Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Pasarela de comunicación entre MQTT y DDS

Autor: Manuel Alejandro Valenzuela Acosta

Tutor: Isabel Román Martínez

Dep. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Proyecto Fin de Carrera

Ingeniería de Telecomunicación

Pasarela de comunicación entre MQTT y DDS

Autor:

Manuel Alejandro Valenzuela Acosta

Tutor:

Isabel Román Martínez

Profesora Colaboradora

Dep. Ingeniería Telemática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2015

Proyecto Fin de Carrera: Pasarela de comunicación entre MQTT y DDS

Autor: Manuel Alejandro Valenzuela Acosta
Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Índice Abreviado

Índice Abreviado I

Índice de figuras VIII

Índice de tablas XI

Índice de códigos XIII

Índice de pruebas XV

Agradecimientos **XIX**

Resumen **XXI**

Abstract **XXIII**

1 Motivación y objetivos **1**

2 Estado del arte **3**

2.1 Soluciones de publicación-suscripción con *broker* centralizado 3

2.2 Soluciones de publicación-suscripción distribuidas 11

2.3 Soluciones fuera del patrón publicación-suscripción 15

2.4 Serialización 16

2.5 Almacenamiento 17

2.6 Conceptos de diseño 18

2.7 Otros elementos a considerar 23

3 Desarrollo 25

3.1 Programa principal 25

3.2 Cliente REST (Kabuki) 164

4 Ampliaciones 207

5 Conclusiones 209

Guía de instalación 211

1 Programa principal 211

2 Aplicación Ruby 212

Bibliografía 213

Índice

Índice Abreviado I

Índice de figuras VIII
Índice de tablas XI
Índice de códigos XIII
Índice de pruebas XV

Agradecimientos **XIX**

Resumen **XXI**

Abstract **XXIII**

1 Motivación y objetivos **1**

2 Estado del arte **3**
2.1 Soluciones de publicación-suscripción con *broker* centralizado 3
2.1.1 MQTT 4
2.1.1.1 Protocolo 5
2.1.1.2 Implementaciones 10
2.2 Soluciones de publicación-suscripción distribuidas 11
2.2.1 DDS 11
2.2.1.1 Arquitectura 11
2.2.1.2 *Built-in topics* 13
2.2.1.3 Implementaciones 13
2.3 Soluciones fuera del patrón publicación-suscripción 15
2.4 Serialización 16
2.5 Almacenamiento 17
2.6 Conceptos de diseño 18
2.6.1 Inversión de control 18

2.6.2 Principio de inversión de dependencia 19
2.6.3 Inyección de dependencias 19
2.6.3.1 Guice 20
2.7 Otros elementos a considerar 23

3 Desarrollo 25
3.1 Programa principal 25
3.1.1 Estructura 27
3.1.2 Criterios de diseño 30
3.1.3 Paquete Gateway 31
3.1.3.1 MqttDDSGateway 32
3.1.3.2 EntryPoint 33
3.1.3.3 GeneralModule 35
3.1.3.4 PropertyReader 38
3.1.4 Paquete DDS 40
3.1.4.1 ApplicationParticipantFactory 41
3.1.4.2 DDSListener 41
3.1.4.3 DDSModule 43
3.1.4.4 DDSPublisher 45
3.1.4.5 DDSSubscriber 47
3.1.4.6 DomainContainer 49
3.1.4.7 DomainContainerProvider 50
3.1.4.8 DomainParticipantProvider 50
3.1.4.9 DynamicDataSerializerProvider 52
3.1.4.10 DynamicDataTypeRegistry 53
3.1.4.11 DynamicTopicBuilder 54
3.1.4.12 DynamicTopicProcessor 56
3.1.4.13 TypeResolver 58
3.1.5 Ingreso de datos para DDS 62
3.1.5.1 DynamicDataPopulator 63
3.1.5.2 DynamicDataPopulatorFactory 65
3.1.6 Ingreso de datos en *arrays* 67
3.1.6.1 DynamicDataArrayPopulator 67
3.1.6.2 DynamicDataArrayPopulatorInterface 69
3.1.6.3 DynamicDataArrayPopulatorFactory 70
3.1.6.4 DynamicDataBooleanArrayPopulator 71
3.1.6.5 DynamicDataByteArrayPopulator 72
3.1.6.6 DynamicDataCharacterArrayPopulator 73
3.1.6.7 DynamicDataComplexArrayPopulator 74
3.1.6.8 DynamicDataDoubleArrayPopulator 76
3.1.6.9 DynamicDataFloatArrayPopulator 77

