reply pattern on top of this infrastructure. Excellent results can be achieved since the Publish/Subscribe is a meta-pattern, in other words a pattern generator for Point-To-Point and Request/Reply and potential alternatives.

3.4.2 Interface Definition Language (IDL)

The Advanced Middleware GE supports a novel IDL to describe the Data Types and Operations. Following is a list of the main features it supports:

- **IDL, Dynamic Types & Application Types**: It support the usual schema of IDL compilation to generate support code for the data types.
- **IDL Grammar**: An OMG-like grammar for the IDL as in DDS, Thrift, ZeroC ICE, CORBA, etc.

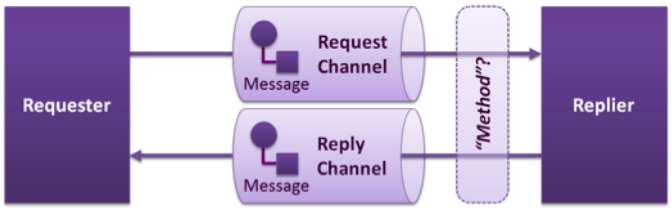

Fig. 3.3: Request/Reply Pattern

- **Types**: Support of simple set of basic types, structs, and various high level types such as lists, sets, and dictionaries (maps).
- **Type inheritance, Extensible Types, Versioning**: Advanced data types, extensions, and inheritance, and other advanced features will be supported.
- **Annotation Language**: The IDL is extended with an annotation language to add properties to the data types and operations. These will, for example, allows adding security policies and QoS requirements.
- **Security**: The IDL allows for annotating operations and data types though the annotation feature of our IDL, allowing setting up security even at the field level.

3.5 Generic Architecture

3.5.1 General Note

In contrast to other GEs, the Advanced Middleware GE is not a standalone service running in the network, but a set of compile-/runtime tools and a communication library to be integrated with the application.

The Advanced Middleware (AMi) architecture presented here offers a number of key advantages over other available middleware implementations:

- **High-Level Service Architecture:** AMi offers applications a high-level architecture that can shield them from the complexities and dangers of network programming. When applications declare services and data structures they can annotate them with the QoS, security, and other requirements while AMi automatically implement them. Thus application developers can exclusively focus on the application functionality.
- **Security:** The network is the main security threat to most applications today but existing middleware has offered only limited security functionality that has often been added as an afterthought and requires the application developer (who are often not security experts) to configure the security functionality. Instead, AMi offers *Security by Design* where security has been designed into the architecture from the start. Applications can simply declare their security needs in the form of security policies (security rules) and apply them to data structures and service at development time or even later during deployment definitions. AMi then makes sure that these requirements are met before any communication takes place and applies any suitable security measures (e.g. encryption, signatures, etc.) during the communication.
- **Dynamic Multi-Protocol support:** The AMi architecture can select at run-time the best way to communicate with a remote service. Thus, an AMi application can simultaneously talk with legacy services via their predefined protocols (e.g. DDS). It also supports various communication patterns, like Publish-/Subscribe (PubSub), Point-To-Point, or Request-/Reply (RPC).
- **QoS and Software Defined networking:** Where possible the QoS annotations are also used to configure the network using modern Software Defined networking functionality, e.g. to reserve bandwidth.

3.5.2 The Advanced Middleware GE components

The following layer diagram shows the main components of the Advanced Communication Middleware GE.

In the diagram the main communication flow goes from top to bottom for sending data, respectively and from bottom to top for receiving data. As in a typical layer diagram each layer is responsible for specific features and builds on top of the layers below. Some modules are cross cutting and go therefore over several layers (e.g. Security).

Here are some of the highlights of the AMi architecture shown in the diagram:

- AMi clearly separates the definition of WHAT data must be communicated (the communication contract via one of many interface definition languages (IDLs)) from WHERE that data comes from in the application and from HOW that data is transmitted. This *separation of concerns* is critical to support some advanced functionality and be portable to a wide range of services and their communication mechanisms.

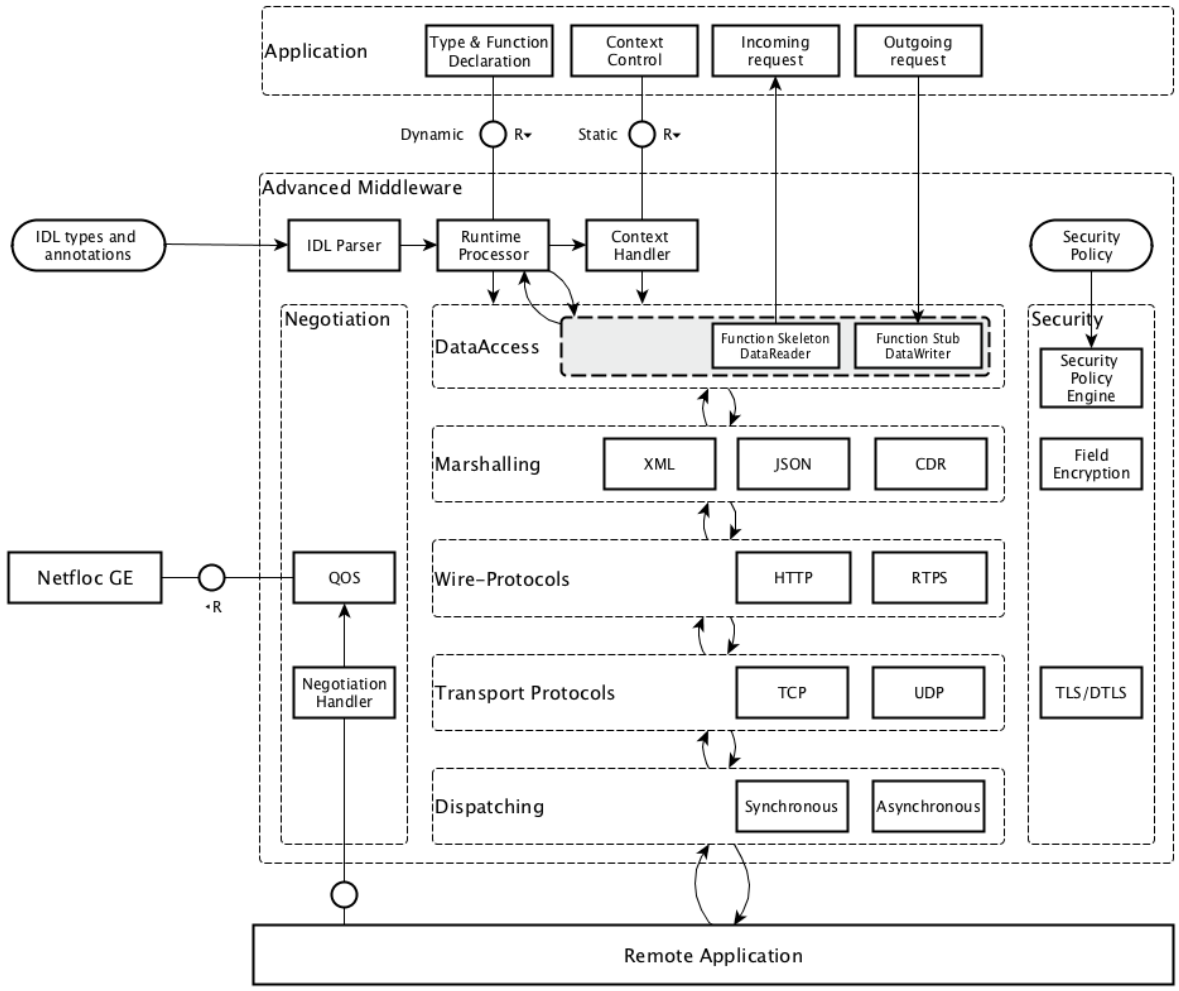

Fig. 3.4: I2ND Middleware Architecture Overview

- AMi supports multiple IDLs to define what data needs to be communicated. On establishing the connection the interface definition of a service are obtained (explicitly or implicitly).
- AMi offers annotations for QoS, security, or other features that can be added to the data declared by the application, to the IDL, as well as later during deployment. They are used by the middleware to automatically implement its functionality by requesting QoS functionality from the network layer or automatically enforcing security measures.
- As the connection to a service is established, both sides choose a common mechanism and protocol (negotiation) to best communicate with each other.

The most efficient transport and protocol method supported by both sides will be selected. AMi has been designed to also support Software Defined Networking in order to configure QOS parameters in the network.

- AMi offers an efficient dispatching mechanism for scheduling incoming request to the correct service implementation.

Below we give a short description of the different layers and components.

API & Data Access

The application accesses the communication middleware using a set of defined function calls provided by the API-layer. They may vary depending on the communication pattern (see below).

The main functionality of the Data Access Layer is to provide the mapping of data types and Function Stubs/Skeletons (request/response pattern) or DataReaders/-Writers (publish/subscribe or point-to-point pattern).

The Advanced Middleware GE provides two variants of this functionality:

- A **basic static compile-time Data-Mapping and generation of Function Stubs/Skeletons or DataReaders/-Writers**, created by a compile time IDL-Parser/Generator from the remote service description, which is provided in an *Interface Definition Language (IDL)* syntax based on the Object Management Group (OMG) IDL (see below), which is submitted as a W3C draft.
- A **dynamic runtime Data-Mapping and invocation of Function or DataReader/-Writer proxies**, by parsing the IDL description of the remote service at runtime and map it to the function/data definition provided by the developer when setting up the connection.

Quality of Service (QoS) parameters and Security Policies may be provided through the API and/or IDL-Annotations. This information will be used by the QoS and Security modules to ensure the requested guarantees.

Depending on the communication pattern, different communication mechanisms will be used.

- For **publish/subscribe** and **point-to-point** scenarios, the DDS services and operations will be provided. When opening connections, a **DataWriter** for publishers/sender and a **DataReader** for subscribers/receivers will be created, which can be used by the application to send or receive DDS messages.
- For **request/reply** scenarios the **Function Stubs/Skeletons** created at compile-time can be used to send or receive requests/replies.

Marshalling

Depending on configuration, communication pattern and type of end-points the data will be serialized to the required transmission format when sending and deserialized to the application data structures when receiving.

- **Common Data Representation (CDR)** an OMG specification used for all DDS/RTPS and high-speed communication.
- **Extensible Markup Language (XML)** for WebService compatibility.
- **JavaScript Object Notation (JSON)** for WebService compatibility.

Wire Protocols

Depending on configuration, communication pattern and type of end-points the matching Wire-Protocol will be chosen.

- For **publish/subscribe** and **point-to-point** patterns the **Real Time Publish Subscribe (RTPS)** Protocol is used.
- For **request/reply** pattern with WebService compatibility the **HTTP** Protocol is used.
- For **request/reply** pattern between DDS end-points the **Real Time Publish Subscribe (RTPS)** Protocol is used.

Dispatching

The dispatching module is supporting various threading models and scheduling mechanisms. The module is providing single-threaded, multi-threaded and thread-pool operation and allows synchronous and asynchronous operation. Priority or time constraint scheduling mechanisms can be specified through QoS parameters.

Transport Mechanisms

Based on the QoS parameters and the runtime-environment the **QoS module** will decide which transport mechanisms and protocols to choose for data transmission.

In Software Defined Networking (SDN) environments, the **QoS module** will interface with the Netfloc GE to get additional network information or even provision the network components to provide the requested quality of service or privacy.

Transport Protocols

All standard transport protocols (TCP, UDP) as well as encrypted tunnels (TLS, DTLS) are supported.

Security

The security module is responsible for authentication of communication partners and will ensure in the whole middleware stack, the requested data security and privacy. The required information can be provided with Security Annotations in the IDL and by providing a security policy via the API.

Negotiation

The negotiation module provides mechanisms to discover or negotiate the optimal transmission format and protocols when peers are connecting. It discovers automatically the participants in the distributed system, searching through the different transports available (shared memory and UDP by default, TCP for WebService compatibility) and evaluates the communication paradigms and and the corresponding associated QoS parameters and security policies.

3.6 Main Interactions

As explained above, the middleware can be used in different communication scenarios. Depending on the scenario, the interaction mechanisms and the set of API-functions for application developers may vary.

3.6.1 API versions

There will be two versions of APIs provided:

- **RPC Static API**
  Static compile-time parsing of IDL and generation of Stub-/Skeletons and DataReader/DataWriter
- **RPC Dynamic API**
  Dynamic runtime parsing of IDL and run-time invocation of operations.

Additionally following features will be provided as API extensions:

- Advanced security policy and QoS parameters
- Publish/subscribe functionality compatible to RPC-DDS and DDS applications

3.6.2 Classification of functions

The API-Functions can be classified in the following groups:

- **Preparation:** statically at compile-time (Static API) or dynamically at run-time (Dynamic API)
  - Declare the local applications datatypes/functions (Dynamic API only)
  - Parsing the Interface Definition of the remote side (IDL-Parser)
  - Generate Stubs-/Skeletons, DataReader-/Writer
  - Build your application against the Stubs-/Skeletons, DataReader-/Writer (Static API only)
- **Initialization:**
  - Create the context (set up the environment, global QoS/Transport/Security policy,...)
  - Open connection (provide connection specific parameters: QoS/Transport/Security policy, Authentication, Tunnel encryption, Threading policy,...)
- **Communication**
  - Send Message/Request/Response (sync/async, enforce security)
  - Receive Message/Request/Response (sync/async, enforce security)
  - Exception Handling
- **Shutdown**
  - Close connection (cleanup topics, subscribers, publishers)
  - Close the context (Free resources)

Detailed description of the APIs and tools can be found in the User and Developer Guide, which will be updated for every release of the Advanced Middleware GE.

3.7 Basic Design Principles

Implementations of the Advanced Middleware GE have to comply to the following basic design principles:

- All modules have to provide defined and documented APIs.
- Modules may only be accessed through these documented APIs and not use any internal undocumented functions of other modules.
- Modules in the above layer model may only depend on APIs of lower level modules and never access APIs of higher level modules.
- All information required by lower level modules has to be provided by the higher levels modules through the API or from a common configuration.

- If a module provides variants of internal functionalities (e.g. Protocols, Authentication Mechanisms, ...) these should be encapsulated as Plugins with a defined interface.

CHAPTER 4

Open Specifiation

4.1 Preface

Within this document you find a self-contained open specification of a FI-WARE generic enabler, please consult as well the FI-WARE Product Vision, the website on http://www.fiware.org and similar pages in order to understand the complete context of the FI-WARE project.

4.2 Copyright

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

4.3 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

4.4 Overview

This specification describes the Advanced Middleware GE, which enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FIWARE GEs.

Middleware in general provides a software layer between the application and the communication network and allows application to abstract from the intricacies of how to send a piece of data to a service offered by a another application and possibly return results. The middleware offers functionality to find and establish a connection to a service, negotiate the best wire and transport protocols, access the applications data structures and encode the necessary data in a format suitable for the chosen protocol, and finally send that data and possibly receive results in return. In a similar way an application can use the middleware also to offer services to other applications by registering suitable service functionality and interfaces, which can then be used as targets of communication.

4.5 Basic Concepts

In this section several basic concepts of the Advanced Communication Middleware are explained. We assume that the reader is familiar with the basic functionality of communication middleware like CORBA or WebServices.

Fig. 4.1: Publish/Subscribe Pattern

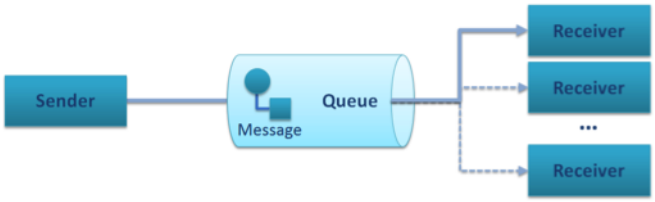

Fig. 4.2: Point-To-Point Pattern

4.5.1 Communication Patterns

We can distinguish between three main different messaging patterns, Publish/Subscribe, Point-to-Point, and Request/Reply, shown schematically below:

All available middleware technologies implement one or more of these messaging patterns and may incorporate more advanced patterns on top of them. Most RPC middleware is based on the Request/Reply pattern and more recently, is extended towards support of Publish/Subscribe and/or the Point-To-Point pattern.

W3C Web Service standards define a Request/Reply and a Publish/Subscribe pattern which is built on top on that (WS-Notification). CORBA, in a similar way, build its Publish/Subscribe pattern (Notification Service) on top of a Request/Reply infrastructure. In either case the adopted architecture is largely ruled by historical artifacts instead of performance or functional efficiency. The adopted approach is to emulate the Publish/Subscribe pattern on top of the more complex pattern thus inevitably leading to poor performance and complex implementations.

The approach of the Advanced Middleware takes the other direction. It provides native Publish/Subscribe and implements the Request/Reply pattern on top of this infrastructure. Excellent results can be achieved since the Publish/Subscribe is a meta-pattern, in other words a pattern generator for Point-To-Point and Request/Reply and potential alternatives.

4.5.2 Interface Definition Language (IDL)

The Advanced Middleware GE supports a novel IDL to describe the Data Types and Operations. Following is a list of the main features it supports:

- **IDL, Dynamic Types & Application Types**: It support the usual schema of IDL compilation to generate support code for the data types.
- **IDL Grammar**: An OMG-like grammar for the IDL as in DDS, Thrift, ZeroC ICE, CORBA, etc.

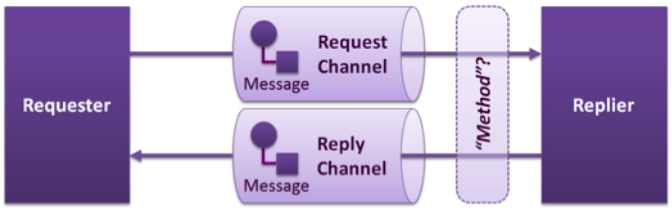

Fig. 4.3: Request/Reply Pattern

- **Types**: Support of simple set of basic types, structs, and various high level types such as lists, sets, and dictionaries (maps).
- **Type inheritance, Extensible Types, Versioning**: Advanced data types, extensions, and inheritance, and other advanced features will be supported.
- **Annotation Language**: The IDL is extended with an annotation language to add properties to the data types and operations. These will, for example, allows adding security policies and QoS requirements.
- **Security**: The IDL allows for annotating operations and data types though the annotation feature of our IDL, allowing setting up security even at the field level.

4.6 Generic Architecture

4.6.1 General Note

In contrast to other GEs, the Advanced Middleware GE is not a standalone service running in the network, but a set of compile-/runtime tools and a communication library to be integrated with the application.

The Advanced Middleware (AMi) architecture presented here offers a number of key advantages over other available middleware implementations:

- **High-Level Service Architecture:** AMi offers applications a high-level architecture that can shield them from the complexities and dangers of network programming. When applications declare services and data structures they can annotate them with the QoS, security, and other requirements while AMi automatically implement them. Thus application developers can exclusively focus on the application functionality.
- **Security:** The network is the main security threat to most applications today but existing middleware has offered only limited security functionality that has often been added as an afterthought and requires the application developer (who are often not security experts) to configure the security functionality. Instead, AMi offers *Security by Design* where security has been designed into the architecture from the start. Applications can simply declare their security needs in the form of security policies (security rules) and apply them to data structures and service at development time or even later during deployment definitions. AMi then makes sure that these requirements are met before any communication takes place and applies any suitable security measures (e.g. encryption, signatures, etc.) during the communication.
- **Dynamic Multi-Protocol support:** The AMi architecture can select at run-time the best way to communicate with a remote service. Thus, an AMi application can simultaneously talk with legacy services via their predefined protocols (e.g. DDS). It also supports various communication patterns, like Publish-/Subscribe (PubSub), Point-To-Point, or Request-/Reply (RPC).
- **QoS and Software Defined networking:** Where possible the QoS annotations are also used to configure the network using modern Software Defined networking functionality, e.g. to reserve bandwidth.

4.6.2 The Advanced Middleware GE components

The following layer diagram shows the main components of the Advanced Communication Middleware GE.

In the diagram the main communication flow goes from top to bottom for sending data, respectively and from bottom to top for receiving data. As in a typical layer diagram each layer is responsible for specific features and builds on top of the layers below. Some modules are cross cutting and go therefore over several layers (e.g. Security).

Here are some of the highlights of the AMi architecture shown in the diagram:

- AMi clearly separates the definition of WHAT data must be communicated (the communication contract via one of many interface definition languages (IDLs)) from WHERE that data comes from in the application and from HOW that data is transmitted. This *separation of concerns* is critical to support some advanced functionality and be portable to a wide range of services and their communication mechanisms.

