FIWARE Advanced Middleware KIARA Documentation

Release 0.4.0

eProsima, DFKI, ZHAW

October 20, 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*

Please also check out our Github repository

CHAPTER 1

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)

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.

1.1.1 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`.

1.2 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.

1.3 Programmers guide

1.3.1 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.

1.3.2 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.

1.3.3 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

1.3.4 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”

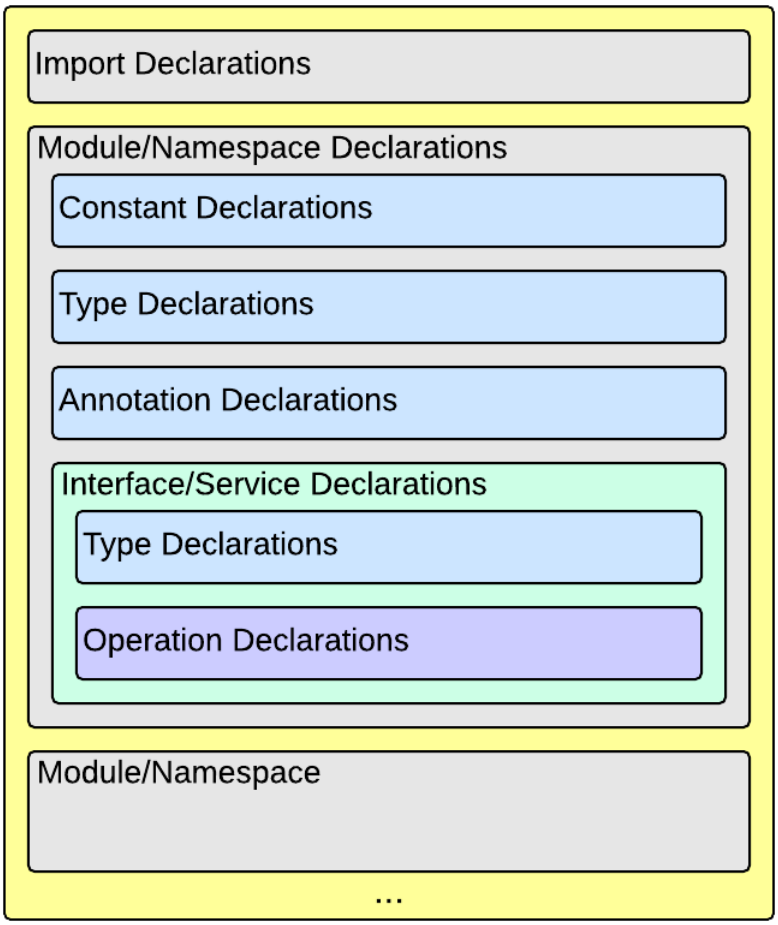

Fig. 1.1: IDL File Structure

    - **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.

1.3.5 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`.

1.3.6 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");
```

1.3.7 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");

    }

}
```

1.3.8 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.