3.1.6.10 DynamicDataIntegerArrayPopulator 78
3.1.6.11 DynamicDataShortArrayPopulator 79
3.1.6.12 DynamicDataStringArrayPopulator 80
3.1.7 Ingreso de datos en miembros simples 83
3.1.7.1 DynamicDataSimplePopulatorInterface 83
3.1.7.2 DynamicDataSimplePopulatorFactory 84
3.1.7.3 DynamicDataBooleanMemberPopulator 86
3.1.7.4 DynamicDataByteMemberPopulator 86
3.1.7.5 DynamicDataCharacterMemberPopulator 87
3.1.7.6 DynamicDataDoubleMemberPopulator 88
3.1.7.7 DynamicDataFloatMemberPopulator 89
3.1.7.8 DynamicDataIntegerMemberPopulator 90
3.1.7.9 DynamicDataShortMemberPopulator 91
3.1.7.10 DynamicDataStringMemberPopulator 92
3.1.8 Procesamiento DDS 93
3.1.8.1 DynamicDataProcessor 94
3.1.8.2 DynamicDataMemberInfoExtractor 95
3.1.9 Procesamiento de miembros complejos 96
3.1.9.1 DynamicDataComplexProcessorInterface<T> 96
3.1.9.2 DynamicDataComplexProcessorFactory 97
3.1.9.3 DynamicDataStructProcessor 98
3.1.9.4 DynamicDataArrayProcessor 100
3.1.9.5 DynamicDataComplexArrayMemberProcessor 102
3.1.10 Procesamiento de miembros simples 104
3.1.10.1 DynamicDataSimpleProcessorInterface<T> 104
3.1.10.2 DynamicDataSimpleProcessorFactory 105
3.1.10.3 DynamicDataBooleanMemberProcessor 107
3.1.10.4 DynamicDataByteMemberProcessor 107
3.1.10.5 DynamicDataCharacterMemberProcessor 108
3.1.10.6 DynamicDataDoubleMemberProcessor 109
3.1.10.7 DynamicDataFloatMemberProcessor 110
3.1.10.8 DynamicDataIntegerMemberProcessor 111
3.1.10.9 DynamicDataShortMemberProcessor 111
3.1.10.10 DynamicDataStringMemberProcessor 112
3.1.11 Serialización 114
3.1.11.1 DynamicDataSerializer 114
3.1.11.2 IDLSerializer 115
3.1.12 *Logging* 117
3.1.12.1 ApplicationLoggerFactory 117
3.1.13 Paquete MQTT 118

3.1.13.1 MqttClientProvider 119
3.1.13.2 MqttClientProxy 120
3.1.13.3 MqttClientProxyProvider 122
3.1.13.4 MqttMessageBuilder 123
3.1.13.5 MqttMessageDeserializer 123
3.1.13.6 MqttMessageListener 125
3.1.13.7 MqttMessageListenerProvider 127
3.1.13.8 MqttModule 128
3.1.13.9 MqttPublication 129
3.1.13.10 MqttPublisher 130
3.1.13.11 MqttSubscriber 131
3.1.14 Paquete de persistencia 133
3.1.14.1 JedisPoolProvider 133
3.1.14.2 RedisModule 134
3.1.14.3 TypeCodesRedis 135
3.1.14.4 TypeCodesRedisProvider 137
3.1.14.5 TypeCodesSerializer 138
3.1.14.6 TypeCodesStore 139
3.1.15 *Tests* 142
3.1.15.1 TestSuite 142
3.1.15.2 EntryPointTest 142
3.1.15.3 DDSPublisherTest 142
3.1.15.4 DynamicTopicBuilderTest 143
3.1.15.5 DynamicTopicProcessorTest 143
3.1.15.6 DynamicDataPopulatorTest 144
3.1.15.7 DynamicDataProcessorTest 146
3.1.15.8 DynamicDataSimpleMembersProcessorTest 149
3.1.15.9 DynamicDataStructProcessorTest 149
3.1.15.10 DynamicDataArrayProcessorTest 150
3.1.15.11 DynamicDataSerializerTest 150
3.1.15.12 IDLSerializerTest 150
3.1.15.13 MqttClientProxyTest 151
3.1.15.14 MqttMessageBuilderTest 151
3.1.15.15 MqttMessageDeserializerTest 151
3.1.15.16 MqttMessageListenerTest 151
3.1.15.17 MqttPublicationTest 152
3.1.15.18 MqttPublisherTest 152
3.1.15.19 MqttSubscriberTest 152
3.1.15.20 TypeCodesRedisTest 153
3.1.15.21 TypeCodesSerializerTest 153