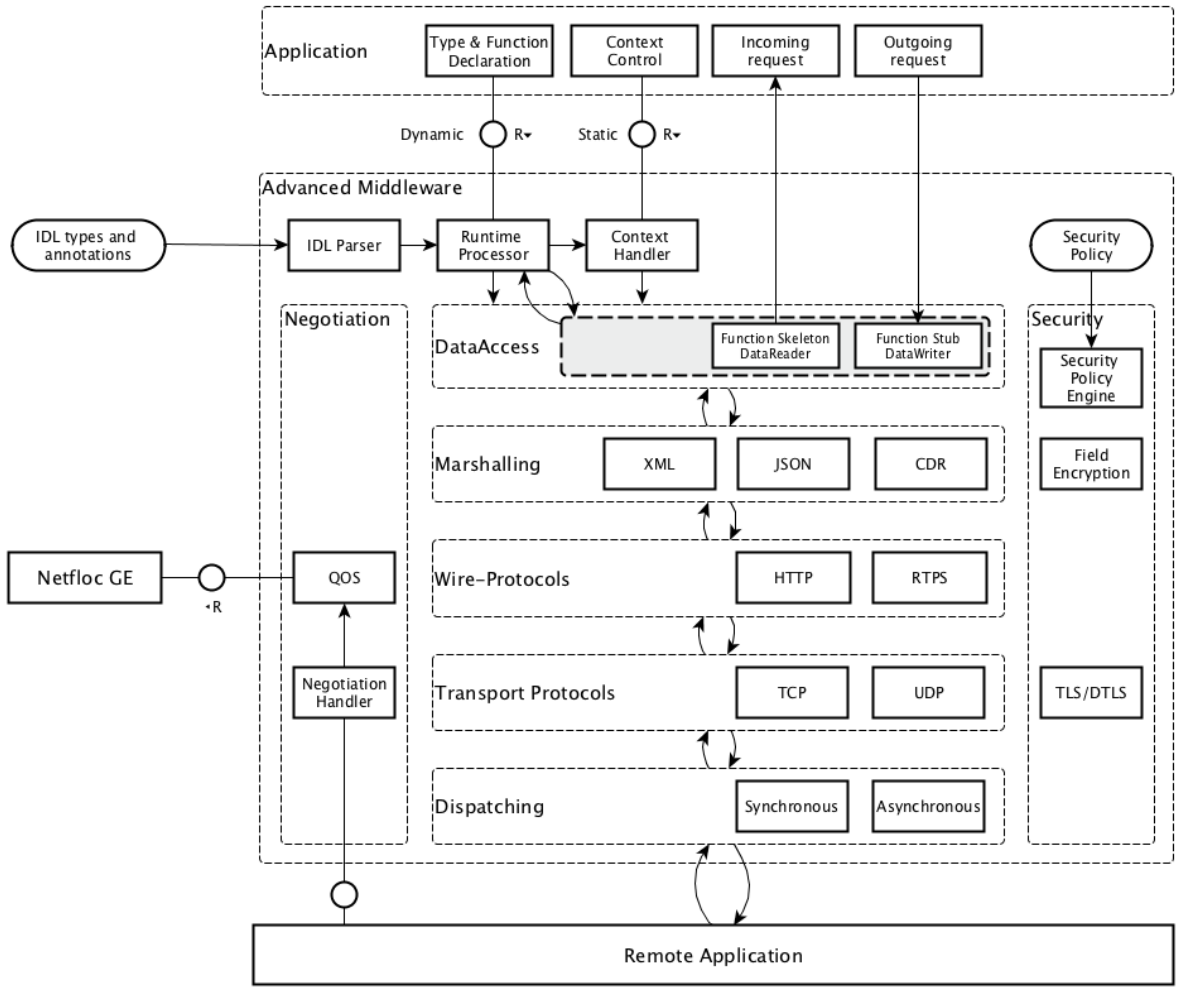

Fig. 4.4: I2ND Middleware Architecture Overview

- AMi supports multiple IDLs to define what data needs to be communicated. On establishing the connection the interface definition of a service are obtained (explicitly or implicitly).
- AMi offers annotations for QoS, security, or other features that can be added to the data declared by the application, to the IDL, as well as later during deployment. They are used by the middleware to automatically implement its functionality by requesting QoS functionality from the network layer or automatically enforcing security measures.
- As the connection to a service is established, both sides choose a common mechanism and protocol (negotiation) to best communicate with each other.

The most efficient transport and protocol method supported by both sides will be selected. AMi has been designed to also support Software Defined Networking in order to configure QOS parameters in the network.

- AMi offers an efficient dispatching mechanism for scheduling incoming request to the correct service implementation.

Below we give a short description of the different layers and components.

API & Data Access

The application accesses the communication middleware using a set of defined function calls provided by the API-layer. They may vary depending on the communication pattern (see below).

The main functionality of the Data Access Layer is to provide the mapping of data types and Function Stubs/Skeletons (request/response pattern) or DataReaders/-Writers (publish/subscribe or point-to-point pattern).

The Advanced Middleware GE provides two variants of this functionality:

- A **basic static compile-time Data-Mapping and generation of Function Stubs/Skeletons or DataReaders/-Writers**, created by a compile time IDL-Parser/Generator from the remote service description, which is provided in an *Interface Definition Language (IDL)* syntax based on the Object Management Group (OMG) IDL (see below), which is submitted as a W3C draft.
- A **dynamic runtime Data-Mapping and invocation of Function or DataReader/-Writer proxies**, by parsing the IDL description of the remote service at runtime and map it to the function/data definition provided by the developer when setting up the connection.

Quality of Service (QoS) parameters and Security Policies may be provided through the API and/or IDL-Annotations. This information will be used by the QoS and Security modules to ensure the requested guarantees.

Depending on the communication pattern, different communication mechanisms will be used.

- For **publish/subscribe** and **point-to-point** scenarios, the DDS services and operations will be provided. When opening connections, a **DataWriter** for publishers/sender and a **DataReader** for subscribers/receivers will be created, which can be used by the application to send or receive DDS messages.
- For **request/reply** scenarios the **Function Stubs/Skeletons** created at compile-time can be used to send or receive requests/replies.

Marshalling

Depending on configuration, communication pattern and type of end-points the data will be serialized to the required transmission format when sending and deserialized to the application data structures when receiving.

- **Common Data Representation (CDR)** an OMG specification used for all DDS/RTPS and high-speed communication.
- **Extensible Markup Language (XML)** for WebService compatibility.
- **JavaScript Object Notation (JSON)** for WebService compatibility.

Wire Protocols

Depending on configuration, communication pattern and type of end-points the matching Wire-Protocol will be chosen.

- For **publish/subscribe** and **point-to-point** patterns the **Real Time Publish Subscribe (RTPS)** Protocol is used.
- For **request/reply** pattern with WebService compatibility the **HTTP** Protocol is used.
- For **request/reply** pattern between DDS end-points the **Real Time Publish Subscribe (RTPS)** Protocol is used.

Dispatching

The dispatching module is supporting various threading models and scheduling mechanisms. The module is providing single-threaded, multi-threaded and thread-pool operation and allows synchronous and asynchronous operation. Priority or time constraint scheduling mechanisms can be specified through QoS parameters.

Transport Mechanisms

Based on the QoS parameters and the runtime-environment the **QoS module** will decide which transport mechanisms and protocols to choose for data transmission.

In Software Defined Networking (SDN) environments, the **QoS module** will interface with the Netfloc GE to get additional network information or even provision the network components to provide the requested quality of service or privacy.

Transport Protocols

All standard transport protocols (TCP, UDP) as well as encrypted tunnels (TLS, DTLS) are supported.

Security

The security module is responsible for authentication of communication partners and will ensure in the whole middleware stack, the requested data security and privacy. The required information can be provided with Security Annotations in the IDL and by providing a security policy via the API.

Negotiation

The negotiation module provides mechanisms to discover or negotiate the optimal transmission format and protocols when peers are connecting. It discovers automatically the participants in the distributed system, searching through the different transports available (shared memory and UDP by default, TCP for WebService compatibility) and evaluates the communication paradigms and and the corresponding associated QoS parameters and security policies.

4.7 Main Interactions

As explained above, the middleware can be used in different communication scenarios. Depending on the scenario, the interaction mechanisms and the set of API-functions for application developers may vary.

4.7.1 API versions

There will be two versions of APIs provided:

- **RPC Static API**
  Static compile-time parsing of IDL and generation of Stub-/Skeletons and DataReader/DataWriter
- **RPC Dynamic API**
  Dynamic runtime parsing of IDL and run-time invocation of operations.

Additionally following features will be provided as API extensions:

- Advanced security policy and QoS parameters
- Publish/subscribe functionality compatible to RPC-DDS and DDS applications

4.7.2 Classification of functions

The API-Functions can be classified in the following groups:

- **Preparation:** statically at compile-time (Static API) or dynamically at run-time (Dynamic API)
  - Declare the local applications datatypes/functions (Dynamic API only)
  - Parsing the Interface Definition of the remote side (IDL-Parser)
  - Generate Stubs-/Skeletons, DataReader-/Writer
  - Build your application against the Stubs-/Skeletons, DataReader-/Writer (Static API only)
- **Initialization:**
  - Create the context (set up the environment, global QoS/Transport/Security policy,...)
  - Open connection (provide connection specific parameters: QoS/Transport/Security policy, Authentication, Tunnel encryption, Threading policy,...)
- **Communication**
  - Send Message/Request/Response (sync/async, enforce security)
  - Receive Message/Request/Response (sync/async, enforce security)
  - Exception Handling
- **Shutdown**
  - Close connection (cleanup topics, subscribers, publishers)
  - Close the context (Free resources)

Detailed description of the APIs and tools can be found in the User and Developer Guide, which will be updated for every release of the Advanced Middleware GE.

4.8 Basic Design Principles

Implementations of the Advanced Middleware GE have to comply to the following basic design principles:

- All modules have to provide defined and documented APIs.
- Modules may only be accessed through these documented APIs and not use any internal undocumented functions of other modules.
- Modules in the above layer model may only depend on APIs of lower level modules and never access APIs of higher level modules.
- All information required by lower level modules has to be provided by the higher levels modules through the API or from a common configuration.

- If a module provides variants of internal functionalities (e.g. Protocols, Authentication Mechanisms, ...) these should be encapsulated as Plugins with a defined interface.

4.9 Detailed Specifications

Following is a list of Open Specifications linked to this Generic Enabler. Specifications labelled as "PRELIMINARY" are considered stable but subject to minor changes derived from lessons learned during last interactions of the development of a first reference implementation planned for the current Major Release of FI-WARE. Specifications labelled as "DRAFT" are planned for future Major Releases of FI-WARE but they are provided for the sake of future users.

4.9.1 Open API Specifications

- Advanced Middleware IDL Specification
- Advanced Middleware RPC API Specification
- Advanced Middleware RPC Dynamic Types API Specification
- Advanced Middlware Pub-Sub API Specification

4.10 Re-utilised Technologies/Specifications

The technologies and specifications re-used in this GE are:

- RTPS - Realtime Publish Subscribe Wire Protocol (OMG Standard V2.1) : Used as the protocol for the Pub/sub mechanism
- CDR - Common Data Representation (CDR Page 4 of the PDF) : Used as the serialization mechanism

4.11 Terms and definitions

This section comprises a summary of terms and definitions introduced during the previous sections. It intends to establish a vocabulary that will be help to carry out discussions internally and with third parties (e.g., Use Case projects in the EU FP7 Future Internet PPP). For a summary of terms and definitions managed at overall FI-WARE level, please refer to FIWARE Global Terms and Definitions

CHAPTER 5

Advanced Middleware RPC API Specification

Date: 18th January 2016

- Version: *0.4.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

5.1 Abstract

Ahe Advanced Middleware GE enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FIWARE GEs. The **Middleware RPC API Specification** describes the interfaces and procedures to do Request/Reply type Remote Procedure Calls (RPC).

It provides basic static compile-time Data-Mapping and generation of Function Stubs/Skeletons, created by a compile time IDL-Parser/Generator from the remote service description, which is provided in the Advanced Middleware Interface Definition Language (IDL) syntax, which is based on the Object Management Group (OMG) IDL draft submitted to W3C.

5.2 Status of this Document

Date	Description
7-November-2014	Release 0.1.0
8-April-2015	Release 0.2.0
10-October-2015	Release 0.3.0
18-January-2016	Release 0.4.0

5.3 Introduction

5.3.1 Purpose

This document attempts to describe the Advanced Middleware RPC API.

5.3.2 Reference Material

- Advanced Middleware IDL Specification
- Advanced Middleware RPC Dynamic Types API Specification

5.4 A quick Example

Before the description of the public Advanced Middleware RPC API, a quick example is provided. It shows how a simple client is created, as well as a simple server. The example uses the following Advanced Middleware interface definition:

```
service Calculator
{
    i32 add(i32 num1, i32 num2);
};
```

5.4.1 Creating a client

The following code shows how to instanciate and start a client and execute a call to the server:

```
Context context = Advanced Middleware.createContext();
Connection connection = context.connect("tcp://192.168.1.18:8080?serialization=cdr");
CalculatorClient client = connection.getServiceProxy(CalculatorClient.class);

int result = client.add(3,4);
```

5.4.2 Creating a server

The following code shows how to instanciate and start a server:

```
Context context = Advanced Middleware.createContext();
Server server = context.createServer();
Service service = context.createService();

// User creates its implementation of the Middleware IDL service.
Calculator calculator_impl = new CalculatorImpl();

service.register(calculator_impl);
server.addService(service, "tcp://0.0.0.0:8080", "cdr");

server.run();
```

5.5 API Overview

This section enumerates and describes the classes provided by Advanced Middleware RPC API.

5.5.1 Main entry point

org.fiware.kiara.Kiara

This class is the main entry point to use the Advanced Middlware. It creates or provides implementation of the top level Advanced Middleware interfaces, especially the `Context`.

Functions:

- **getTypeDescriptorBuilder**: This function returns an instance of the type descriptor builder. It is a part of the dynamic API and is described here.
- **getDynamicValueBuilder**: This function returns an instance of the dynamic value builder. It is a part of the dynamic API and is described here.
- **createContext**: This function creates a new instance of the Context class, which is described below.
- **shutdown**: This function closes and releases all internal Advanced Middleware structures (e.g. stops all pending tasks). Call this before you exit your application.

5.5.2 Common interfaces

org.fiware.kiara.Context

This interface is the starting point to use the Advanced Middleware. It holds the configuration of the middleware and hides the process of negotiation, selection, and configuration of the correct implementation classes. Also it provides users a way to instantiate Advanced Middleware components.

Functions:

- **connect**: This function creates a new connection to the server. This connection might be used by proxies to send requests to the server.
- **createTransport**: This function provides a direct way to create a specific network `Transport` instance which can be configured for specific use cases.
- **createSerializer**: This function provides a direct way to create a specific `Serializer` instance which can be configured for specific use cases.
- **createServer**: This function creates a new `Server` instance used to add `Service` instances.
- **createService**: This function creates a new `Service` instance used to register `Servant` instances.

5.5.3 Network transports

org.fiware.kiara.transport.Transport

This interface provides a basic abstraction for network transport implementations. To create a `Transport` instance directly, the developer must use the factory method `createTransport` of the interface *org.fiware.kiara.Context*, which will return a compliant network transport implementation.

Functions:

- **getTransportFactory**: This function returns an instance of the factory class used to create this transport instance.

---

org.fiware.kiara.transport.ServerTransport

This interface provides an abstraction for a server-side connection endpoint waiting for incoming connections.

Functions:

- **getTransportFactory**: This function returns an instance of a factory class which was used to create this server transport instance.
- **setDispatchingExecutor**: This function sets executor service used for dispatching incoming messages.
- **getDispatchingExecutor**: Returns executor service previously set.
- **isRunning**: Returns true if server is up and waiting for incoming connections.

- **startServer**: Starts server.
- **stopServer**: Stops server.
- **getLocalTransportAddress**: Returns transport address to which this server is bound.

org.fiware.kiara.client.AsyncCallback

This interface provides an abstraction used by the client to return the server's reply when the call was asynchronous.

Functions:

- **onSuccess**: This function will be called when the remote function call was successfull. It must be implemented by the user.
- **onFailure**: This function will be called when the remote function call was *not* successfull.It must be implemented by the user.

5.5.4 Server API

org.fiware.kiara.server.Server

Using this interface, users can start up multiple services on different ports. The implementation uses serialization mechanisms and network transports to listen for client requests and executes the proper `Servant` implementation. The optional negotiation protocol provides automatic discovery of all available services via the HTTP protocol.

Functions:

- **enableNegotiationService**: Enables the negotiation service on the specified port and configuration path.
- **disableNegotiationService**: Disables the negotiation service.
- **addService**: This function registers the service on a specified URL and with a specified serialization protocol.
- **removeService**: Removes a previously registered service.
- **run**: Starts the server.

---

org.fiware.kiara.server.Service

This interface represent a service that can be registered with the server.

Functions:

- **register:** Register a `Servant` object or `DynamicHandler` with the service.
- **loadServiceIDLFromString:** Load the service IDL from a string. This function is only required when the service is handled via dynamic handlers.

---

org.fiware.kiara.server.Servant

This interface provides an abstraction used by the server to execute the provided functions when a client request is received.

Functions:

- **getServiceName**: Returns the name of the service implemented by this servant.

- **process**: This function processes the incoming request message and returns the produced response message. It is automatically generated.

5.5.5 Dependent API

This subsection contains the interfaces and classes that are dependent from the user Advanced Middleware IDL definition. In the static version of the Advanced Middleware implementation these interfaces and classes should be generated by the compile time preprocessor.
This section uses the example in section *API Usage Examples*.

---

x.y.<IDL-ServiceName>

This interface is a mapping of the Advanced Middleware IDL service. It exposes the service's procedures. All classes that implement these service's procedures, have to inherit from this interface. For example the imlementation of the servant have to inherit from this interface, allowing the user to implement the service's procedures.

Functions:

- **add**: This function is the mapping of the Advanced Middleware IDL service procedure `add()`.

---

x.y.<IDL-ServiceName>Async

This interface is a mapping of the Advanced Middleware IDL service. It exposes the asynchronous version of the service's procedures. All classes that that implement these service's asynchronous procedures have to inherit from this interface.

Functions:

- **add**: This function is the asynchronous version of the Advanced Middleware IDL service's procedure `add()`. It has no return value.

---

x.y.<IDL-ServiceName>Process

This class is a mapping of the Advanced Middleware IDL service. It provides the asynchronous version of the service's processing procedures.

Functions:

- **add_processAsync**: This function is the asynchronous version of the Advanced Middleware IDL service's process procedure. It has no return value.

---

x.y.<IDL-ServiceName>Client

This interface provides the synchronous and asynchronous version of the Advanced Middleware IDL service, because it implements the previously described interfaces x.y.<IDL-ServiceName> and x.y.<IDL-ServiceInterface>Async. The Advanced Middleware IDL service proxy will implement this interface, allowing the user to call the service's remote procedures synchronously or asynchronously. It is only used on the client side in order to make the Proxy to implement all the functions for this service (both synchronous and asynchronous).

Functions:

- **add**: Function inherited from *x.y.<IDL-ServiceName>* interface. This function is the mapping of the Advanced Middleware IDL service.
- **add**: Function inherited from *x.y.<IDL-ServiceName>Async* interface. This function is the asynchronous version of the Advanced Middleware IDL service's procedure.

---

x.y.<IDL-ServiceName>Proxy

This class encapsulates the implementation of the interface *x.y.<IDL-ServiceName>Client*. It provides the logic to call the Advanced Middleware IDL service's remote procedures, synchronously or asynchronously.

Functions:

- **add**: Function inherited from *x.y.<IDL-ServiceName>Client* interface. This function is the mapping of the Advanced Middleware IDL service.
- **add**: Function inherited from *x.y.<IDL-ServiceName>Client* interface. This function is the asynchronous version of the Advanced Middleware IDL service's procedure.

---

x.y.<IDL-ServiceName>Servant

This abstract class can be used by users to implement the Advanced Middleware IDL service's procedures. This class implements the interface *org.fiware.kiara.server.Servant*, providing the mechanism the server will use to call the user's procedure implementations. Also it inherits from the interface *x.y.<IDL-ServiceName>* leaving the implementation of this functions to the user.

5.6 Detailed API

This section defines in detail the API provided by the classes defined above.