3.1.15.22 TypeCodesStoreTest 154
3.1.16 Librerías 155
3.1.16.1 Gson 157
3.1.16.2 Guava 157
3.1.16.3 Guice 158
3.1.16.4 Jedis 159
3.1.16.5 JUnit 159
3.1.16.6 Log4j 160
3.1.16.7 Mockito 161
3.1.16.8 Paho MQTT 162
3.1.16.9 RTI Connext DDS 163
3.2 Cliente REST (Kabuki) 164
3.2.1 Estructura 165
3.2.2 Fichero de configuración de Rack 167
3.2.3 Módulo Kabuki 167
3.2.4 App 168
3.2.5 AppTree 169
3.2.6 Cli 171
3.2.7 Config 172
3.2.8 Initializable 174
3.2.9 SchemaBuilder 175
3.2.10 Aplicaciones de Rack 176
3.2.10.1 Api 176
3.2.10.2 ExceptionHandler 178
3.2.10.3 HealthCheckFactory 179
3.2.10.4 HttpServerFactory 180
3.2.10.5 RackApps 180
3.2.11 Agentes MQTT 182
3.2.11.1 MqttAgents 182
3.2.11.2 MqttMessageProcessor 183
3.2.11.3 MqttPublisher 184
3.2.11.4 MqttSubscriber 185
3.2.11.5 MqttSubscriberThread 185
3.2.12 Persistencia 187
3.2.12.1 DataMapper 187
3.2.12.2 Message 188
3.2.12.3 SchemaStore 189
3.2.12.4 TopicStore 190
3.2.12.5 Stores 191
3.2.13 *Tests* 193

3.2.13.1 Api 193
3.2.13.2 App 195
3.2.13.3 Cli 195
3.2.13.4 Config 196
3.2.13.5 DataMapper 196
3.2.13.6 HttpServerFactory 196
3.2.13.7 MqttPublisher 197
3.2.13.8 MqttSubscriber 197
3.2.13.9 SchemaBuilder 197
3.2.13.10 SchemaStore 197
3.2.13.11 TopicStore 198
3.2.14 Gemas 199
3.2.14.1 Grape 199
3.2.14.2 Thor 200
3.2.14.3 Puma 200
3.2.14.4 MiniCheck 200
3.2.14.5 ActiveSupport 200
3.2.14.6 Rake 200
3.2.14.7 Mqtt 200
3.2.14.8 JsonSchemaGenerator 201
3.2.14.9 Redis 201
3.2.14.10 Mongoid 201
3.2.14.11 Hashie 201
3.2.14.12 Pry 201
3.2.14.13 RSpec 201
3.2.14.14 Rack/Test 201
3.2.14.15 Guard 202
3.2.15 *Front-end* 203
3.2.15.1 Página de publicación 203
3.2.15.2 Páginas de listado 204