5.6.1 Main entry point

org.fiware.kiara.Kiara			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getTypeDescriptorBuilder		TypeDescriptorBuilder	
getDynamicValueBuilder		DynamicValueBuilder	
createContext		Context	
shutdown		void	

5.6.2 Common interfaces

org.fiware.kiara.Context			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
connect		Connection	IOException
	url	String	
connect		Connection	IOException
	transport	Transport	
	serializer	Serializer	
createService		Service	
createServer		Server	
createTransport		Transport	IOException
	String	url	
createServerTransport		ServerTransport	IOException
	url	String	
createSerializer		Serializer	IOException
	name	String	

5.6.3 Network transports

org.fiware.kiara.transport.Transport			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getTransportFactory		TransportFactory	

5.6.4 Dependent API

Cause the described classes in this section are dependant of the Advanced Middleware IDL service, this section will use the example in section *API Examples* to define them.

x.y.<IDL-ServiceName>			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
add		int	
	num1	int	
	num2	int	

x.y.<IDL-ServiceName>Async			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
add		void	
	num1	int	
	num2	int	
	callback	AsyncCallback<Integer>	

x.y.<IDL-ServiceName>Client			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
Inherited from x.y.<IDL-ServiceName> and x.y.<IDL-ServiceName>Async			

x.y.<IDL-ServiceName>Proxy			
Attributes			
Name	*Type*		
m_ser	org.fiware.kiara.serialization.Serializer		
m_transport	org.fiware.kiara.transport.Transport		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
Inherited from x.y.<IDL-ServiceName> and x.y.<IDL-ServiceName>Async			

x.y.<IDL-ServiceName>Servant			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getServiceName		String	
process		TransportMessage	
	ser	Serializer	
	message	TransportMessage	
	transport	Transport	
	messageId	Object	
	bis	BinaryInputStream	

x.y.<IDL-ServiceName>Process			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
add_processsAsync		void	
	message	TransportMessage	
	ser	Serializer	
	callback	AsyncCallback	

5.7 API Usage Examples

Examples used in this section are based on the following Advanced Middleware IDL:

```
service Calculator
{
    i32 add(i32 num1, i32 num2);
};
```

5.7.1 Client API

Direct connection to remote service

This example shows how to create a direct connection to a server using the TCP transport and the CDR serialization. After it creates the connection, the service proxy is instantiated and used to call a remote procedure.

```
Context context = Kiara.createContext();
Connection connection = context.connect("tcp://192.168.1.18:8080?serialization=cdr");
CalculatorClient client = connection.getServiceProxy(CalculatorClient.class);

int result = client.add(3,4);
```

`Transport` and `Serialization` instances are implizitly created by the connection, based on the string parameter of the `connect` method.

Secure connection to remote service

KIARA also provides a way to stablish secure TCP connections with servers running in secure mode. To to so, simply change the connection URL to use TCPS instead of TCP.

```
Context context = Kiara.createContext();
Connection connection = context.connect("tcps://192.168.1.18:8080?serialization=cdr");
CalculatorClient client = connection.getServiceProxy(CalculatorClient.class);

int result = client.add(3,4);
```

Note that a client running normal (non secure) TCP transport can initialize a connection with a non-secure TCP server, but this does not happen in the opposite way.

Explicitly instanciate and configure Advanced Middleware components

This examples shows how to create a direct connection as above, but using a TCP transport and CDR serialization created and configured explicitly by the user.

```
Context context = Kiara.createContext();
// User instantiates a transport object which can be configured later.
Transport transport = context.createTransport("tcp://192.168.1.18:8080");
// User instantiates a serialization object which can be configured later.
Serializer serializer = context.createSerializer("cdr");
Connection connection = context.connect(transport, serializer);
CalculatorClient client = connection.getServiceProxy(CalculatorClient.class);

int result = client.add(3,4);
```

5.7.2 Server API

Providing a service

This examples shows how to create a server and add a service to it.

```
Context context = Kiara.createContext();
Server server = context.createServer();
Service service = context.createService();

// User creates and registers it's implementation of the servant.
Calculator calculator_impl = new CalculatorServantImpl();
service.register(calculator_impl);

// Add the service to the server
server.addService(service, "tcp://0.0.0.0:8080", "cdr");

server.run();
```

`Transport` and `Serialization` instances are implizitly created by the connection, based on the string parameters of the `addService` method.

Explicitly instanciate and configure Advanced Middleware components

This example shows how to provide a service as above, but using a TCP transport and CDR serialization created and configured explicitly by the user.

```
Context context = Kiara.createContext();
Server server = context.createServer();
Service service = context.createService();

// User creates and registers it's implementation of the servant.
Calculator calculator_impl = new CalculatorServantImpl();
service.register(calculator_impl);

// Transport and Serializer are expicitly created ...
Transport transport = context.createTransport("tcp://0.0.0.0:8080");
Serializer serializer = context.createSerializer("cdr");

// ... and bound to the service when adding it to the server
server.addService(service, transport, serializer);

server.run();
```

Secure TCP server

KIARA allows the server to initialize using a secure TCP layer so that only clients using the same protocol can establish a connection with it.

The transport used to handle secure connections is SSL v3.1 (Secure Sockets Layer) over TLS v1.2 (Transport Layer Security). In order to start a secure server, the only change the user has to take into account is changing the URL of the server so that it uses TCPS instead of TCP.

This example shows how to create a secure server and add a service to it.

```
Context context = Kiara.createContext();
Server server = context.createServer();
Service service = context.createService();

// User creates and registers it's implementation of the servant.
Calculator calculator_impl = new CalculatorServantImpl();
```

```
service.register(calculator_impl);

// Add the service to the server
server.addService(service, "tcps://0.0.0.0:8080", "cdr");

server.run();
```

`Transport` and `Serialization` instances are implicitly created by the connection, based on the string parameters of the `addService` method.

CHAPTER 6

Advanced Middleware RPC Dynamic Types API Specification

Date: 18th January 2016

- Version: *0.4.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

6.1 Abstract

The Advanced Middleware GE enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FIWARE GEs. The **Middleware RPC Dynamic Types API Specification** describes the extensions to the **Middleware RPC API Specification** to do *dynamic* Request/Reply type Remote Procedure Calls (RPC).

It provides a *dynamic runtime Data-Mapping and invocation of Function proxies*, by parsing the IDL description of the remote service at runtime and map it to the function/data definition provided by the developer when setting up the connection.

6.2 Status of this Document

Date	Description
30-January-2015	Frist release
04-February-2015	Update after review meeting
08-April-2015	Release 0.2.0
10-October-2015	Release 0.3.0
18-January-2016	Release 0.4.0

6.3 Introduction

6.3.1 Purpose

The purpose of this document is to specify the dynamic Remote Procedure Call (RPC) Application Programming Interface (API) for the Advanced Middleware GE.

6.3.2 Reference Material

- Advanced Middleware IDL Specification
- Advanced Middleware RPC API Specification

6.4 A quick Example

Before the description of the public Advanced Middleware RPC Dynamic Types API, a quick example is provided. This example shows how a client should use this dynamic API framework to call a remote server and how a server can send the response.

First of all, let's bear in mind that the server provides an IDL defining the services and their functions. The example uses the following Advanced Middleware interface definition:

```
service Calculator
{
    i32 add(i32 num1, i32 num2);
};
```

6.4.1 Loading the services' definitions

To be able to call remote functions dynamically, it is required to know the services and what functions they offer. The use case is to load the `TypeCode`elements when creating the connection, so it will be the `Connection` class which offers an API to get all the data definitions from the server.

6.4.2 Creating a client

On the client side, there is no differnce between the dynamic and the static API from the developers point of view in terms of creating the connection. In the dynamic case this connection will be used to obtain all the types and functions offered by the server.

This means, the user has to create a connection and then use it to get a definition of the function he wants to execute. Let's walk through it based on the following example.

```
// Create context and connect with the server
Context context = Kiara.createContext();
Connection connection = context.connect("kiara://127.0.0.1:8080/service");

// Get a generic proxy based on the service interface
 DynamicProxy client = connection.getDynamicProxy("Calculator");
```

When connecting to the server, the scheme specified in the URI is used by the Context to decide from where to download the information. In this example, the scheme "kiara" means the connection information is going to be downloaded from the server and then used in the negotiation process. Otherwise, information such as the transport protocol and serialization mechanism must be specified in the URI itself.

Before being able to call remotely a function on the server, the client will need to have access to its functions, and in a typical RPC framework, this can be done by using a Proxy. The class named `DynamicProxy` allows the user to have access to this information from the data that has been downloaded from the server.

To do so, the `Connection` object offers a function called `getDynamicProxy`, which looks inside the dynamic data types created when connecting to the server and retrieves a `DynamicProxy` whose name is the same as the service name specified as a parameter.

Once the user has obtained this `DynamicProxy`, all the functions defined inside the service are available. To use them, two objects are necessary, the `DynamicFunctionRequest` and the `DynamicFunctionResponse`.

The `DynamicFunctionRequest` object is created at run-time by using the name of the function the user wants to execute on the server's side. If there is a function whose name fits the one specified, this object will be filled with all the `DynamicValue` objects necessary to execute the function.

On the other hand, the DynamicFunctionResponse object will be created and filled with the response obtained from the server after the execution is finished (either if it finished properly or not).

```
// Create the function request
DynamicFunctionRequest request = dclient.createFunctionRequest("add");
((DynamicPrimitive) request.getParameterAt(0)).set(3.5);
((DynamicPrimitive) request.getParameterAt(1)).set(5.2);

// Execute the Remote Procedure Call
DynamicFunctionResponse response = drequest.execute();
```

In this example, the `createFunctionRequest` method has been executed specifying "add as" the function name. Therefore, the `DynamicFunctionRequest` object will have two primitive `DynamicValue` objects (`DynamicPrimitive`) inside (one for each parameter defined in the IDL description of the function). The user can easily modify these values and call the execute method on the request object, obtaining this way a `DynamicFunctionResponse` which holds the result of the function execution.

The `execute` method will have all the business logic so that the service name, the operation name, message ID, etc. as well as all the parameters are serialized properly according to the function that is going to be executed.

The same thing happens with the return type of each function. Depending on the `DynamicValue` that defines it, a different deserialization method will be executed. By using this method, the user only has to specify which function must be executed on the server's side, and all the information will be (de)serialized automatically.

In order to know if the function finished the way it should, the `DynamicFunctionResponse` object offers a function named `isException`, which will return true if and only if the function did raise an exception. The following code snippet shows this behaviour:

```
// Check RPC result
if (dresponse.isException()) {
    DynamicData result = dresponse.getReturnValue();
    System.out.println("Exception = " + (DynamicPrimitive) result);
} else {
    DynamicData result = dresponse.getReturnValue();
    System.out.println("Result = " + ((DynamicPrimitive) result).get());
}
```

6.4.3 Creating a secure client

KIARA allows to use the Dynamic RPC API to connecto to a secure TCP server. This behaviour does not apply to the dynamic API, and therefore it can be found in the Advanced Middleware RPC API Specification document, in the section API Usage Examples.

6.5 API Overview

This section enumerates and describes the classes provided by Advanced Middleware Dynamic Types RPC API.

6.5.1 Main entry point

org.fiware.kiara.Kiara

This class is the main entry point to use Advanced Middleware middleware. It creates or provides implementation of top level Advanced Middleware interfaces, especially `Context`.

Functions:

- **getTypeDescriptorBuilder**: This function returns an instance of the type `DescriptorBuilder` described below.
- **getDynamicValueBuilder**: This function returns an instance of the `DynamicValueBuilder` described below.
- **createContext**: This function creates a new instance of the Context class, which is part of the public Advanced Middleware RPC API.
- **shutdown**: This function closes releases all internal Advanced Middleware structures, and is a part of the public Advanced Middleware RPC API.

6.5.2 Serialization mechanisms

org.fiware.kiara.serialization.Serializer

This interface is part of the public Advanced Middleware RPC API.

org.fiware.kiara.serialization.impl.Serializable

This interface is the one that must be implemented by all the used defined data types in order to be serializable. It defines the methods `serialize` and `deserialize` for each data type. This class will not be described in this document, for more information take a look at the Advanced Middleware RPC API document.

6.5.3 Client API

org.fiware.kiara.client.Connection

The `Connection` interface manages the connection to the server. It holds the required `Transport` objects and `Serialization` objects. Also it can create these object automatically depending on the server information. The connection provides the service proxy interfaces, which will be used by the application to call remote functions.

Functions:

- **getDynamicProxy**: This function looks in the endpoint for a service whose name is the same as the one specified as a parameter, and creates a new `DynamicProxy` representing that service. This `DynamicProxy` will provide the user with all the functions defined in such a service.

6.5.4 TypeDescriptor

This subsection contains the interfaces and classes that are dependent on the user. This section will use the example in section API Usage Examples to define them.

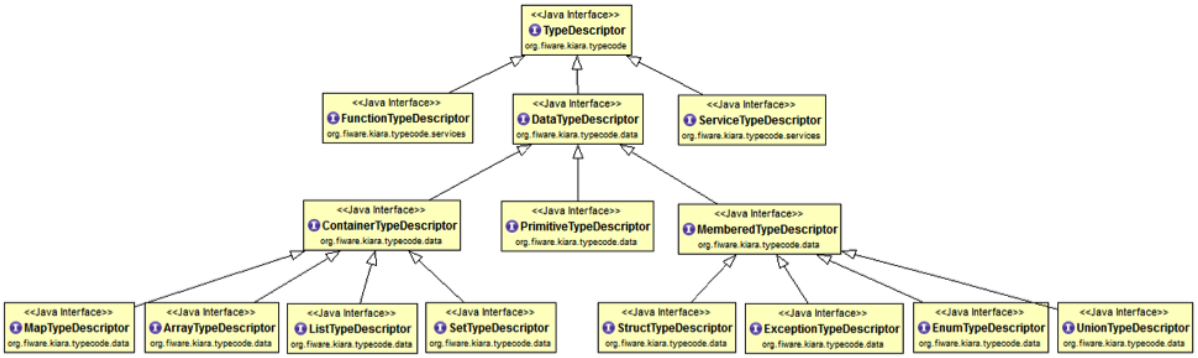

Fig. 6.1: Class Diagram TypeDescriptor

org.fiware.kiara.typecode.TypeDescriptorBuilder

This interface defined the operations used to create type-describing objects. It allows the users to create every supported data type inside Advanced Middleware by acting as a single access builder.

Functions:

- **createVoidType:** This function creates a new `DataTypeDescriptor` representing a void data type..
- **createPrimitiveType:** This function returns a new `PrimitiveTypeDescriptor` whose kind is the same specified as a parameter.
- **createArrayType**: Function that creates a new `ArrayTypeDescriptor` object representing an array.
- **createListType**: This function creates a new `ListTypeDescriptor` object representing a list of objects.
- **createSetType**: Function that creates a new `SetTypeDescriptor` object representing a set. A set is defined as a list with no repeated objects.
- **createMapType**: This function is used to create a `MapTypeDescriptor` object that represents a map data type.
- **createStructType**: This function creates a new `StructTypeDescriptor` object representing a struct data type.
- **createEnumType**: Function that creates a new `EnumTypeDescriptor` object representing an enumeration.
- **createUnionType**: This function can be used to create a new `UnionTypeDescriptor` that represents a union data type.
- **createExceptionType**: Function that creates a new `ExceptionTypeDescriptor` used to represent an exception data type.
- **createFunctionType**: This function can be used to create a new `FunctionTypeDescriptor` representing a Remote Procedure Call (RPC).
- **createServiceType**: Function that creates a new `ServiceTypeDescriptor` object used to represent a service defined in the server's side.

---

org.fiware.kiara.typecode.TypeDescriptor

This class is used to manipulate the objects used to describe the data types. It allows the users to know what type of data an object represents.

Functions:

- **getKind:** Function that returns the `TypeKind` of a `TypeDescriptor` object.

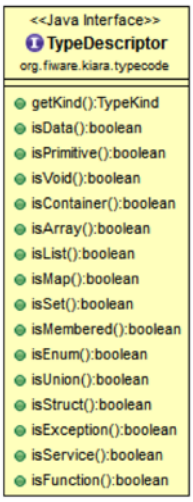

Fig. 6.2: Interface TypeDescriptor

- **isData:** This function returns true if and only if the `TypeDescriptor` represented by the object in which is invoked describes a data type. Functions and services are not considered data types.
- **isPrimitive:** Function used to know if a `TypeCode` object is a description of a primitive data type.
- **isVoid:** This function returns true if the `TypeDescriptor` object represents a void data type.
- **isContainer:** This function can be used to check if a `TypeDescriptor` object is representing a container type. The types considered as container data types are arrays, lists, sets and maps.
- **isArray:** Function used to know if a `TypeDescriptor` object is a description of an array data type.
- **isList:** Function used to know if a `TypeDescriptor` object is a description of a list data type.
- **isMap:** Function used to know if a `TypeDescriptor` object is a description of a map data type.
- **isSet:** Function used to know if a `TypeDescriptor` object is a description of a set data type.
- **isMembered:** This function is used to know if a `TypeDescriptor` object is a description of a membered data type. `Membered` types are structs, enumerations, unions and exceptions.
- **isStruct:** Function used to know if a `TypeDescriptor` object is a description of a struct data type.
- **isEnum:** Function used to know if a `TypeDescriptor` object is a description of an enumeration data type.
- **isUnion:** Function used to know if a `TypeDescriptor` object is a description of a union data type.
- **isException:** Function used to know if a `TypeDescriptor` object is a description of an exception data type.
- **isFunction:** Function used to know if a `TypeDescriptor` object is a description of a function.
- **isService:** Function used to know if a `TypeDescriptor` object is a description of a service.

---

org.fiware.kiara.typecode.data.DataTypeDescriptor

Interface that represents the top level class of the data type hierarchy. It is used as a generic type to englobe only and exclusively data type descriptors.

Functions: None

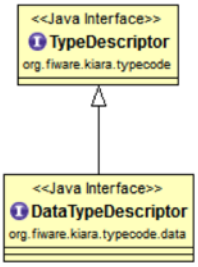

Fig. 6.3: Interface DataTypeDescriptor

org.fiware.kiara.typecode.data.PrimitiveTypeDescriptor

Interface that represents a primitive data type. Primitive types include **boolean**, **byte**, **i16**, **ui16**, **i32**, **ui32**, **i64**, **ui64**, **float32**, **float64**, **char** and **string**.

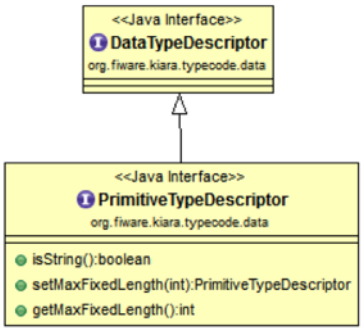

Fig. 6.4: Interface PrimitiveTypeDescriptor

Functions:

- **isString**: This function returns true if and only if the `PrimitiveTypeDescriptor` object represents a string data type.
- **setMaxFixedLength**: This function can only be used with string types. It sets the maximum length value for a specific string represented by the `PrimitiveTypeDescriptor` object.
- **getMaxFixedLength**: This function returns the maximum length specified when creating the `PrimitiveTypeDescriptor` object if it represents a string data type.

---

org.fiware.kiara.typecode.data.ContainerTypeDescriptor

Interface that represents a container data type. Container data types are **arrays**, **lists**, **maps** and **sets**.

Functions:

- **setMaxSize**: This function sets the maximum size of a container data type.
- **getMaxSize**: This function returns the maximum size of a container data type.

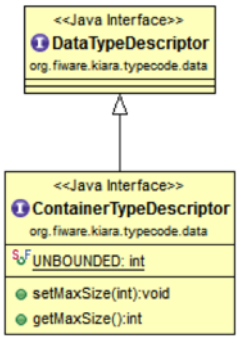

Fig. 6.5: Interface ContainerTypeDescriptor

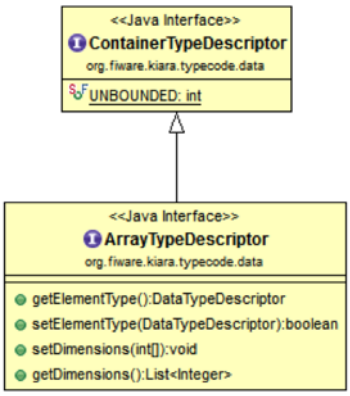

Fig. 6.6: Interface ArrayTypeDescriptor

org.fiware.kiara.typecode.data.ArrayTypeDescriptor

Interface that represents an array data type. Arrays can hold multiple repeated objects of the same data type inside.

Functions:

- **getElementType**: This function returns the `DataTypeDescriptor` object describing the content type of the array.
- **setElementType**: This function sets the `DataTypeDescriptor` object describing the content type of the array.
- **setDimensions**: This method sets the dimensions of the array.
- **getDimensions**: This method returns the different dimensions of the array.

---

org.fiware.kiara.typecode.data.ListTypeDescriptor

Interface that represents a list data type. Lists can hold multiple repeated objects of the same data type inside.

Functions:

- **getElementType**: This function returns the `DataTypeDescriptor` object describing the content type of the list.
- **setElementType**: This function sets the `DataTypeDescriptor` object describing the content type of the list.

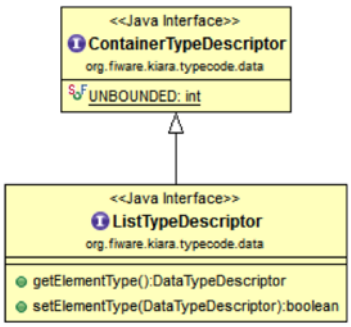

Fig. 6.7: Interface ListTypeDescriptor

org.fiware.kiara.typecode.data.SetTypeDescriptor

Interface that represents a set data type. Sets can have non repeated objects of the same data type inside.