4 Ampliaciones 207

5 Conclusiones 209

Guía de instalación 211

1 Programa principal 211
2 Aplicación Ruby 212

Bibliografía 213

Índice de Figuras

2.1	Diagrama de secuencia de publicación MQTT con QoS 0	8
2.2	Diagrama de secuencia de publicación MQTT con QoS 1	9
2.3	Diagrama de secuencia de publicación MQTT con QoS 2	10
2.4	Modelo de inyección de Guice	21
3.1	Ejemplo de un sistema con la aplicación desplegada	25
3.2	Estructura de paquetes de la aplicación Java	27
3.3	Diagrama UML del paquete Gateway	31
3.4	Diagrama UML de los componentes principales	35
3.5	Diagrama UML del paquete DDS	40
3.6	Diagrama UML del paquete de ingreso de datos DDS	62
3.7	Diagrama UML del paquete de Procesamiento DDS	93
3.8	Diagrama UML del paquete MQTT	118
3.9	Diagrama UML del paquete de persistencia	133
3.10	Estructura de directorios de la aplicación Ruby	165
3.11	Selector de la página de publicación	203
3.12	Editor de la página de publicación	204
3.13	Página de listado por tipo	205
3.14	Página de listado por cliente	205
3.15	Página de listado por *topic*	206

Índice de Tablas

2.1	Tipos de paquetes de control	6
2.2	Bits de banderas	7
3.1	*Bindings* de Guice en `GeneralModule`	37
3.2	*Bindings* de Guice en `DDSModule`	44
3.3	Pares clave-valor de `TypeResolver`	59
3.4	Métodos para ingresar datos en miembros de `DynamicData`	64
3.5	Clases que permite instanciar `DynamicDataArrayPopulatorFactory`	70
3.6	Clases que permite instanciar `DynamicDataSimplePopulatorFactory`	85
3.7	Clases que permite instanciar `DynamicDataComplexProcessorFactory`	97
3.8	Clases que permite instanciar `DynamicDataSimpleProcessorFactory`	106
3.9	Pares clave-valor del mensaje MQTT	115
3.10	*Bindings* de Guice en `MqttModule`	129
3.11	*Bindings* de Guice en `RedisModule`	135

Índice de Códigos

2.1	Interacción con el usuario sin IoC	18
2.2	Interacción con el usuario usando IoC	18
3.1	Estructura JSON de los mensajes MQTT	26
3.2	Listado de propiedades	38
3.3	Ejemplo con objeto Method	65
3.4	JSON de estructura	144
3.5	JSON de estructura anidada	144
3.6	JSON de estructura con miembros simples	145
3.7	JSON de estructura con *arrays*	145
3.8	JSON de estructura con *arrays* de cadenas	146
3.9	JSON de estructura con *arrays* de estructuras complejas	146
3.10	JSON de estructura	147
3.11	JSON de estructura anidada	147
3.12	JSON de estructura con miembros simples	147
3.13	JSON de estructura con *arrays*	148
3.14	JSON de estructura con *arrays* de cadenas	148
3.15	JSON de estructura con *arrays* de estructuras complejas	148
3.16	Contenido de `pom.xml`	155
3.17	Configuración de logging	160
3.18	Contenido de `Gemfile`	199

Índice de pruebas

1	JUnit Test (Ejecución de `Runnable`)	142
2	JUnit Test (Ejecución de `ddsSubscriber` y `mqttSubscriber`)	142
3	JUnit Test (Carácter asíncrono)	143
4	JUnit Test (Obtención del `Topic`)	143
5	JUnit Test (Obtención del `DataWriter`)	143
6	JUnit Test (Creación del objeto `DynamicData`)	143
7	JUnit Test (Obtención del `Topic`)	143
8	JUnit Test (Registra el `Topic`)	143
9	JUnit Test (Instancia un nuevo `Topic` si no se encuentra)	143
10	JUnit Test (Toma el tipo de memoria)	144
11	JUnit Test (Almacena el tipo si no se encuentra en memoria)	144
12	JUnit Test (Crea el `Topic`)	144
13	JUnit Test (Crea el `DataReader`)	144
14	JUnit Test (No crea el `Topic` si está registrado)	144
15	JUnit Test (No crea el `DataReader` si está registrado)	144
16	JUnit Test (Población de estructuras)	144
17	JUnit Test (Población de estructuras anidadas)	144
18	JUnit Test (Población de miembros simples)	145
19	JUnit Test (Población de miembros *arrays*)	145
20	JUnit Test (Población de *arrays* de `String`)	145
21	JUnit Test (Población de *arrays* de estructuras complejas)	146
22	JUnit Test (Procesamiento de estructuras)	146
23	JUnit Test (Procesamiento de estructuras anidadas)	147
24	JUnit Test (Procesamiento de miembros simples)	147
25	JUnit Test (Procesamiento de miembros *arrays*)	147
26	JUnit Test (Procesamiento de *arrays* de `String`)	148
27	JUnit Test (Procesamiento de *arrays* de estructuras complejas)	148