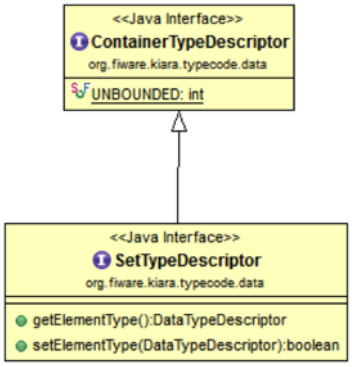

Fig. 6.8: Interface SetTypeDescriptor

Functions:

- **getElementType**: This function returns the `DataTypeDescriptor` object describing the content type of the set.
- **setElementType**: This function sets the `DataTypeDescriptor` object describing the content type of the set.

---

org.fiware.kiara.typecode.data.MapTypeDescriptor

Interface that represents a map data type. Maps can hold multiple key-object pairs inside if and only if the key objects are unique.

Functions:

- **getKeyTypeDescriptor**: This function returns the `DataTypeDescriptor` object describing the key type of the map.
- **setKeyTypeDescriptor**: This function sets the `DataTypeDescriptor` object describing the key type of the map.
- **getValueTypeDescriptor**: This function returns the `DataTypeDescriptor` object describing the value type of the map.
- **setValueTypeDescriptor**: This function sets the `DataTypeDescriptor` object describing the value type of the map.

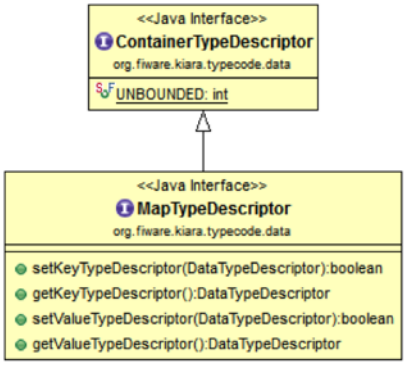

Fig. 6.9: Interface MapTypeDescriptor

---

org.fiware.kiara.typecode.data.MemberedTypeDescriptor

Interface that represents a membered data type. `Membered` data types are **structs**, **enumerations**, **unions** and **exceptions**.

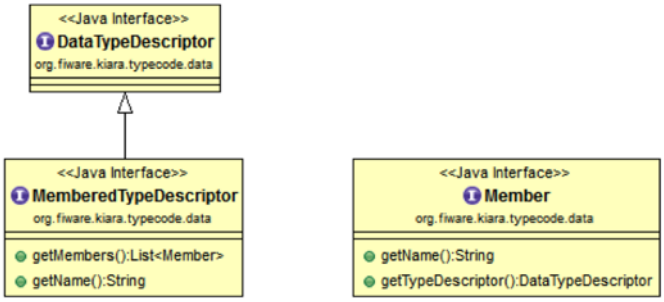

Fig. 6.10: Interface MemberedTypeDescriptor

Functions:

- **getMembers**: This function returns the list of member objects stored in a `ContainerTypeDescriptor` object.
- **getName**: This function returns the name of the `ContainerTypeDescriptor` object.

---

org.fiware.kiara.typecode.data.StructTypeDescriptor

Interface that represents a struct data type. Structs can have multiple different `DataTypeDescriptor` objects inside stored as members. Every struct member is identified by a unique name.

Functions:

- **addMember**: This function adds a new `TypeDescriptor` object as a member using a specific name.
- **getMember**: This function returns a `DataTypeDescriptor` object identified by the name introduced as a parameter.

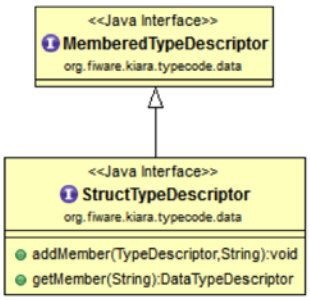

Fig. 6.11: Interface StructTypeDescriptor

org.fiware.kiara.typecode.data.EnumTypeDescriptor

Interface that represents an enumeration data type. Enumerations are formed by a group of different string values.

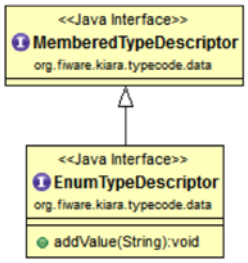

Fig. 6.12: Interface EnumTypeDescriptor

Functions:

- **addValue**: This function adds a new value to the enumeration using the string object received as a parameter.

---

org.fiware.kiara.typecode.data.UnionTypeDescriptor

Interface that represents a union data type. Unions are formed by a group of members identified by their names and the labels of the discriminator to which they are assigned.

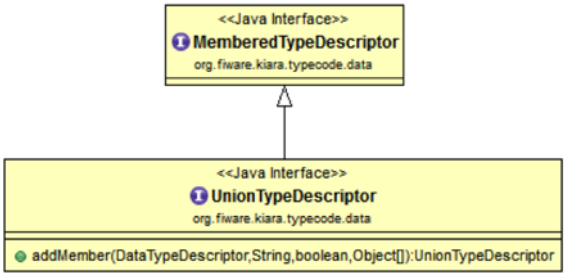

Fig. 6.13: Interface UnionTypeDescriptor

Functions:

- **addMember**: This function adds a new `TypeDescriptor` object as a member using a specific name and the labels of the discriminator.

---

org.fiware.kiara.typecode.data.ExceptionTypeDescriptor

Interface that represents a struct data type. Exceptions can have multiple different `DataTypeDescriptor` objects inside stored as members. Every struct member is identified by a unique name.

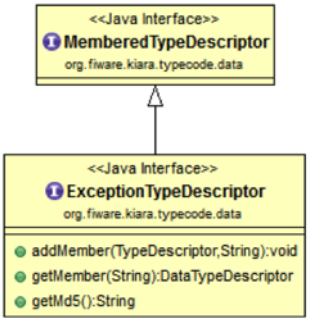

Fig. 6.14: Interface ExceptionTypeDescriptor

Functions:

- **addMember**: This function adds a new `TypeDescriptor` object as a member using a specific name.
- **getMember**: This function returns a `DataTypeDescriptor` object identified by the name introduced as a parameter.
- **getMd5**: This function returns the Md5 hash string of the exception name.

---

org.fiware.kiara.typecode.data.Member

Interface that represents a member of a `MemberedTypeDescriptor` object. Each member is identified by its name and the `TypeDescriptor` object that it holds.

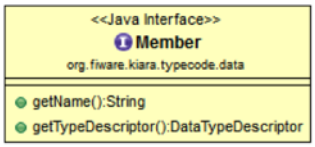

Fig. 6.15: Interface Member

Functions:

- **getName**: This function returns the member's name.
- **getTypeDescriptor**: This function returns a `DataTypeDescriptor` object stored inside the member.

---

org.fiware.kiara.typecode.data.EnumMember

Interface that represents a member of a `EnumTypeDescriptor` object. It inherits from `Member` interface and therefore it has no new methods.

Functions: None

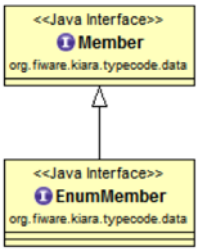

Fig. 6.16: Interface EnumMember

org.fiware.kiara.typecode.data.UnionMember

Interface that represents a member of a `UnionTypeDescriptor` object. It inherits from Member interface and therefore it has no new methods.

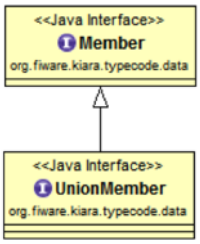

Fig. 6.17: Interface UnionMember

Functions: None

---

org.fiware.kiara.typecode.services.FunctionTypeDescriptor

This interface represents a function, providing methods to easily describe it by setting its return type, parameters and exceptions that it might throw.

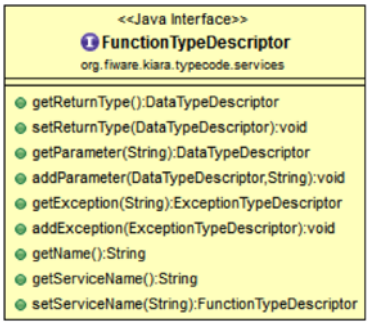

Fig. 6.18: Interface FunctionTypeDescriptor

Functions:

- **getReturnType**:This function returns the return `DataTypeDescriptor` of the function.
- **setReturnType**: This function sets the return `DataTypeDescriptor` of the function.
- **getParameter**: This function returns a `DataTypeDescriptor` representing a parameter whose name is the same as the one indicated.

- **addParameter**: This function adds a new `DataTypeDescriptor` to the parameters list with the name indicated.
- **getException**: This function returns an `ExceptionTypeDescriptor` whose name is the same as the one specified as a parameter.
- **addException**: This function adds a new `ExceptionTypeDescriptor` to the exceptions list.
- **getName**: This function returns the function name.
- **getServiceName**: This function returns the name of the `ServiceTypeDescriptor` in which the `FunctionTypeDescriptor` is defined.
- **setServiceName**: This function sets the name of the `ServiceTypeDescriptor` in which the `FunctionTypeDescriptor` is defined.

---

org.fiware.kiara.typecode.services.ServiceTypeDescriptor

This interface represents a service, providing methods to add the FunctionTypeDescriptor objects representing every function defined in a specific service.

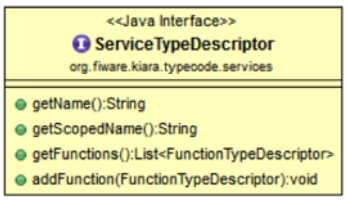

Fig. 6.19: Interface ServiceTypeDescriptor

Functions:

- **getName**: This function returns the service name.
- **getScopedName**: This function returns the service scoped name.
- **getFunctions**: This function returns the list of `FunctionTypeDescriptor` objects stored inside the `ServiceTypeDescriptor`.
- **addFunction**: This function adds a `FunctionTypeDescriptor` to the list of functions defined inside the service.

6.5.5 Dynamic

This subsection contains the interfaces and classes that are designed to provide the developer with functions to create and manage dynamic data types.

---

org.fiware.kiara.dynamic.DynamicValueBuilder

This class allows the users to create new data types based on their `TypeCode` descriptions.

Functions:

- **createData:** This function allows the user to create new `DynamicData` objects by using their `TypeDescriptor`.

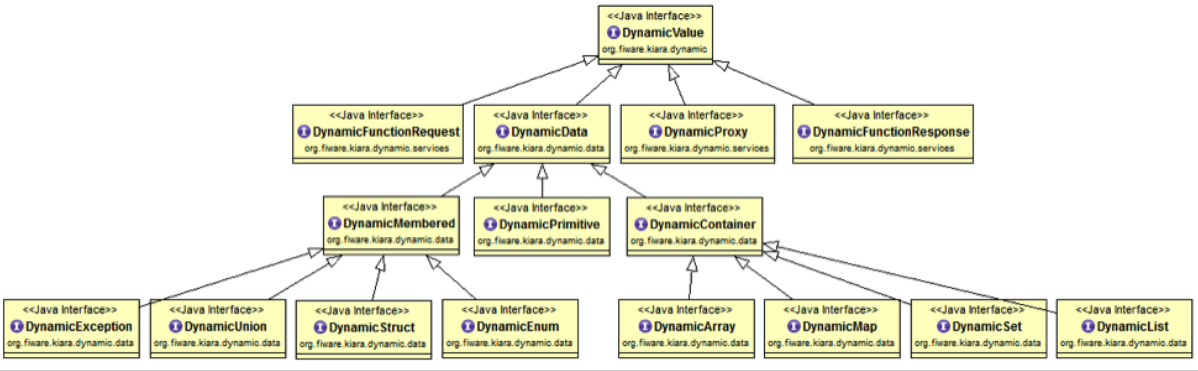

Fig. 6.20: Class Diagramm DynamicValue

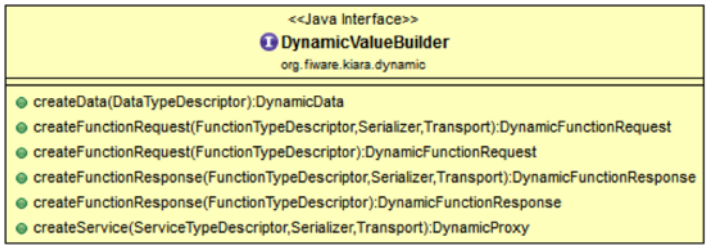

Fig. 6.21: Interface DynamicValueBuilder

- **createFunctionRequest:** This function receives a `FunctionTypeDescriptor` object describing a function, and it generates a new `DynamicFunctionRequest` (which inherits from `DynamicData`) object representing it.
- **createFunctionResponse:** This function receives a `FunctionTypeDescriptor` object describing a function, and it generates a new `DynamicFunctionResponse` (which inherits from `DynamicData`) object representing it.
- **createService:** This function receives a `ServiceTypeDescriptor` object describing a function, and it creates a new `DynamicService` object representing it.

---

org.fiware.kiara.dynamic.DynamicValue

Interface that acts as a supertype for every dynamic value that can be managed. Every `DynamicValue` object is defined by using a `TypeDescriptor` which is used to describe the data. It defines the common serialization functions as well as a function to retrieve the `TypeDescriptor` object it was created from.

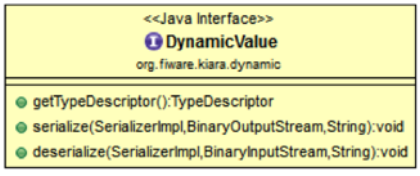

Fig. 6.22: Interface DynamicValue

Functions:

- **getTypeDescriptor**: This function returns the TypeDescriptor used when creating the DynamicValue object.
- **serialize:** This function serializes the content of the DynamicValue object inside a BinaryOutputStream message.

- **deserialize:** This function deserializes the content of a BinaryInputStream message into a DynamicValue object.

---

org.fiware.kiara.dynamic.data.DynamicData

Interface that is used to group all the `DynamicValues` representing data types.

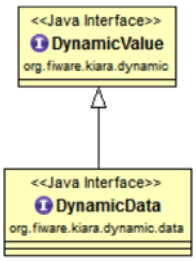

Fig. 6.23: Interface DynamicData

Functions: None

---

org.fiware.kiara.dynamic.data.DynamicPrimitive

This class allows the users to manipulate `DynamicData` objects made from `PrimitiveTypeDescriptor` objects.

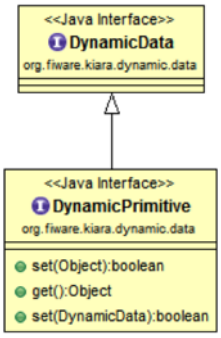

Fig. 6.24: Interface DynamicPrimitive

Functions:

- **set:** This function sets the inner value of a `DynamicPrimitive` object according to the `TypeDescriptor` specified when creating it.
- **get**: This function returns the value of a `DynamicPrimitive` object.

---

org.fiware.kiara.dynamic.data.DynamicContainer

This class holds the data values of a `DynamicData` object created from a `ContainerTypeDescriptor`.

Functions: None

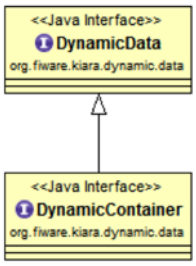

Fig. 6.25: Interface DynamicContainer

org.fiware.kiara.dynamic.data.DynamicArray

This class holds the data values of a `DynamicData` object created from an `ArrayTypeDescriptor`. A `DynamicArray` contains a group of `DynamicData` objects (all must be the same type) stored in single or multi dimensional matrixes.

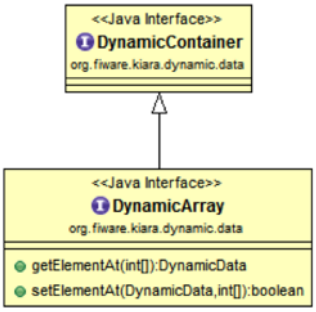

Fig. 6.26: Interface DynamicArray

Functions:

- **getElementAt:** This function returns `DynamicData` object stored in a certain position or coordinate..
- **setElementAt**: This function sets a `DynamicData` object in a specific position inside the array. If the array has multiple dimensions, the object will be set in a specific coordinate.

---

org.fiware.kiara.dynamic.data.DynamicList

This class holds the data values of a DynamicData object created from a ListTypeDescriptor. A list can only have one dimension and it has a maximum length. All the DynamicData objects stored inside a DynamicList must have been created from the same TypeDescriptor definition.

Functions:

- **add:** This function adds a `DynamicData` object into the list in the last position or in the position specified via parameter.
- **get**: This function returns a `DynamicData` object stored is a specific position in the list.
- **isEmpty**: This function returns true if the `DynamicList` is empty.

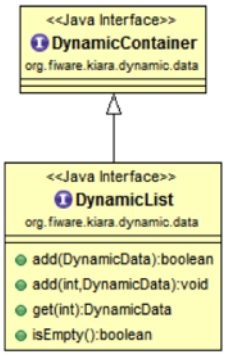

Fig. 6.27: Interface DynamicList

org.fiware.kiara.dynamic.data.DynamicSet

This class holds the data values of a DynamicData object created from a SetTypeDescriptor. A set can only have one dimension and it has a maximum length. All the DynamicData objects stored inside a DynamicSet must have been created from the same TypeDescriptor definition and it cannot be duplicated objects.

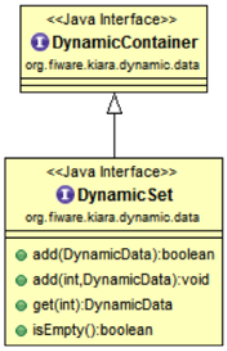

Fig. 6.28: Interface DynamicSet

Functions:

- **add:** This function adds a `DynamicData` object into the list in the last position or in the position specified via parameter.
- **get**: This function returns a `DynamicData` object stored is a specific position in the list.
- **isEmpty**: This function returns true if the `DynamicSet` is empty.

---

org.fiware.kiara.dynamic.data.DynamicMap

This class holds a list of pairs key-value instances of DynamicData. In a DynamicMap, the key values cannot be duplicated.

Functions:

- **put:** This function adds a new key-value pair using the DynamicData objets introduces as parameters. It will return false if the key value already exists in the map.
- **containsKey**: This function returns true if the DynamicMap contains at least one key-value pair in which the key DynamicData object is equal to the one introduced as a parameter.
- **containsValue**: This function returns true if the DynamicMap contains at least one key-value pair in which the value DynamicData object is equal to the one introduced as a parameter.

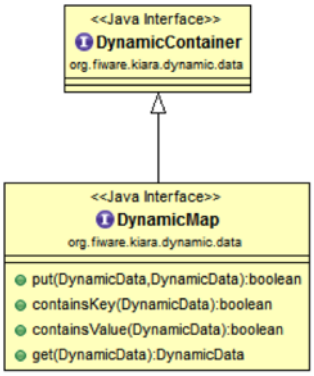

Fig. 6.29: Interface DynamicMap

- **get**: This function returns a DynamicData object from a key-value pair whose key is equal to the one introduced as a parameter.

---

org.fiware.kiara.dynamic.data.DynamicMembered

This class represents a `DynamicData` type formed by multiple `DynamicData` objects stored into a class named `DynamicMember`.

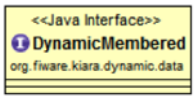

Fig. 6.30: Interface DynamicMembered

Functions: None

---

org.fiware.kiara.dynamic.data.DynamicStruct

This class holds group of `DynamicData` objects acting as members of a stucture. Each member is identified by its name.

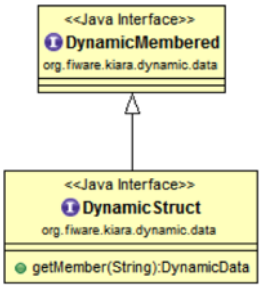

Fig. 6.31: Interface DynamicStruct

Functions:

- **getMember:** This function returns a `DynamicData` object (acting as a member of the structure) whose name is the same as the one introduced as a parameter.

---

org.fiware.kiara.dynamic.data.DynamicEnum

This class is used to dynamically manipulate enumerations described by a specific `EnumTypeDescriptor` object.

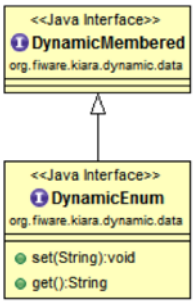

Fig. 6.32: Interface DynamicEnum

Functions:

- **set:** This function sets the actual value of the DynamicEnum object to the one specified as a parameter.
- **get**: This function returns the actual value of the DynamicEnum object.

---

org.fiware.kiara.dynamic.data.DynamicUnion

This class is used to dynamically manipulate unions described by a specific `UnionTypeDescriptor` object. A union is formed by some `DynamicData` objects, and the valid one is selected by using a discriminator.

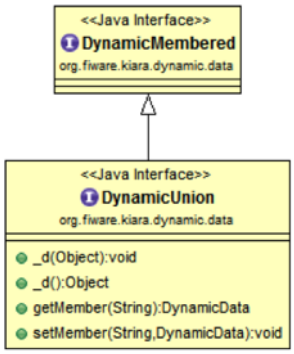

Fig. 6.33: Interface DynamicUnion

Functions:

- **_d:** This function either returns the discriminator or sets a new one, depending on the existence of an object parameter indicating a new value.
- **getMember**: This function returns valid `DynamicData` value depending on the selected discriminator.
- **setMember**: This function sets the `DynamicData` object received as a parameter in the member whose name is the same as the one introduced (if and only if the discriminator value is correct).

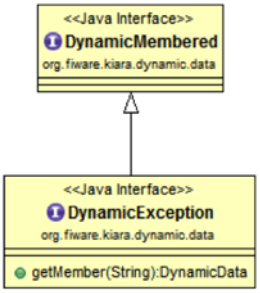

Fig. 6.34: Interface DynamicException

org.fiware.kiara.dynamic.data.DynamicException

This class holds group of `DynamicData` objects acting as members of an exception. Each member is identified by its own name.

Functions:

- **getMember:** This function returns a `DynamicData` object whose name is the same as the one introduced as a parameter.

---

org.fiware.kiara.dynamic.data.DynamicMember

This class represents a dynamic member of any DynamicMembered object. It is used to store the DynamicData objects inside structures, unions, enumerations and exceptions.

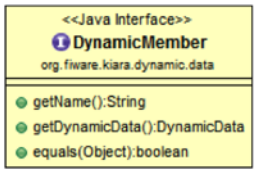

Fig. 6.35: Interface DynamicMember

Functions:

- **getName:** This function returns the member's name.
- **getDynamicData**: This function returns the `DynamicData` stored inside a `DynamicMember` object.
- **equals**: It returns true if two `DynamicMember` objects are equal.

---

org.fiware.kiara.dynamic.service.DynamicFunctionRequest

This class represents a dynamic function request. This class is used to create objects whose objective is to invoke functions remotely.

Functions:

- **getParameter:** This function returns a `DynamicData` object stored in the parameter list depending on its name or its position in such list.
- **execute**: This function executes a function remotely. It serializes all the necessary information and sends the request over the wire. It returns a `DynamicFunctionResponse` with the result.

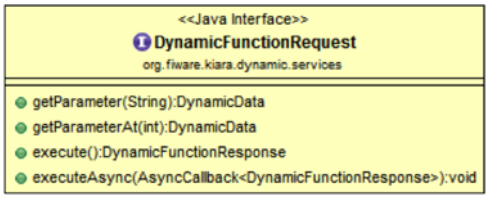

Fig. 6.36: Interface DynamicFunctionRequest

- **executeAsync**: This function behaves the same way as the function `execute`. The only difference is that it needs a callback to be executed when the response arrives from the server.

---

org.fiware.kiara.dynamic.service.DynamicFunctionResponse

This class represents a dynamic function response. This class is used to retrieve the information sent from the server after a remote procedure call.

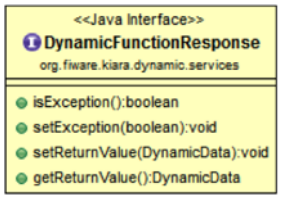

Fig. 6.37: Interface DynamicFunctionResponse

Functions:

- **isException:** This function returns true if the server raised an exception when executing the function.
- **setException**: This method sets the attribute indicating that an exception has been thrown on the server side.
- **setReturnValue**: This function sets a `DynamicData` object as a return value for the remote call.
- **getReturnValue**: This function returns the `DynamicData` representing the result of the remote call.

---

org.fiware.kiara.dynamic.service.DynamicProxy

This class represents a proxy than can be dynamically used to create an instance of DynamicFunctionRequest or a DynamicFunctionResponse depending if the user wants an object to execute a remote call or to store the result.

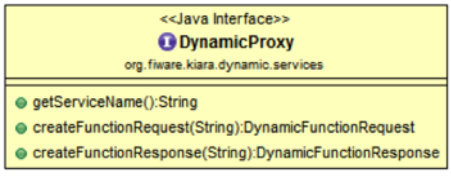

Fig. 6.38: Interface DynamicProxy

Functions:

- **getServiceName:** This function returns the service name.

- **createFunctionRequest**: This function creates a new object instance of `DynamicFunctionRequest` according to the `FunctionTypeDescriptor` that was used to describe it.
- **createFunctionResponse**: This function creates a new object instance of `DynamicFunctionResponse` according to the `FunctionTypeDescriptor` that was used to describe it.

---

org.fiware.kiara.dynamic.service.DynamicFunctionHandler

This class represents a dynamic object used to hold the implementation of a specific function. Its process method must be defined by the user when creating the object, and it will be used to register the service's functions on the server's side.

Fig. 6.39: Interface DynamicFunctionHandler

Functions:

- **process:** This function is the one that will be registered to be executed when a client invokes remotely a function. It must be implemented by the user.

6.6 Detailed API

This section defines in detail the API provided by the classes defined above.

6.6.1 Main entry point

org.fiware.kiara.Kiara			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getTypeDescriptorBuilder		TypeDescriptorBuilder	
getDynamicValueBuilder		DynamicValueBuilder	
createContext		Context	
shutdown		void	

6.6.2 Client API

This classes are those related to the client side API. This section includes all the relevant classes, attributes and methods.

org.fiware.kiara.client.Connection			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getTransport		Transport	
getSerializer		Serializer	
getServiceProxy		T	Exception
	interfaceClass	Class<T>	
getDynamicProxy		DynamicProxy	
	name	String	

6.6.3 TypeDescriptor

This classes are those related to the client's side API. This section includes all the relevant classes, attributes and methods.

org.fiware.kiara.typecode.TypeDescriptorBuilder			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
createVoidType		DataTypeDescriptor	
createPrimitiveType		PrimitiveTypeDescriptor	
	kind	TypeKind	
createArrayType		ArrayTypeDescriptor	
	contentDescriptor	DataTypeDescriptor	
	dimensions	int[]	
createListType		ListTypeDescriptor	
	contentDescriptor	DataTypeDescriptor	
	maxSize	int	
createSetType		SetTypeDescriptor	
	contentDescriptor	DataTypeDescriptor	
	maxSize	int	
createMapType		MapTypeDescriptor	
	keyDescriptor	DataTypeDescriptor	
	valueDescriptor	DataTypeDescriptor	
	maxSize	int	
createStructType		StructTypeDescriptor	
	name	String	
createEnumType		EnumTypeDescriptor	
	name	String	
	values	String[]	
createUnionType		UnionTypeDescriptor	
	name	String	
	discriminatorDesc	DataTypeDescriptor	
createExceptionType		ExceptionTypeDescriptor	
	name	String	
createFunctionType		FunctionTypeDescriptor	
	name	String	
createServiceType		ServiceTypeDescriptor	
	name	String	
	scopedName	String	

org.fiware.kiara.typecode.TypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getKind		TypeKind	
isData		boolean	
isPrimitive		boolean	
isVoid		boolean	
isContainer		boolean	
isArray		boolean	
isList		boolean	
isMap		boolean	
isSet		boolean	
isMembered		boolean	
isEnum		boolean	
isUnion		boolean	
isStruct		boolean	
isException		boolean	
isService		boolean	
isFunction		boolean	

org.fiware.kiara.typecode.data.DataTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
n/a			

org.fiware.kiara.typecode.data.PrimitiveTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
isString		boolean	
setMaxFixedLength		PrimitiveTypeDescriptor	
	length	int	
getMaxFixedLength		int	

org.fiware.kiara.typecode.data.ContainerTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
setMaxSize		void	
	length	int	
getMaxSize		int	

org.fiware.kiara.typecode.data.ArrayTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getElementType		DataTypeDescriptor	
setElementType		boolean	
	contentType	DataTypeDescriptor	
setDimensions		void	
	dimensions	int[]	
getDimensions		List<Integer>	

org.fiware.kiara.typecode.data.ListTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getElementType		DataTypeDescriptor	
setElementType		boolean	
	contentType	DataTypeDescriptor	

org.fiware.kiara.typecode.data.SetTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getElementType		DataTypeDescriptor	
setElementType		boolean	
	contentType	DataTypeDescriptor	

org.fiware.kiara.typecode.data.MapTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
setKeyTypeDescriptor		boolean	
	keyTypeDescriptor	DataTypeDescriptor	
getKeyTypeDescriptor		DataTypeDescriptor	
setValueTypeDescriptor		boolean	
	valueTypeDescriptor	DataTypeDescriptor	
getValueTypeDescriptor		DataTypeDescriptor	

org.fiware.kiara.typecode.data.MemberedTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getMembers		List<Member>	
getName		String	

org.fiware.kiara.typecode.data.StructTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
addMember		void	
	member	TypeDescriptor	
	name	String	
getMember		DataTypeDescriptor	
	name	String	

org.fiware.kiara.typecode.data.EnumTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
addValue		void	
	value	String	

org.fiware.kiara.typecode.data.UnionTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
addMember		UnionTypeDescriptor	
	typeDescriptor	DataTypeDescriptor	
	name	String	
	isDefault	boolean	
	labels	Object[]	

org.fiware.kiara.typecode.data.FunctionTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getReturnType		DataTypeDescriptor	
setReturnType		void	
	returnType	DataTypeDescriptor	
getParameter		DataTypeDescriptor	
	name	String	
addParameter		void	
	parameter	DataTypeDescriptor	
	name	String	
getException		ExceptionType-Descriptor	
	name	String	
addException		void	
	exception	ExceptionType-Descriptor	
getName		String	
getServiceName		String	
setServiceName		FunctionTypeDescrip-tor	
	service-Name	String	

org.fiware.kiara.typecode.data.ServiceTypeDescriptor			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getName		String	
getScopedName		String	
getFunctions		List<FunctionTypeDescriptor>	
addFunction		void	
	functionType-Desc	FunctionTypeDescriptor	

6.6.4 Dynamic

The following classes are those related to creation and management of dynamic types, including data definition and function description and execution.

org.fiware.kiara.dynamic.DynamicValueBuilder			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
createData		DynamicData	
	dataDescriptor	DataTypeDescriptor	
createFunctionRequest		DynamicFunctionRequest	
	functionDescriptor	FunctionTypeDescriptor	
	serializer	Serializer	
	transport	Transport	
createFunctionRequest		DynamicFunctionRequest	
	functionDescriptor	FunctionTypeDescriptor	
createFunctionResponse		DynamicFunctionResponse	
	functionDescriptor	FunctionTypeDescriptor	
	serializer	Serializer	
	transport	Transport	
createFunctionResponse		DynamicFunctionResponse	
	functionDescriptor	FunctionTypeDescriptor	
createService		DynamicProxy	
	serviceDescriptor	ServiceTypeDescriptor	
	serializer	Serializer	
	transport	Transport	

org.fiware.kiara.dynamic.DynamicValue			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getTypeDescriptor		TypeDescriptor	
serialize		void	IOException
	impl	SerializerImpl	
	message	BinaryOutputStream	
	name	String	
deserialize		void	IOException
	impl	SerializerImpl	
	message	BinaryInputStream	
	name	String	

org.fiware.kiara.dynamic.data.DynamicData			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
n/a			

org.fiware.kiara.dynamic.data.DynamicPrimitive			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
set		boolean	
	value	Object	
get		Object	
set		boolean	
	value	DynamicData	

org.fiware.kiara.dynamic.data.DynamicContainer			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
n/a			

org.fiware.kiara.dynamic.data.DynamicArray			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getElement		DynamicData	
	position	int[]	
setElementAt		boolean	
	value	DynamicData	
	position	int[]	

org.fiware.kiara.dynamic.data.DynamicList			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
add		boolean	
	element	DynamicData	
add		void	
	index	int	
	element	DynamicData	
get		DynamicData	
	index	int	
isEmpty		boolean	

org.fiware.kiara.dynamic.data.DynamicSet			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
add		boolean	
	element	DynamicData	
add		void	
	index	int	
	element	DynamicData	
get		DynamicData	
	index	int	
isEmpty		boolean	

org.fiware.kiara.dynamic.data.DynamicMap			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
put		boolean	
	key	DynamicData	
	value	DynamicData	
containsKey		boolean	
	key	DynamicData	
containsValue		boolean	
	value	DynamicData	
get		DynamicData	
	key	DynamicData	

org.fiware.kiara.dynamic.data.DynamicMembered			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
n/a			

org.fiware.kiara.dynamic.data.DynamicStruct			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getMember		DynamicData	
	name	String	

org.fiware.kiara.dynamic.data.DynamicEnum			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
set		void	
	value	String	
get		String	

org.fiware.kiara.dynamic.data.Dynamic			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
_d		void	
	value	Object	
_d		Object	
getMember		DynamicData	
	name	String	
setMember		void	
	name	String	
	data	DynamicData	

org.fiware.kiara.dynamic.data.DynamicException			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getMember		DynamicData	
	name	String	

org.fiware.kiara.dynamic.data.DynamicMember			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getName		String	
getDynamicData		DynamicData	
equals		boolean	
	anotherObject	Object	

org.fiware.kiara.dynamic.service.DynamicFunctionRequest			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getParameter		DynamicData	
	name	String	
getParameterAt		DynamicData	
	index	int	
execute		DynamicFunctionResponse	
executeAsync		void	
	callback	AsyncCallback<DynamicFunctionResponse>	

org.fiware.kiara.dynamic.service.DynamicFunctionResponse			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
isException		boolean	
setException		void	
	isException	boolean	
setReturnValue		void	
	returnType	DynamicData	
getReturnValue		DynamicData	

org.fiware.kiara.dynamic.service.DynamicProxy			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
getServiceName		String	
createFunctionRequest		DynamicFunctionRequest	
	name	String	
createFunctionReqponse		DynamicFunctionResponse	
	name	String	

org.fiware.kiara.dynamic.service.DynamicFunctionRequest			
Attributes			
Name	*Type*		
n/a	n/a		
Public Operations			
Name	*Parameters*	*Returns/Type*	*Raises*
process		void	
	request	DynamicFunctionRequest	
	response	DynamicFunctionResponse	

CHAPTER 7

Advanced Middleware Publication Subscription API Specification

Date: 01th November 2015

- Version: *0.3.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

7.1 Abstract

This document presents a proposal for a publication-subscription API. The goal of this document is to define the basic principles of a new KIARA Publication-Subscription framework.

The API proposed in this document will be very user-friendly in order to allow FI-WARE users to easily migrate to KIARA's Publisher-Subscriber framework. The presented API should allow users to set up a publisher or a subscriber with very few calls and with little knowledge of the technology behind the framework.

7.2 Status of this Document

Date	Description
29-April-2015	Release 0.2.0
01-November-2015	Release 0.3.0

7.3 Conformance

The conformance clause identifies which clauses of the specification are mandatory (or conditionally mandatory) and which are optional in order for an implementation to claim conformance to the specification.

Note: For conditionally mandatory clauses, the conditions must be specified.

7.4 Reference Material

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

7.5 API Overview

This section enumerates and describes the classes provided by the KIARA Publication-Subscription API.

The Publication-Subscription framework is a messaging pattern by which the information producers, in this case called publishers, publish messages characterized in a specific manner. On the other hand, the information consumers, called subscribers, state their interest in a certain type of messages. This pattern provides a larger flexibility and scalability compared to other messaging paradigms.

This pattern also allows for message filtering, and the most common way of doing so is by using a topic-based filter, where each publisher or subscriber produces or consumes information of a specific topic. Each topic has a unique name in the network, as well as a specific data type associated with it.

The main objective of this API is to allow users to easily have access to the basics of a Publication-Subscription messaging framework, providing them with a friendly interface and lowering the learning curve.

7.5.1 Main classes

There are four main classes in this API:

org.fiware.kiara.ps.participant.Participant

Singleton class representing a node in the network. It might have multiple publishers and subscribers associated to it. This object acts as a supervisor of this node in the network. All the methods in this class will be static, since the constructor will be hidden to the user.

Functions:

- **registerTopicDataType:** This function is used to register new TopicDataType objects in a Participant (network node). The registration acts as a contract defining which data types a publisher is able to understand. It allows the creation of publishers and subscribers that use the registered TopicDataType.
- **createPublisher:** Function to create a new Publisher for a specific TopicDataType object and associate it with a listener.
- **createSubscriber:** Function to create a new Subscriber for a specific TopicDataType object and associate it with a listener.
- **removePubisher:** This function removes a Publisher from this Participant.
- **removeSubscriber:** This function removes a Subscriber from this Participant.
- **registerType:** This function is used to register a TopicDataType in this participant.

org.fiware.kiara.ps.topic.TopicDataType

This interface describes the serialization and deserialization procedures of the selected data type. It also defines how the key is obtained from the data object in case the topic is a keyed topic.

Functions:

- **serialize:** This function defines the signature of the serialization functions for every TopicDataType object existing in the framework.

- **deserialize:** This function defines the signature of the deserialization functions for every TopicDataType object existing in the framework.
- **createData:** This function returns a new object of the type represented by the TopicDataType.
- **getKey:** This function is used to return the IntanceHandle object representing the 16-Byte key of the TopicDataType.

org.fiware.kiara.ps.publisher.Publisher

This class is the one used to describe a data publisher inside a specific node. It has multiple parameters grouped into a single PublisherAttributes object, which will be detailed later.

It also provides functions to easily change and manipulate the parameters of the publisher, as well as a function to send data over the wire.

Functions:

- **getAttributes:** This function is used to retrieve the PublisherAttributes object contained within this class.
- **setAttributes:** This function is used to set the PublisherAttributes object inside this class.
- **write:** This function is the one used to send data through the network. Its function is to publish information about a specific topic which the publisher is able to use.
- **destroy:** This function is used to delete all the entities associated to this publisher.

org.fiware.kiara.ps.subscriber.Subscriber

This class is similar to the Publisher class. All the parameters associated with each subscriber are aggregated into a single SubscriberAttributes object that can be retrieved and changed by using its accessor functions.

Functions:

- **getAttributes:** This function is used to retrieve the SubscriberAttributes object contained within this class.
- **setAttributes:** This function is used to set the SubscriberAttributes object inside this class.
- **readNextData:** This function is used to retrieve an unread CacheChange with the data sent through the network. The data received belongs to a topic the subscriber has subscribed to.
- **takeNextData:** This function is used to retrieve and remove the next unread CacheChange with data receiver over the wire.
- **waitForMessage:** This method blocks the execution thread until a message is received. This message can be retrieved then using the **read** function.
- **destroy:** This function is used to delete all the entities associated to this subscriber.

7.5.2 Secondary classes

This classes are those necessary for the main ones described before. In this group are included the classes that are used to deliberately specify a certain behaviour for an event that happened in the publisher or the subscriber side.

org.fiware.kiara.ps.listeners.PublisherListener

This interface is designed to be implemented for those classes that ought to manage certain events in the publisher side. It defines a set of methods that can be overwritten by the user to specify the behaviour of the publisher when certain events occur. An example of this would be a new subscriber that has been discovered.

Functions:

- **onPublicationMatched:** This function is the one that will be called when the data published by a publisher matches with the subscriber for a specific topic. The MatchingInfo class provided as a parameter gives the user information about the matched subscriber.

org.fiware.kiara.ps.listeners.SubscriberListener

This interface is similar to the PublisherListener interface described above, but in this case it defines a set of methods used to specify the subscriber's behaviour. An example of this would be a new message that has been received.

Functions:

- **onSubscriptionMatched:** This method's objective is the same as the one described in the PublisherListener class, but in this case, it will be executed when a new subscription matches with the data a publisher is publishing.
- **onNewDataMessage:** This function will be executed when a new message is received by the subscriber.

7.5.3 Auxiliary classes

org.fiware.kiara.ps.utils.InstanceHandle

This class contains the serialized data of a specific message the user is sending or receiving through the network. It provides functions to retrieve the information of such message.

Functions: None

org.fiware.kiara.ps.attributes.TopicAttributes

This structure contains all the different attributes of a Topic. These attributes will include the topic name, the data type name and the topic kind.

Attributes:

- **topicKind:** This attribute represents the kind of the Topic (with key or without key)
- **topicName:** This attribute represents the topic name.
- **topicDataTypeName:** This attribute represents the name of the data type for a specific topic.
- **historyQos:** This attribute represents the History QoS.
- **resourceLimitQos:** This attribute represents the limit of the resources for this topic.

org.fiware.kiara.ps.attributes.PublisherAttributes

This structure contains all the different attributes of a publisher. These attributes will include the topic attributes (topic name, topic data type, etc), as well as the list of locators, times and writer QoS.

Attributes:

- **topic:** Object instance of TopicAttributes. It holds all the attributes of the Topic.
- **wqos:** Represents the Qualities of Service associated to the Writer.
- **times:** Time values associated to the Publisher entity.
- **unicastLocatorList:** List of unicast locators representing different Endpoints.
- **multicastLocatorList:** List of multicast locators representing different Endpoints.

Functions:

- **getUserDefinedID:** Returns the user defined identifier for this object.

- **setUserDefinedID:** Sets the user defined identifier for this object.
- **getEntityId:** Returns the EntityID that uses this attributes class.
- **setEntityId:** Sets the EntityID that uses this attributes class.

org.fiware.kiara.ps.attributes.SubscriberAttributes

This structure contains all the different attributes of subscriber. These attributes will include the topic attributes (name, topic data type, etc), as well as the list of locators, times and reader QoS.

Attributes:

- **topic:** Object instance of TopicAttributes. It holds all the attributes of the Topic.
- **rqos:** Represents the Qualities of Service associated to the Reader.
- **times:** Time values associated to the Subscriber entity.
- **unicastLocatorList:** List of unicast locators representing different Endpoints.
- **multicastLocatorList:** List of multicast locators representing different Endpoints.
- **expectsInlineQos:** This attribute defines whether or not the Subscriber will expect inline QoS.

Functions:

- **getUserDefinedID:** Returns the user defined identifier for this object.
- **setUserDefinedID:** Sets the user defined identifier for this object.
- **getEntityId:** Returns the EntityID that uses this attributes class.
- **setEntityId:** Sets the EntityID that uses this attributes class.

org.fiware.kiara.ps.utils.SampleInfo

This class contains information about each particular message, for example the publisher who originated it or a timestamp of its creation time. The GUID of the writer is related to the node from where the information comes from.

Functions:

- **getWriterGUID:** This function is used to obtain the GUID of the writer who originated a specific message.
- **getTimestamp:** This function returns the timestamp value indicating the exact time when the message was created.

org.fiware.kiara.ps.utils.MatchingInfo

This class informs the user of whether the event is a matching or an un-matching event and also the global identifier of the remote endpoint.

Functions:

- **getMatchingStatus:** This function allows the users to know the matching status of a specific endpoint with the risen event.
- **getRemoteEndpointGUID:** This function returns the endpoint's unique identifier.

org.fiware.kiara.ps.common.GUID

This class represents a unique identifier of a node in the network. It is formed by two members, a prefix (12 Bytes) and an entity ID (4 Bytes).

Functions:

- **getEntityId:** This function returns the EntityId associated to the GUID.
- **getGUIDPrefix:** This function returns the GUIDPrefix associated to the GUID.

Note: There are some classes that do not appear yet in this document, and this is because their definition is too long (they will be included in an external annex)

A few examples of these classes are:

- QoS related classes: QoSList, ReaderQos, WriterQos and QoSPolicy (and all its subclasses).
- SerializableDataType: Interface between Serializable objects defined in KIARA and TopicDataTypes.
- History classes: HistoryCache, SubscriberHistory and PublisherHistory

---

7.6 API Description

This section details the classes of this API and all their methods.

7.6.1 Main classes

org.fiware.kiara.ps.participant.Participant

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
registerTopicDataType		boolean	
	dataType	TopicDataType<T>	
createPublisher		Publisher<T>	
	attributes	PublisherAttributes	
	listener	PublisherListener	
	topic	TopicDataType<T>	
createSubscriber		Subscriber<T>	
	attributes	SubscriberAttributes	
	listener	SubscriberListener	
	topic	TopicDataType<T>	
removePublisher		boolean	
	publisher	Publisher<T>	
removeSubscriber		boolean	
	subscriber	Subscriber<T>	
registerType		boolean	
	type	TopicDataType<T>	

org.fiware.kiara.ps.topic.TopicDataType

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
serialize		void	IOException
	payload	SerializedPayload	
	object	T	
deserialize		T	IOException
	payload	SerializerPayload	
createData		T	
getKey		InstanceHandle	IOException
	object	T	

org.fiware.kiara.ps.publisher.Publisher

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
getAttributes		Attributes	
setAttributes		void	PublisherException
	attributes	PublisherAttributes	
write		boolean	PublisherException
	data	TopicDataType<T>	
destroy		void	

org.fiware.kiara.ps.subscriber.Subscriber

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
getAttributes		Attributes	
setAttributes		void	SubscriberException
	attributes	SubscriberException	
waitForMessage		void	SubscriberException
readNextData		boolean	
	info	SampleInfo	
takeNextData		boolean	
	info	SampleInfo	
destroy		void	

7.6.2 Secondary classes

org.fiware.kiara.ps.publisher.PublisherListener

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
onPublicationMatched		void	PublisherException
	info	MatchingInfo	
	pub	Publisher	

org.fiware.kiara.ps.subscriber.SubscriberListener

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
onSubscriptionMatched		void	SubscriberException
	info	MatchingInfo	
	sub	Subscriber	
onNewDataMessage		void	SubscriberException
	sub	Subscriber	

7.6.3 Auxiliary classes

org.fiware.kiara.ps.utils.InstanceHandle

The public methods of this class are listed below:

Attributes

- None

Public Operations

- None

org.fiware.kiara.ps.attributes.TopicAttributes

The public methods of this class are listed below:

Attributes

+——————+——————+ | | **Name** | **Type** | | +=====================+=========================
| | topicKind | TopicKind | | +——————+——————+ | | topicName | String | | +——————+——————+ | | topicDataTypeName | String | | +——————+——————+ | | historyQos | HistoryPolicyQos | | +——————+——————+ | | resourceLimitQos | Resource-LimitsQosPolicy | | +——————+——————+ |

Public Operations

- None

org.fiware.kiara.ps.attributes.PublisherAttributes

The public methods of this class are listed below:

Attributes

+——————+——————-+ | | **Name** | **Type** | | +========================+====================+
| | topic | TopicAttributes | | +——————+——————-+ | | wqos | WriterQos | | +——————+——————-+ | | times | WriterTimes | | +——————+——————-+ | | unicastLocatorList | LocatorList | | +——————+——————-+ | | multicastLocatorList | LocatorList | | +——————+——————-+ |

Public Operations

Name	Parameters	Returns/Type	Raises
getUserDefinedID		short	
setUserDefinedID		void	
	userDefinedID	short	
getEntityID		short	
setEntityID		void	
	entityID	short	

org.fiware.kiara.ps.attributes.SubscriberAttributes

The public methods of this class are listed below:

Attributes

+——————+————-+ | | **Name** | **Type** | | +=======================+==================+ | | topic | TopicAttributes | | +——————+————-+ | | rqos | ReaderQos | | +——————+————-+ | | times | ReaderTimes | | +——————+————-+ | | unicastLocatorList | LocatorList | | +——————+————-+ | | multicastLocatorList | LocatorList | | +——————+————-+ | | expectsInlineQos | boolean | | +——————+————-+ |

Public Operations

Name	Parameters	Returns/Type	Raises
getUserDefinedID		short	
setUserDefinedID		void	
	userDefinedID	short	
getEntityID		short	
setEntityID		void	
	entityID	short	

org.fiware.kiara.ps.utils.SampleInfo

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
getWriterGUID		GUID	
getTimestamp		Timestamp	

org.fiware.kiara.ps.utils.MatchingInfo

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
getMatchingStatus		MatchingStatus	
geRemoteEndpointGUID		GUID	

org.fiware.kiara.ps.common.GUID

The public methods of this class are listed below:

Attributes

- None

Public Operations

Name	Parameters	Returns/Type	Raises
getEntityId		EntityId	
setEntityId		void	
	entityID	EntityId	
getGUIDPrefix		GuidPrefix	
setGUIDPrefix		void	
	guidPrefix	GUIDPrefix	
equals		boolean	
	other	Object	

CHAPTER 8

Advanced Middleware IDL Specification

Date: 10th October 2015

- Version: *0.3.0*
- Latest version: latest

Editors:

- eProsima - The Middleware Experts
- DFKI - German Research Center for Artificial Intelligence
- ZHAW - School of Engineering (ICCLab)

Copyright © 2013-2015 by eProsima, DFKI, ZHAW. All Rights Reserved

8.1 Abstract

Ahe Advanced Middleware GE enables flexible, efficient, scalable, and secure communication between distributed applications and to/between FIWARE GEs. The **Interface Definition Language (IDL)** Specifications allowes to describes the Data Types and Operations supported by a Service. Following is a list of the main features it supports:

- **IDL, Dynamic Types & Application Types**: It support the usual schema of IDL compilation to generate support code for the data types.
- **IDL Grammar**: An OMG-like grammar for the IDL as in DDS, Thrift, ZeroC ICE, CORBA, etc.
- **Types**: Support of simple set of basic types, structs, and various high level types such as lists, sets, and dictionaries (maps).
- **Type inheritance, Extensible Types, Versioning**: Advanced data types, extensions, and inheritance, and other advanced features will be supported.
- **Annotation Language**: The IDL is extended with an annotation language to add properties to the data types and operations. These will, for example, allows adding security policies and QoS requirements.

8.2 Status of this Document

Date	Description
8-April-2015	Release 0.2.0
10-October-2015	Release 0.3.0

8.3 Preface

The foundation of the FIWARE Middleware Interface Definition Language (IDL) is the Object Management Group (OMG) IDL 3.5. See *Appendix C* for the OMG IDL 3.5 grammar.

To maintain backward compatibility, the FIWARE Middleware IDL grammar embraces all OMG IDL 3.5 features. IDL parsers are not required to implement all of the extended OMG features. Check the documentation of the specific parser implementations.

The basic subset needed by DDS and future standard RPC over DDS must be supported.

8.4 Related documentation

- OMG DDS-XTypes 1.0

8.5 Syntax Definition

The FIWARE Middleware IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. Some definitions are allowed in the grammar for backward compatibility, but are not used by the FIWARE Middleware IDL and therefore will be ignored by the implmentations.

```
<specification> ::= <import>* <definition>+
<definition> ::= <type_dcl> ";"
              |   <const_dcl> ";"
              |   <except_dcl> ";"
              |   <interface> ";"
              |   <module> ";"
              |   <value> ";"
              |   <type_id_dcl> ";"
              |   <type_prefix_dcl> ";"
              |   <event> ";"
              |   <component> ";"
              |   <home_dcl> ";"
              |   <annotation_dcl> ";"
              |   <annotation_appl> <definition>
```

See section *Import Declaration* for the specification of `<import>`
See section *Module Declaration* for the specification of `<module\>`
See section *Interface Declaration* for the specification of `<interface>`
See section *Value Declaration* for the specification of `<value\>`
See section *Constant Declaration* for the specification of `<const_dcl>`
See section *Type Declaration* for the specification of `<type_dcl>`
See section *Exception Declaration* for the specification of `<except_dcl>`
See section *Repository Identity Related Declarations* for the specification of `<type_id_dcl>` and `<type_prefix_dcl>`
See section *Event Declaration* for the specification of `<event>`
See section *Component Declaration* for the specification of `<component>`
See section *Event Declaration* for the specification of `<home_dcl>`
See section *Annotation Declaration* for the specification of `<annotation_dcl>`
See section *Annotation Application* for the specification of `< annotation_appl>`

8.5.1 Import Declaration

An import statement conforms to the following syntax:

```
<import> ::= "import" <imported_scope> ";"
<imported_scope> ::= <scoped_name> | <string_lieral>
```

Import declarations are not supported by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and ignore the declaration.

8.5.2 Module Declaration

A module definition conforms to the following syntax:

```
<module> ::= ("module" | "namespace") <identifier> "{" <definition> + "}"
```

The module construct is used to scope IDL identifiers. FIWARE Middleware IDL supports the OMG IDL 3.5 keyword `module`, but also adds the modern keyword `namespace` as an alias.

Examples of module definitions:

```
namespace MyNamespace {
   ...
};

namespace YourNamespace {
   namespace HisNamespace {
      ...
   };
};
```

8.5.3 Interface Declaration

An interface definition conforms to the following syntax:

```
<interface> ::= <interface_dcl> | <forward_dcl>
<interface_dcl> ::= <interface_header> "{" <interface_body> "}"
<forward_dcl> ::= [ "abstract" | "local" ] ("interface" | "service") <identifier>
<interface_header> ::= [ "abstract" | "local" ]("interface" | "service") <identifier>
                       [ <interface_inheritance_spec> ]
<interface_body> ::= <export>*
<export> ::= <type_dcl> ";"
          |   <const_dcl> ";"
          |   <except_dcl> ";"
          |   <attr_dcl> ";"
          |   <op_dcl> ";"
          |   <type_id_dcl> ";"
          |   <type_prefix_dcl> ";"
```

Example of interface definition:

```
service MyService {
   ...
};
```

Interface Header

The interface header consists of three elements:

1. An optional modifier specifying if the interface is an abstract interface.

2. The interface name. The name must be preceded by the old OMG IDL 3.5 keyword `interface` or the new modern keyword `service`.
3. An optional inheritance specification.

An interface declaration containing the keyword `abstract` in its header, declares an abstract interface. Abstract interfaces have slightly different rules from *regular* interfaces, as described in section *Abstract interface*.

An interface declaration containing the keyword `local` in its header, declares a local interface. Local interfaces are not currently supported by the FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this, and explain the interface will be used as a *regular* interface.

Interface Inheritance Specification

The syntax for interface inheritance is as follows:

```
<interface_inheritance_spec> ::= ":" <interface_name> { "," <interface_name> }*
<interface_name> ::= <scoped_name>
<scoped_name> ::= <identifier>
            |   "::" <identifier>
            |   <scoped_name> "::" <identifier>
```

Each `<scoped_name>` in an `<interface_inheritance_spec>` must be the name of a previously defined interface or an alias to a previously defined interface.

Interface Body

The interface body contains the following kind of declarations:

- Constant declarations whose syntax is described in section *Constant Declaration*.
- Type declarations whose syntax is described in section *Type Declaration*.
- Exception declarations whose syntax is described in section *Exception Declaration*.
- Attribute declarations whose syntax is described in section *Attribute Declaration*.
- Operation declarations whose syntax is described in section *Operation Declaration*.

Abstract interface

An interface declaration contains the keyword `abstract` in its header, declares an abstract interface. The following special rule apply to abstract interfaces:

- Abstract interfaces may only inherit from other abstract interfaces.

8.5.4 Value Declaration

Value type declarations are supported by FIWARE Middleware IDL, but aren't by FIWARE Middleware. Any FIWARE Middleware IDL parser has to explain that these declarations are not used and the parser will ignore them.

8.5.5 Constant Declaration

A constant definition conforms to the following syntax:

```
<const_dcl> ::= "const" <const_type>
                <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
             |   <char_type>
             |   <wide_char_type>
             |   <boolean_type>
             |   <floating_pt_type>
             |   <string_type>
             |   <wide_string_type>
             |   <fixed_pt_const_type>
             |   <scoped_name>
             |   <octet_type>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
           |  <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
           |   <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
           |   <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
             |   <shift_expr> ">>" <add_expr>
             |   <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
           |   <add_expr> "+" <mult_expr>
           |   <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
            |   <mult_expr> "*" <unary_expr>
            |   <mult_expr> "/" <unary_expr>
            |   <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
             |   <primary_expr>
<unary_operator> ::= "-"
                 |   "+"
                 |   "~"
<primary_expr> ::= <scoped_name>
               |   <literal>
               |   "(" <const_exp> ")"
<literal> ::= <integer_literal>
          |   <string_literal>
          |   <wide_string_literal>
          |   <character_literal>
          |   <wide_character_literal>
          |   <fixed_pt_literal>
          |   <floating_pt_literal>
          |   <boolean_literal>
<boolean_literal> ::= "TRUE"
                  |   "FALSE"
<positive_int_const> ::= <const_exp>
```

Examples for constant declarations:

```
const string c_str = "HelloWorld";
const i32 c_int = 34;
const boolean c_bool = true;
```

8.5.6 Type Declaration

As in OMG IDL 3.5, FIWARE Middleware IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. The IDL uses the keyword `typedef` to associate a name with a data type.

Type declarations conform to the following syntax:

```
<type_dcl> ::= "typedef" <type_declarator>
            |   <struct_type>
            |   <union_type>
            |   <enum_type>
            |   "native" <simple_declarator>
            |   <constr_forward_decl>
<type_declarator> ::= <type_spec> <declarators>
```

For type declarations, FIWARE Middleware IDL defines a set of type specifiers to represent typed value. The syntax is as follows:

```
<type_spec> ::= <simple_type_spec>
             |   <constr_type_spec>
<simple_type_spec> ::= <base_type_spec>
                     |   <template_type_spec>
                     |   <scoped_name>
<base_type_spec> ::= <floating_pt_type>
                   |   <integer_type>
                   |   <char_type>
                   |   <wide_char_type>
                   |   <boolean_type>
                   |   <octet_type>
                   |   <any_type>
                   |   <object_type>
                   |   <value_base_type>
<template_type_spec> ::= <sequence_type>
                       |   <set_type>
                       |   <map_type>
                       |   <string_type>
                       |   <wide_string_type>
                       |   <fixed_pt_type>
<constr_type_spec> ::= <struct_type>
                     |   <union_type>
                     |   <enum_type>
<declarators> ::= <declarator> { "," <declarator> }*
<declarator> ::= <simple_declarator>
              |   <complex_declarator>
<simple_declarator> ::= <identifier>
<complex_declarator> ::= <array_declarator>
```

The <scoped_name\> in <simple_type_spec> must be a previously defined type introduced by a type declaration(<type_dcl> - see section *Type Declaration*).

The next subsections describe basic and constructed type specifiers.

Basic Types

The syntax for the supported basic types is as follows:

```
<floating_pt_type> ::= "float"
                     |   "double"
                     |   "long" "double"
                     |   "float32"
                     |   "float64"
                     |   "float128"
<integer_type> ::= <signed_int>
                 |   <unsigned_int>
<signed_int> ::= <signed_short_int>
               |   <signed_long_int>
               |   <signed_longlong_int>
<signed_short_int> ::= "short"
                     |   "i16"
```

```
<signed_long_int> ::= "long"
                    |    "i32"
<signed_longlong_int> ::= "long" "long"
                        |   "i64"
<unsigned_int> ::= <unsigned_short_int>
                |   <unsigned_long_int>
                |   <unsigned_longlong_int>
<unsigned_short_int> ::= "unsigned" "short"
                       |   "ui16"
<unsigned_long_int> ::= "unsigned" "long"
                      |   "ui32"
<unsigned_longlong_int> ::= "unsigned" "long" "long"
                          |   "ui64"
<char_type> ::= "char"
<wide_char_type> ::= "wchar"
<boolean_type> ::= "boolean"
<octet_type> ::= "octet"
              |   "byte"
<any_type> ::= "any"
```

Each IDL data type is mapped to a native data type via the appropriate language mapping. The syntax allows to use some OMG IDL 3.5 keywords and to use new modern keyword. For example, FIWARE Middleware IDL supports both keywords: `long` and `i32`.

The **any** type is not supported currently by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this.

8.5.7 Constructed Types

Constructed types are **structs**, **unions**, and **enums**.
Their syntax is as follows:

```
<type_dcl> ::= "typedef" <type_declarator>
            |   <struct_type>
            |   <union_type>
            |   <enum_type>
            |   "native" <simple_declarator>
            |   <constr_forward_decl>
<constr_type_spec> ::= <struct_type>
                     |   <union_type>
                     |   <enum_type>
<constr_forward_decl> ::= "struct" <identifier>
                        |   "union" <identifier>
```

Structures

The syntax for the `struct` type is as follows:

```
<struct_type> ::= "struct" <identifier> "{" <member_list> "}"
<member_list> ::= <member> +
<member> ::= <type_spec> <declarators> ";"
```

Example of struct syntax:

```
struct MyStruct {
    i32 f_int;
    string f_str;
    boolean f_bool;
};
```

Unions

The syntax for the `union` type is as follows:

```
<union_type> ::= "union" <identifier> "switch"
                 "(" <switch_type_spec> ")"
                 "{" <switch_body> "}"
<switch_type_spec> ::= <integer_type>
                     |   <char_type>
                     |   <boolean_type>
                     |   <enum_type>
                     |   <scoped_name>
<switch_body> ::= <case> +
<case> ::= <case_label> + <element_spec> ";"
<case_label> ::= "case" <const_exp> ":"
               |   "default" ":"
<element_spec> ::= <type_spec> <declarator>
```

The `<scoped_name>` in the `<switch_type_spec>` production must be a previously defined `integer`, `char`, `boolean` or `enum` type.

Example of union syntax:

```
union MyUnion switch(i32)
{
   case 1:
      i32 f_int;
   case 2:
      string f_str;
   default:
      boolean f_bool;
};
```

Enumerations

Enumerated types consist of ordered lists of identifiers.
The syntax is as follows:

```
<enum_type> ::= "enum" <identifier>
                "{" <enumerator> { "," <enumerator> } * "}"
<enumerator> ::= <identifier>
```

Example of an enumerated type:

```
enum MyEnum {
   ENUM1,
   ENUM2,
   ENUM3
};
```

Template Types

Template types are:

```
<template_type_spec> ::= <sequence_type>
                       |   <set_type>
                       |   <map_type>
                       |   <string_type>
                       |   <wide_string_type>
                       |   <fixed_pt_type>
```

Lists

The FIWARE Middleware IDL defined the template type `list`. A list is similar to the OMG IDL 3.5 `sequence` type. It is one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time). The syntax is as follows:

```
<sequence_type> ::= "sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
                |   "sequence" "<" <simple_type_spec> ">"
                |   "list" "<" <simple_type_spec> "," <positive_int_const> ">"
                |   "list" "<" <simple_type_spec> ">"
```

Examples of list type declarations:

```
list<string> mylist;
list<string, 32> myboundedlist;
```

Sets

The FIWARE Middleware IDL includes the template type `set`. At marshalling level it is like the template type `list`. But at a higher level, contrary to the list type, a set can only contain unique values. The syntax is as follows:

```
<set_type> ::= "set" "<" <simple_type_spec> "," <positive_int_const> ">"
                |   "set" "<" <simple_type_spec> ">"
```

Examples of set type declarations:

```
set<string> myset;
set<string, 32> myboundedset;
```

Maps

The FIWARE Middleware IDL includes the template type `map`, using the upcoming definition in OMG IDL 4.0. Maps are a collections, similar to lists, but items are associated with a *key*. Like lists, maps may be bounded or unbounded. The syntax is as follows:

```
<map_type> ::= "map" "<" <simple_type_spec> ","
                    <simple_type_spec> "," <positive_int_const> ">"
                |   "map" "<" <simple_type_spec> "," <simple_type_spec> ">"
```

Examples of map type declaration:

```
map<i32, string> mymap;
map<i32, string, 32> myboundedmap;
```

In CDR marshalling, objects of type map shall be represented according to the following equivalent OMG IDL 3.5 definition:

```
struct MapEntry_<key_type>_<value_type>[_<bound>] {
    <key_type> key;
    value_type> value;
};

typedef sequence<MapEntry_<key_type>_<value_type>[_<bound>][, <bound>]>
   Map_<key_type>_<value_type>[_<bound>];
```

Strings

The syntax for defining a string is as follows:

```
<string_type> ::= "string" "<" <positive_int_const> ">"
                |   "string"
```

Wstrings

The syntax for defining a wstring is as follows:

```
<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
                     |   "wstring"
```

Fixed Type

The `fixed` data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is a non-negative integer less than or equal to the total number of digits.

The `fixed` data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data types. The syntax of the fixed type is as follows:

```
<fixed_pt_type> ::= "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
<fixed_pt_const_type> ::= "fixed"
```

Complex Types

Arrays

The syntax for array is as follows:

```
<array_declarator> ::= <identifier> <fixed_array_size>+
<fixed_array_size> ::= "[" <positive_int_const> "]"
```

Example of array type declarations:

```
i32 myi32array[32];
string mystrarray[32];
```

Native Types

The syntax for native types is as follows:

```
<type_dcl> ::= "native" <simple_declarator>
<simple_declarator> ::= <identifier>
```

Native types are not supported by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and ignore this definition.

8.5.8 Exception Declaration

Exception declarations permit the declaration of struct-like data structures, which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

```
<except_dcl> ::= "exception" <identifier> "{" <member>* "}"
```

Example of an exception declaration:

```
exception myException {
   string msg;
   i32 code;
};
```

8.5.9 Operation Declaration

Operation declarations in OMG IDL 3.5 and FIWARE Middleware IDL are similar to C function declarations. The syntax is as follows:

```
<op_dcl> ::= [ <op_attribute> ] <op_type_spec>
             <identifier> <parameter_dcls>
             [ <raises_expr> ] [ <context_expr> ]
<op_attribute> ::= "oneway"
<op_type_spec> ::= <param_type_spec>
                 | "void"
```

Example of an operation declaration:

```
service myService {
    void set(i32 param);
    i32 get();
    i32 add(i32 param1, i32 param2) raises (myException);
};
```

An operation declaration consists of:

- An optional *operation attribute* that is supported by FIWARE Middleware IDL for backward compatibility. Operation attributes are described in section *Operation attribute*.
- The *type* of the operation's return result. Operations that do not return a result must specify the void type.
- An *identifier* that names the operation in the scope of the interface in which it is defined.
- A *parameter list* that specifies zero or more parameter declarations for the operation. Parameter declaration is described in section *Parameter Declarations*.
- An optional *raises expression* that indicates which exception may be raised as a result of an invocation of this operation. Raises expression are described in section *Raises Expressions*.
- An optional *context expression* that is inherited from OMG IDL 3.5, but FIWARE Middleware will not use. Context expressions are described in section *Context Expressions*.

Operation attribute

The syntax for operation attributes is as follows:

```
<op_attribute> ::= "oneway"
```

This attribute is supported in FIWARE Middleware for backward compatibility. But in FIWARE Middleware IDL the preferedby way to define a **oneway** function is using the **@Oneway** annotation as described in section *Oneway functions*.

Parameter Declarations

Parameter declarations in FIWARE Middleware IDL operation declarations have the following syntax:

```
<parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                   |   "(" ")"
<param_dcl> ::= [ <param_attribute> ] <param_type_spec> <simple_declarator>
<param_attribute> ::= "in"
                    |   "out"
```

```
                    |   "inout"
<raises_expr> ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"
<param_type_spec> ::= <base_type_spec>
                    |   <string_type>
                    |   <wide_string_type>
                    |   <scoped_name>
```

The FIWARE Middleware IDL will *not* use output parameters, as modern IDLs do. It supports the keywords `in`, `inout`, and `out`, but any FIWARE Middleware IDL parser will inform users all parameters will be input parameters.

Raises Expressions

There are two kinds of raises expressions.

Raises Expression

A raises expression specifies which exceptions may be raised as a result of an invocation of the operation or accessing a readonly attribute. The syntax for its specification is as follows:

```
<raises_expr> ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"
```

The `<scoped_name>`s in the raises expression must be previously defined exceptions.

getraises and setraises Expression

The syntax is as follows:

```
<attr_raises_expr> ::= <get_excep_expr> [ <set_excep_expr> ]
                     |   <set_excep_expr>
<get_excep_expr> ::= "getraises" <exception_list>
<set_excep_expr> ::= "setraises" <exception_list>
<exception_list> ::= "(" <scoped_name> { "," <scoped_name> }* ")"
```

`getraises` and `setraises` expressions are used in attribute declarations. Like in attribute declarations, theses expressions are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these expressions.

Context Expressions

The syntax for content expressions is as follows:

```
<context_expr> ::= "context" "(" <string_literal> { "," <string_literal> }* ")"
```

Context expressions are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these expressions.

8.5.10 Attribute Declaration

The syntax for attribute declarations is as follows:

```
<attr_dcl> ::= <readonly_attr_spec> | <attr_spec>
<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
                         <readonly_attr_declarator>
<readonly_attr_declarator> ::= <simple_declarator> <raises_expr>
                             |   <simple_declarator> { "," <simple_declarator> }*
<attr_spec> ::= "attribute" <param_type_spec>
```

```
                <attr_declarator>
<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
                  |   <simple_declarator> { "," <simple_declarator> }*
```

These declarations are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these declarations.

8.5.11 Repository Identity Related Declarations

The syntax for repository identity related declarations is as follows:

```
<type_id_dcl> ::= "typeid" <scoped_name> <string_literal>
<type_prefix_dcl> ::= "typeprefix" <scoped_name> <string_literal>
```

These declarations are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these declarations.

8.5.12 Event Declaration

The syntax for event declarations is as follows:

```
<event> ::= ( <event_dcl> | <event_abs_dcl> | <event_forward_dcl>)
<event_forward_dcl> ::= [ "abstract" ] "eventtype" <identifier>
<event_abs_dcl> ::= "abstract" "eventtype" <identifier>
                    [ <value_inheritance_spec> ]
                    "{" <export>* "}"
<event_dcl> ::= <event_header> "{" <value_element> * "}"
<event_header> ::= [ "custom" ] "eventtype"
                   <identifier> [ <value_inheritance_spec> ]
```

These declarations are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these declarations.

8.5.13 Component Declaration

The syntax for component declarations is as follows:

```
<component> ::= <component_dcl> | <component_forward_dcl>
<component_forward_dcl> ::= "component" <identifier>
<component_dcl> ::= <component_header> "{" <component_body> "}"
<component_header> ::= "component" <identifier>
                       [ <component_inheritance_spec> ]
                       [ <supported_interface_spec> ]
<supported_interface_spec> ::= "supports" <scoped_name> { "," <scoped_name> }*
<component_inheritance_spec> ::= ":" <scoped_name>
<component_body> ::= <component_export>*
<component_export> ::= <provides_dcl> ";"
                   |   <uses_dcl> ";"
                   |   <emits_dcl> ";"
                   |   <publishes_dcl> ";"
                   |   <consumes_dcl> ";"
                   |   <attr_dcl> ";"
<provides_dcl> ::= "provides" <interface_type> <identifier>
<interface_type> ::= <scoped_name> | "Object"
<uses_dcl> ::= "uses" [ "multiple" ] <interface_type> <identifier>
<emits_dcl> ::= "emits" <scoped_name> <identifier>
<publishes_dcl> ::= "publishes" <scoped_name> <identifier>
<consumes_dcl> ::= "consumes" <scoped_name> <identifier>
```

These declarations are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these declarations.

8.5.14 Home Declaration

The syntax for home declarations is as follows:

```
<home_dcl> ::= <home_header> <home_body>
<home_header> ::= "home" <identifier>
                  [ <home_inheritance_spec> ]
                  [ <supported_interface_spec> ]
                  "manages" <scoped_name>
                  [ <primary_key_spec> ]
<home_inheritance_spec> ::= ":" <scoped_name>
<primary_key_spec> ::= "primarykey" <scoped_name>
<home_body> ::= "{" <home_export>* "}"
<home_export ::= <export>
             |   <factory_dcl> ";"
             |   <finder_dcl> ";"
<factory_dcl> ::= "factory" <identifier>
                  "(" [ <init_param_decls> ] ")"
                  [ <raises_expr> ]
<finder_dcl> ::= "finder" <identifier>
                 "(" [ <init_param_decls> ] ")"
                 [ <raises_expr> ]
```

These declarations are supported by FIWARE Middleware IDL but not by FIWARE Middleware. Any FIWARE Middleware IDL parser has to inform users about this and it will ignore these declarations.

8.5.15 Annotation Declaration

An annotation type is a form of aggregated type similar to a structure with members that could be given constant values. FIWARE Middleware IDL annotations are the ones used in future OMG IDL 4.0, whose are similar to the one provided by Java.

An annotation is defined with a header and a body. The syntax is as follows:

```
<annotation_dcl> ::= <annotation_def> ";"
                 |   <annotation_forward_dcl>
<annotation_def> ::= <annotation_header> "{" <annotation_body> "}"
```

Annotation Header

The header consists of: - The keyword `@annotation`, followed by an identifier that is the name given to the annotation. - Optionally a single inheritance specification.

The syntax of an annotation header is as follows:

```
<annotation_header> ::= "@annotation" <identifier> [<annotation_inheritance_spec>]
<annotation_inheritance_spec> ::= ":" <scoped_name>
```

Annotation Body

The body contains a list of zero to several member embedded within braces. Each attribute consists of: - The keyword `attribute`. - The member type, which must be a constant type `<const_type>`. - The name given to the member. - An optional default value, given by a constant expression `<const_expr>` prefixed with the keyword **default**. The constant expression must be compatible with the member type.

The syntax of annotation body is as follows:

```
<annotation_body> ::= <annotation_member>*
<annotation_member> ::= <const_type> <simple_declarator>
                        [ "default" <const_expr> ] ";"
```

Annotation Forwarding

Annotations may also be forward-declared, which allow referencing an annotation whose definition is not provided yet.

The syntax of a forwarding annotation is as follows:

```
<annotation_forward_dcl> ::= "@annotation" <scoped_name>
```

8.5.16 Annotation Application

An annotation, once its type defined, may be applied using the following syntax:

```
<annotation_appl> ::= "@" <scoped_name> [ "(" [ <annotation_appl_params> ] ")" ]
<annotation_appl_params> ::= <const_exp>
                         |   <annotation_appl_param> { "," <annotation_appl_param> }*
<annotation_appl_param> ::= <identifier> "=" <const_exp>
```

Applying an annotation consists in prefixing the element under annotation with: - The annotation name prefixed with a commercial at (@) - Followed by the list of values given to the annotation's members within parentheses and separated by comma. Each parameter value consist in: - The name of the member - The symbol '=' - A constant expression, whose type must be compatible with the member's declaration.

Members may be indicated in any order. Members with no default value must be given a value. Members with default value may be omitted. In that case, the member is considered as valued with its default value.

Two shortened forms exist: - In case, there is no member, the annotation application may be as short as just the name of the annotation prefixed by '@' - In case there is only one member, the annotation application may be as short as the name of the annotation prefixed by '@' and followed with the constant value of that unique member within (). The type of the provided constant expression must compatible with the members' declaration

An annotation may be applied to almost any IDL construct or sub-construct. Applying and annotation consists actually in adding the related meta-data to the element under annotation. Full FIWARE Middleware IDL described in section *Appendix B: FIWARE Middleware IDL Grammar* shows this.

8.5.17 Built-in annotations

FIWARE Middleware will support some built-in annotations, that any user can use in IDL files.

Member IDs

All members of aggregated types have an integral member ID that uniquely identifies them within their defining type. Because OMG IDL 3.5 has no native syntax for expressing this information, IDs by default are defined implicitly based on the members' relative declaration order. The first member (which, in a union type, is the discriminator) has ID 0, the second ID 1, the third ID 2, and so on.

As described in OMG IDL for X-Types, these implicit ID assignments can be overridden by using the "ID" annotation interface. The equivalent definition of this type is as follows:

```
@annotation ID {
    attribute ui32 value;
};
```

Optional members

The FIWARE Middleware IDL allows to declare a member optional, applying the "Optional" annotation. The definitions is as follows:

```
@annotation Optional {
    attribute boolean value default true;
};
```

The CDR marshalling for this optional members is defined in IDL X-Types standard.

Key members

The FIWARE Middleware IDL allows to declare a member as part of the key, applying the "Key" annotation. This will be needed for future pub/sub communication using DDS. The definitions is as follows:

```
@annotation Key {
    attribute boolean value default true;
};
```

Oneway functions

The FIWARE Middleware IDL allows to declare a function as oneway method, applying the "Oneway" annotation. The definitions is as follows:

```
@annotation Oneway {
    attribute boolean value default true;
};
```

Asynchronous functions

The FIWARE Middleware IDL allows to declare a function as asynchronous method, applying the "Async" annotation. The definitions is as follows:

```
@annotation Async {
    attribute boolean value default true;
}
```

8.6 IDL Complete Example

This section provides a complete example of a FIWARE Middleware IDL file:

```
typedef list<i32> accountList;
// @Encrypted annotation applies to map type declaration.
@Encrypted(mode="sha1")
typedef map<string, i32> userAccountMap;

// @CppMapping annotation applies to the namespace
@CppMapping
namespace ThiefBank {

   // @Authentication annotation applies to the service.
   @Authentication(mechanism="login")
   service AccountService {
      // @Security annotation applies to the structure declaration.
      @Security
      struct AccountInfo {
```

```
            i32 count;
           string user;
       };

       @Oneway
       void setAccounts(userAccountMap uamap);

       //@Encrypted annotation applies to the parameter "account".
       @Oneway
       void setAccount(string user, @Encrypted i32 account);

       //@Encrypted annotation applies to the return value.
       @Encrypted
       AccountInfo get(string user);

       //@FullEncrypted annotation applies to the operation.
       @FullEncrypted(mode="sha1")
       AccountInfo get_secured(string user);
    };
};
```

The annotations used in previous example are defined as follows:

```
@annotation CppMapping {
   attribute boolean value default true;
};

@annotation Authentication {
   attribute string mechanism default "none";
};

@annotation Encrypted {
   attribute string mode default "sha512";
};

@annotation FullEncrypted {
   attribute string mode default "sha512";
};

@annotation Security {
   attribute boolean active default true;
};
```

8.7 Appendix A: Changes from OMG IDL 3.5

This section summarizes in one block all changes applied from OMG IDL 3.5 to the FIWARE Middleware IDL:

- Modern keyword for modules. New keyword is `namespace`. See section *Module Declaration*.
- Modern keyword for interfaces. New keyword is `service`. See section *Interface Header*.
- Modern keywords for basic types. See section *Basic Types*.
- New template types. See section *Template Types*.
- FIWARE Middleware IDL only uses input parameters. See section *Parameter Declarations*
- FIWARE Middleware IDL adds annotations. See sections *Annotation Declaration* and *Annotation Application*.

Also FIWARE Middleware IDL does **not** use and support (and therefore ignores) several OMG IDL 3.5 constructs:

- Import declarations. See section *Import Declaration*.

- Value declarations. See section *Value Declaration*.
- 'Any' type. See section *Basic Types*.
- Native types. See section *Native Types*.
- Context expressions. See section *Context Expressions*.
- Attribute declarations. See section *Attribute Declaration*.
- Repository Identity Related Declarations. See section *Repository Identity Related Declarations*.
- Event declarations. See section *Event Declaration*.
- Component declarations. See section *Component Declaration*.
- Home declarations. See section *Home Declaration*.

8.8 Appendix B: FIWARE Middleware IDL Grammar

```
<specification> ::= <import>* <definition>+
<definition> ::= <type_dcl> ";"
             |   <const_dcl> ";"
             |   <except_dcl> ";"
             |   <interface> ";"
             |   <module> ";"
             |   <value> ";"
             |   <type_id_dcl> ";"
             |   <type_prefix_dcl> ";"
             |   <event> ";"
             |   <component> ";"
             |   <home_dcl> ";"
             |   <annotation_dcl> ";"
             |   <annotation_appl> <definition>
<annotation_dcl> ::= <annotation_def> ";"
                 |   <annotation_forward_dcl>
<annotation_def> ::= <annotation_header> "{" <annotation_body> "}"
<annotation_header> ::= "@annotation" <identifier> [<annotation_inheritance_spec>]
<annotation_inheritance_spec> ::= ":" <scoped_name>
<annotation_body> ::= <annotation_member>*
<annotation_member> ::= <const_type> <simple_declarator>
                        [ "default" <const_expr> ] ";"
<annotation_forward_dcl> ::= "@annotation" <scoped_name>
<annotation_appl> ::= "@" <scoped_name> [ "(" [ <annotation_appl_params> ] ")" ]
<annotation_appl_params> ::= <const_exp>
                         |   <annotation_appl_param> { "," <annotation_appl_param> }*
<annotation_appl_param> ::= <identifier> "=" <const_exp>
<module> ::= ("module" | "namespace") <identifier> "{" <definition> + "}"
<interface> ::= <interface_dcl>
            |   <forward_dcl>
<interface_dcl> ::= <interface_header> "{" <interface_body> "}"
<forward_dcl> ::= [ "abstract" | "local" ] ("interface" | "service") <identifier>
<interface_header> ::= [ "abstract" | "local" ] ("interface" | "service") <identifier>
                       [ <interface_inheritance_spec> ]
<interface_body> ::= <export>*
<export> ::= <type_dcl> ";"
         |   <const_dcl> ";"
         |   <except_dcl> ";"
         |   <attr_dcl> ";"
         |   <op_dcl> ";"
         |   <type_id_dcl> ";"
         |   <type_prefix_dcl> ";"
         |   <annotation_appl> <export>
<interface_inheritance_spec> ::= ":" <interface_name>
```

```
                                { "," <interface_name> }*
<interface_name> ::= <scoped_name>
<scoped_name> ::= <identifier>
            |   "::" <identifier>
            |   <scoped_name> "::" <identifier>
<value> ::= ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::= [ "abstract" ] "valuetype" <identifier>
<value_box_dcl> ::= "valuetype" <identifier> <type_spec>
<value_abs_dcl> ::= "abstract" "valuetype" <identifier>
                    [ <value_inheritance_spec> ]
                    "{" <export>* "}"
<value_dcl> ::= <value_header> "{" <value_element>* "}"
<value_header> ::= ["custom" ] "valuetype" <identifier>
                   [ <value_inheritance_spec> ]
<value_inheritance_spec> ::= [ ":" [ "truncatable" ] <value_name>
                             { "," <value_name> }* ]
                             [ "supports" <interface_name>
                             { "," <interface_name> }* ]
<value_name> ::= <scoped_name>
<value_element> ::= <export> | <state_member> | <init_dcl>
<state_member> ::= ( "public" | "private" )
                   <type_spec> <declarators> ";"
<init_dcl> ::= "factory" <identifier>
               "(" [ <init_param_decls> ] ")"
               [ <raises_expr> ] ";"
<init_param_decls> ::= <init_param_decl> { "," <init_param_decl> }*
<init_param_decl> ::= <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute> ::= "in"
<const_dcl> ::= "const" <const_type>
                <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
             |   <char_type>
             |   <wide_char_type>
             |   <boolean_type>
             |   <floating_pt_type>
             |   <string_type>
             |   <wide_string_type>
             |   <fixed_pt_const_type>
             |   <scoped_name>
             |   <octet_type>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
           |   <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
           |   <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
           |   <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
             |   <shift_expr> ">>" <add_expr>
             |   <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
           |   <add_expr> "+" <mult_expr>
           |   <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
            |   <mult_expr> "*" <unary_expr>
            |   <mult_expr> "/" <unary_expr>
            |   <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
             |   <primary_expr>
<unary_operator> ::= "-"
                 |   "+"
                 |   "~"
<primary_expr> ::= <scoped_name>
```

```
               |   <literal>
               |   "(" <const_exp> ")"
<literal> ::= <integer_literal>
          |   <string_literal>
          |   <wide_string_literal>
          |   <character_literal>
          |   <wide_character_literal>
          |   <fixed_pt_literal>
          |   <floating_pt_literal>
          |   <boolean_literal>
<boolean_literal> ::= "TRUE"
                  |   "FALSE"
<positive_int_const> ::= <const_exp>
<type_dcl> ::= "typedef" <type_declarator>
           |   <struct_type>
           |   <union_type>
           |   <enum_type>
           |   "native" <simple_declarator>
           |   <constr_forward_decl>
<type_declarator> ::= <type_spec> <declarators>
<type_spec> ::= <simple_type_spec>
            |   <constr_type_spec>
<simple_type_spec> ::= <base_type_spec>
                   |   <template_type_spec>
                   |   <scoped_name>
<base_type_spec> ::= <floating_pt_type>
                 |   <integer_type>
                 |   <char_type>
                 |   <wide_char_type>
                 |   <boolean_type>
                 |   <octet_type>
                 |   <any_type>
                 |   <object_type>
                 |   <value_base_type>
<template_type_spec> ::= <sequence_type>
                     |   <set_type>
                     |   <map_type>
                     |   <string_type>
                     |   <wide_string_type>
                     |   <fixed_pt_type>
<constr_type_spec> ::= <struct_type>
                   |   <union_type>
                   |   <enum_type>
<declarators> ::= <declarator> { "," <declarator> }*
<declarator> ::= <simple_declarator>
             |   <complex_declarator>
<simple_declarator> ::= <identifier>
<complex_declarator> ::= <array_declarator>
<floating_pt_type> ::= "float"
                   |   "double"
                   |   "long" "double"
                   |   "float32"
                   |   "float64"
                   |   "float128"
<integer_type> ::= <signed_int>
               |   <unsigned_int>
<signed_int> ::= <signed_short_int>
             |   <signed_long_int>
             |   <signed_longlong_int>
<signed_short_int> ::= "short"
                   |   "i16"
<signed_long_int> ::= "long"
                  |    "i32"
```

```
<signed_longlong_int> ::= "long" "long"
                        |   "i64"
<unsigned_int> ::= <unsigned_short_int>
                |   <unsigned_long_int>
                |   <unsigned_longlong_int>
<unsigned_short_int> ::= "unsigned" "short"
                       |   "ui16"
<unsigned_long_int> ::= "unsigned" "long"
                      |   "ui32"
<unsigned_longlong_int> ::= "unsigned" "long" "long"
                          |   "ui64"
<char_type> ::= "char"
<wide_char_type> ::= "wchar"
<boolean_type> ::= "boolean"
<octet_type> ::= "octet"
              |   "byte"
<any_type> ::= "any"
<object_type> ::= "Object"
<struct_type> ::= "struct" <identifier> "{" <member_list> "}"
<member_list> ::= <member>+
<member> ::= <type_spec> <declarators> ";"
          |   <annotation_appl> <type_spec> <declarators> ";"
<union_type> ::= "union" <identifier> "switch"
                 "(" <switch_type_spec> ")"
                 "{" <switch_body> "}"
<switch_type_spec> ::= <integer_type>
                     |   <char_type>
                     |   <boolean_type>
                     |   <enum_type>
                     |   <scoped_name>
<switch_body> ::= <case> +
<case> ::= <case_label> + <element_spec> ";"
<case_label> ::= "case" <const_exp> ":"
              |   "default" ":"
<element_spec> ::= <type_spec> <declarator>
                |   <annotation_appl> <type_spec> <declarator>
<enum_type> ::= "enum" <identifier>
                "{" <enumerator> { "," <enumerator> } * "}"
<enumerator> ::= <identifier>
<sequence_type> ::= "sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
                 |   "sequence" "<" <simple_type_spec> ">"
                 |   "list" "<" <simple_type_spec> "," <positive_int_const> ">"
                 |   "list" "<" <simple_type_spec> ">"
<set_type> ::= "set" "<" <simple_type_spec> "," <positive_int_const> ">"
                 |   "set" "<" <simple_type_spec> ">"
<map_type> ::= "map" "<" <simple_type_spec> ","
                     <simple_type_spec> "," <positive_int_const> ">"
                 |   "map" "<" <simple_type_spec> "," <simple_type_spec> ">"
<string_type> ::= "string" "<" <positive_int_const> ">"
               |   "string"
<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
                     |   "wstring"
<array_declarator> ::= <identifier> <fixed_array_size>+
<fixed_array_size> ::= "[" <positive_int_const> "]"
<attr_dcl> ::= <readonly_attr_spec>
            |   <attr_spec>
<except_dcl> ::= "exception" <identifier> "{" <member>* "}"
<op_dcl> ::= [ <op_attribute> ] <op_type_spec>
             <identifier> <parameter_dcls>
             [ <raises_expr> ] [ <context_expr> ]
<op_attribute> ::= "oneway"
<op_type_spec> ::= <param_type_spec>
                 | "void"
```

```
<parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> } * ")"
                 |   "(" ")"
<param_dcl> ::= [<param_attribute>] <param_type_spec> <simple_declarator>
            |   [<param_attribute>] <annotation_appl>
                <param_type_spec> <simple_declarator>
<param_attribute> ::= "in"
                  |   "out"
                  |   "inout"
<raises_expr> ::= "raises" "(" <scoped_name>
                  { "," <scoped_name> } * ")"
<context_expr> ::= "context" "(" <string_literal>
                   { "," <string_literal> } * ")"
<param_type_spec> ::= <base_type_spec>
                  |   <string_type>
                  |   <wide_string_type>
                  |   <scoped_name>
<fixed_pt_type> ::= "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
<fixed_pt_const_type> ::= "fixed"
<value_base_type> ::= "ValueBase"
<constr_forward_decl> ::= "struct" <identifier>
                      |   "union" <identifier>
<import> ::= "import" <imported_scope> ";"
<imported_scope> ::= <scoped_name> | <string_literal>
<type_id_dcl> ::= "typeid" <scoped_name> <string_literal>
<type_prefix_dcl> ::= "typeprefix" <scoped_name> <string_literal>
<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
                         <readonly_attr_declarator>
<readonly_attr_declarator> ::= <simple_declarator> <raises_expr>
                           |   <simple_declarator>
                               { "," <simple_declarator> }*
<attr_spec> ::= "attribute" <param_type_spec>
                <attr_declarator>
<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
                  |   <simple_declarator>
                      { "," <simple_declarator> }*
<attr_raises_expr> ::= <get_excep_expr> [ <set_excep_expr> ]
                   |   <set_excep_expr>
<get_excep_expr> ::= "getraises" <exception_list>
<set_excep_expr> ::= "setraises" <exception_list>
<exception_list> ::= "(" <scoped_name>
                     { "," <scoped_name> } * ")"
<component> ::= <component_dcl>
            |   <component_forward_dcl>
<component_forward_dcl> ::= "component" <identifier>
<component_dcl> ::= <component_header>
                    "{" <component_body> "}"
<component_header> ::= "component" <identifier>
                       [ <component_inheritance_spec> ]
                       [ <supported_interface_spec> ]
<supported_interface_spec> ::= "supports" <scoped_name>
                               { "," <scoped_name> }*
<component_inheritance_spec> ::= ":" <scoped_name>
<component_body> ::= <component_export>*
<component_export> ::= <provides_dcl> ";"
                   |   <uses_dcl> ";"
                   |   <emits_dcl> ";"
                   |   <publishes_dcl> ";"
                   |   <consumes_dcl> ";"
                   |   <attr_dcl> ";"
<provides_dcl> ::= "provides" <interface_type> <identifier>
<interface_type> ::= <scoped_name>
                 |   "Object"
<uses_dcl> ::= "uses" [ "multiple" ]
```

```
               < interface_type> <identifier>
<emits_dcl> ::= "emits" <scoped_name> <identifier>
<publishes_dcl> ::= "publishes" <scoped_name> <identifier>
<consumes_dcl> ::= "consumes" <scoped_name> <identifier>
<home_dcl> ::= <home_header> <home_body>
<home_header> ::= "home" <identifier>
                  [ <home_inheritance_spec> ]
                  [ <supported_interface_spec> ]
                  "manages" <scoped_name>
                  [ <primary_key_spec> ]
<home_inheritance_spec> ::= ":" <scoped_name>
<primary_key_spec> ::= "primarykey" <scoped_name>
<home_body> ::= "{" <home_export>* "}"
<home_export ::= <export>
             |   <factory_dcl> ";"
             |   <finder_dcl> ";"
<factory_dcl> ::= "factory" <identifier>
                  "(" [ <init_param_decls> ] ")"
                  [ <raises_expr> ]
<finder_dcl> ::= "finder" <identifier>
                 "(" [ <init_param_decls> ] ")"
                 [ <raises_expr> ]
<event> ::= ( <event_dcl> | <event_abs_dcl> |
            <event_forward_dcl>)
<event_forward_dcl> ::= [ "abstract" ] "eventtype" <identifier>
<event_abs_dcl> ::= "abstract" "eventtype" <identifie
                    [ <value_inheritance_spec> ]
                    "{" <export>* "}"
<event_dcl> ::= <event_header> "{" <value_element> * "}"
<event_header> ::= [ "custom" ] "eventtype"
                   <identifier> [ <value_inheritance_spec> ]
```

8.9 Appendix C: OMG IDL 3.5 Grammar

```
<specification> ::= <import>* <definition>+
<definition> ::= <type_dcl> ";"
             |   <const_dcl> ";"
             |   <except_dcl> ";"
             |   <interface> ";"
             |   <module> ";"
             |   <value> ";"
             |   <type_id_dcl> ";"
             |   <type_prefix_dcl> ";"
             |   <event> ";"
             |   <component> ";"
             |   <home_dcl> ";"
<module> ::= "module" <identifier> "{" <definition> + "}"
<interface> ::= <interface_dcl>
            |   <forward_dcl>
<interface_dcl> ::= <interface_header> "{" <interface_body> "}"
<forward_dcl> ::= [ "abstract" | "local" ] "interface" <identifier>
<interface_header> ::= [ "abstract" | "local" ] "interface" <identifier>
                       [ <interface_inheritance_spec> ]
<interface_body> ::= <export>*
<export> ::= <type_dcl> ";"
         |   <const_dcl> ";"
         |   <except_dcl> ";"
         |   <attr_dcl> ";"
         |   <op_dcl> ";"
         |   <type_id_dcl> ";"
```

```
          |   <type_prefix_dcl> ";"
<interface_inheritance_spec> ::= ":" <interface_name>
                                 { "," <interface_name> }*
<interface_name> ::= <scoped_name>
<scoped_name> ::= <identifier>
            |   "::" <identifier>
            |   <scoped_name> "::" <identifier>
<value> ::= ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::= [ "abstract" ] "valuetype" <identifier>
<value_box_dcl> ::= "valuetype" <identifier> <type_spec>
<value_abs_dcl> ::= "abstract" "valuetype" <identifier>
                    [ <value_inheritance_spec> ]
                    "{" <export>* "}"
<value_dcl> ::= <value_header> "{" < value_element>* "}"
<value_header> ::= ["custom" ] "valuetype" <identifier>
                   [ <value_inheritance_spec> ]
<value_inheritance_spec> ::= [ ":" [ "truncatable" ] <value_name>
                             { "," <value_name> }* ]
                             [ "supports" <interface_name>
                             { "," <interface_name> }* ]
<value_name> ::= <scoped_name>
<value_element> ::= <export> | < state_member> | <init_dcl>
<state_member> ::= ( "public" | "private" )
                   <type_spec> <declarators> ";"
<init_dcl> ::= "factory" <identifier>
               "(" [ <init_param_decls> ] ")"
               [ <raises_expr> ] ";"
<init_param_decls> ::= <init_param_decl> { "," <init_param_decl> }*
<init_param_decl> ::= <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute> ::= "in"
<const_dcl> ::= "const" <const_type>
                <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
             |   <char_type>
             |   <wide_char_type>
             |   <boolean_type>
             |   <floating_pt_type>
             |   <string_type>
             |   <wide_string_type>
             |   <fixed_pt_const_type>
             |   <scoped_name>
             |   <octet_type>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
           |   <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
           |   <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
           |   <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
             |   <shift_expr> ">>" <add_expr>
             |   <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
           |   <add_expr> "+" <mult_expr>
           |   <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
            |   <mult_expr> "*" <unary_expr>
            |   <mult_expr> "/" <unary_expr>
            |   <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
             |   <primary_expr>
<unary_operator> ::= "-"
                 |   "+"
```

```
                |   "~"
<primary_expr> ::= <scoped_name>
               |   <literal>
               |   "(" <const_exp> ")"
<literal> ::= <integer_literal>
          |   <string_literal>
          |   <wide_string_literal>
          |   <character_literal>
          |   <wide_character_literal>
          |   <fixed_pt_literal>
          |   <floating_pt_literal>
          |   <boolean_literal>
<boolean_literal> ::= "TRUE"
                  |   "FALSE"
<positive_int_const> ::= <const_exp>
<type_dcl> ::= "typedef" <type_declarator>
           |   <struct_type>
           |   <union_type>
           |   <enum_type>
           |   "native" <simple_declarator>
           |   <constr_forward_decl>
<type_declarator> ::= <type_spec> <declarators>
<type_spec> ::= <simple_type_spec>
            |   <constr_type_spec>
<simple_type_spec> ::= <base_type_spec>
                   |   <template_type_spec>
                   |   <scoped_name>
<base_type_spec> ::= <floating_pt_type>
                 |   <integer_type>
                 |   <char_type>
                 |   <wide_char_type>
                 |   <boolean_type>
                 |   <octet_type>
                 |   <any_type>
                 |   <object_type>
                 |   <value_base_type>
<template_type_spec> ::= <sequence_type>
                     |   <string_type>
                     |   <wide_string_type>
                     |   <fixed_pt_type>
<constr_type_spec> ::= <struct_type>
                   |   <union_type>
                   |   <enum_type>
<declarators> ::= <declarator> { "," <declarator> }*
<declarator> ::= <simple_declarator>
             |   <complex_declarator>
<simple_declarator> ::= <identifier>
<complex_declarator> ::= <array_declarator>
<floating_pt_type> ::= "float"
                   |   "double"
                   |   "long" "double"
<integer_type> ::= <signed_int>
               |   <unsigned_int>
<signed_int> ::= <signed_short_int>
             |   <signed_long_int>
             |   <signed_longlong_int>
<signed_short_int> ::= "short"
<signed_long_int> ::= "long"
<signed_longlong_int> ::= "long" "long"
<unsigned_int> ::= <unsigned_short_int>
               |   <unsigned_long_int>
               |   <unsigned_longlong_int>
<unsigned_short_int> ::= "unsigned" "short"
```

```
<unsigned_long_int> ::= "unsigned" "long"
<unsigned_longlong_int> ::= "unsigned" "long" "long"
<char_type> ::= "char"
<wide_char_type> ::= "wchar"
<boolean_type> ::= "boolean"
<octet_type> ::= "octet"
<any_type> ::= "any"
<object_type> ::= "Object"
<struct_type> ::= "struct" <identifier> "{" <member_list> "}"
<member_list> ::= <member> +
<member> ::= <type_spec> <declarators> ";"
<union_type> ::= "union" <identifier> "switch"
                 "(" <switch_type_spec> ")"
                 "{" <switch_body> "}"
<switch_type_spec> ::= <integer_type>
                   |   <char_type>
                   |   <boolean_type>
                   |   <enum_type>
                   |   <scoped_name>
<switch_body> ::= <case> +
<case> ::= <case_label> + <element_spec> ";"
<case_label> ::= "case" <const_exp> ":"
             |   "default" ":"
<element_spec> ::= <type_spec> <declarator>
<enum_type> ::= "enum" <identifier>
               "{" <enumerator> { "," <enumerator> } * "}"
<enumerator> ::= <identifier>
<sequence_type> ::= "sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
                |   "sequence" "<" <simple_type_spec> ">"
<string_type> ::= "string" "<" <positive_int_const> ">"
              |   "string"
<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
                   |   "wstring"
<array_declarator> ::= <identifier> <fixed_array_size>+
<fixed_array_size> ::= "[" <positive_int_const> "]"
<attr_dcl> ::= <readonly_attr_spec>
           |   <attr_spec>
<except_dcl> ::= "exception" <identifier> "{" <member>* "}"
<op_dcl> ::= [ <op_attribute> ] <op_type_spec>
             <identifier> <parameter_dcls>
             [ <raises_expr> ] [ <context_expr> ]
<op_attribute> ::= "oneway"
<op_type_spec> ::= <param_type_spec>
               | "void"
<parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> } * ")"
                 |   "(" ")"
<param_dcl> ::= <param_attribute> <param_type_spec> <simple_declarator>
<param_attribute> ::= "in"
                  |   "out"
                  |   "inout"
<raises_expr> ::= "raises" "(" <scoped_name>
                  { "," <scoped_name> } * ")"
<context_expr> ::= "context" "(" <string_literal>
                   { "," <string_literal> } * ")"
<param_type_spec> ::= <base_type_spec>
                  |   <string_type>
                  |   <wide_string_type>
                  |   <scoped_name>
<fixed_pt_type> ::= "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
<fixed_pt_const_type> ::= "fixed"
<value_base_type> ::= "ValueBase"
<constr_forward_decl> ::= "struct" <identifier>
                      |   "union" <identifier>
```

```
<import> ::= "import" <imported_scope> ";"
<imported_scope> ::= <scoped_name> | <string_literal>
<type_id_dcl> ::= "typeid" <scoped_name> <string_literal>
<type_prefix_dcl> ::= "typeprefix" <scoped_name> <string_literal>
<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
                         <readonly_attr_declarator>
<readonly_attr_declarator> ::= <simple_declarator> <raises_expr>
                           |   <simple_declarator>
                               { "," <simple_declarator> }*
<attr_spec> ::= "attribute" <param_type_spec>
                <attr_declarator>
<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
                  |   <simple_declarator>
                      { "," <simple_declarator> }*
<attr_raises_expr> ::= <get_excep_expr> [ <set_excep_expr> ]
                   |   <set_excep_expr>
<get_excep_expr> ::= "getraises" <exception_list>
<set_excep_expr> ::= "setraises" <exception_list>
<exception_list> ::= "(" <scoped_name>
                     { "," <scoped_name> } * ")"
<component> ::= <component_dcl>
            |   <component_forward_dcl>
<component_forward_dcl> ::= "component" <identifier>
<component_dcl> ::= <component_header>
                    "{" <component_body> "}"
<component_header> ::= "component" <identifier>
                       [ <component_inheritance_spec> ]
                       [ <supported_interface_spec> ]
<supported_interface_spec> ::= "supports" <scoped_name>
                               { "," <scoped_name> }*
<component_inheritance_spec> ::= ":" <scoped_name>
<component_body> ::= <component_export>*
<component_export> ::= <provides_dcl> ";"
                   |   <uses_dcl> ";"
                   |   <emits_dcl> ";"
                   |   <publishes_dcl> ";"
                   |   <consumes_dcl> ";"
                   |   <attr_dcl> ";"
<provides_dcl> ::= "provides" <interface_type> <identifier>
<interface_type> ::= <scoped_name>
                 |   "Object"
<uses_dcl> ::= "uses" [ "multiple" ]
               < interface_type> <identifier>
<emits_dcl> ::= "emits" <scoped_name> <identifier>
<publishes_dcl> ::= "publishes" <scoped_name> <identifier>
<consumes_dcl> ::= "consumes" <scoped_name> <identifier>
<home_dcl> ::= <home_header> <home_body>
<home_header> ::= "home" <identifier>
                  [ <home_inheritance_spec> ]
                  [ <supported_interface_spec> ]
                  "manages" <scoped_name>
                  [ <primary_key_spec> ]
<home_inheritance_spec> ::= ":" <scoped_name>
<primary_key_spec> ::= "primarykey" <scoped_name>
<home_body> ::= "{" <home_export>* "}"
<home_export ::= <export>
             |   <factory_dcl> ";"
             |   <finder_dcl> ";"
<factory_dcl> ::= "factory" <identifier>
                  "(" [ <init_param_decls> ] ")"
                  [ <raises_expr> ]
<finder_dcl> ::= "finder" <identifier>
                 "(" [ <init_param_decls> ] ")"
```

```
                [ <raises_expr> ]
<event> ::= ( <event_dcl> | <event_abs_dcl> |
            <event_forward_dcl>)
<event_forward_dcl> ::= [ "abstract" ] "eventtype" <identifier>
<event_abs_dcl> ::= "abstract" "eventtype" <identifie
                    [ <value_inheritance_spec> ]
                    "{" <export>* "}"
<event_dcl> ::= <event_header> "{" <value_element> * "}"
<event_header> ::= [ "custom" ] "eventtype"
                   <identifier> [ <value_inheritance_spec> ]
```