28 JUnit Test (Extracción de un booleano) 149
29 JUnit Test (Extracción de un byte) 149
30 JUnit Test (Extracción de un carácter) 149
31 JUnit Test (Extracción de un `double`) 149
32 JUnit Test (Extracción de un flotante) 149
33 JUnit Test (Extracción de un entero) 149
34 JUnit Test (Extracción de un `short`) 149
35 JUnit Test (Extracción de un `String`) 149
36 JUnit Test (Extracción de un miembro de la estructura) 149
37 JUnit Test (Extracción de varios miembros de la estructura) 149
38 JUnit Test (Extracción de miembros del array) 150
39 JUnit Test (Llamada al procesador) 150
40 JUnit Test (Serializa con identificador de cliente y tipo) 150
41 JUnit Test (Permite serializar un *map* en lugar de un `DynamicData`) 150
42 JUnit Test (Imprime IDL a `String`) 150
43 JUnit Test (Publica un mensaje en un *topic*) 151
44 JUnit Test (Permite desconectar el cliente) 151
45 JUnit Test (Permite agregar un *callback*) 151
46 JUnit Test (Permite suscribirse a un *topic*) 151
47 JUnit Test (Construye un mensaje MQTT correctamente) 151
48 JUnit Test (Construye un mensaje MQTT con QoS correctamente) 151
49 JUnit Test (Deserializa un `MqttMessage`) 151
50 JUnit Test (Deserializa un `String`) 151
51 JUnit Test (Ignora el mensaje si el identificador del cliente coincide con el propio) 152
52 JUnit Test (Busca el tipo en el almacén de `TypeCodes`) 152
53 JUnit Test (Toma los distintos campos del *map*) 152
54 JUnit Test (Publica el mensaje usando el cliente) 152
55 JUnit Test (Ejecuta una publicación) 152
56 JUnit Test (Se suscribe a todos los *topics* y escucha los mensajes) 153
57 JUnit Test (Toma y liberación de recursos en `fetch`) 153
58 JUnit Test (Toma y liberación de recursos en `store`) 153
59 JUnit Test (Almacenamiento correcto) 153
60 JUnit Test (Almacenamiento incorrecto) 153
61 JUnit Test (Extracción correcta (resultado)) 153
62 JUnit Test (Extracción correcta (llamada)) 153
63 JUnit Test (Extracción incorrecta) 153
64 JUnit Test (Comprueba la simetría de la serialización) 153
65 JUnit Test (Serialización correcta) 153
66 JUnit Test (Extracción de memoria (*miss*)) 154
67 JUnit Test (Extracción de memoria (*hit*)) 154

68	JUnit Test (Extracción de Redis (llamada))	154
69	JUnit Test (Extracción de Redis (resultado))	154
70	JUnit Test (Extracción de Redis (almacenamiento en memoria))	154
71	JUnit Test (Almacenamiento en memoria)	154
72	JUnit Test (Almacenamiento en Redis)	154
1	RSpec Test (Devuelve 200)	193
2	RSpec Test (Lista los esquemas)	193
3	RSpec Test (Devuelve 200 si el tipo existe)	193
4	RSpec Test (Devuelve el esquema)	193
5	RSpec Test (Devuelve 404 si el tipo no existe)	193
6	RSpec Test (Devuelve 201 si el tipo existe)	193
7	RSpec Test (Publica un mensaje)	193
8	RSpec Test (Devuelve 404 si el tipo no existe)	194
9	RSpec Test (Devuelve 200)	194
10	RSpec Test (Lista los tipos)	194
11	RSpec Test (Devuelve 200 cuando no hay mensajes)	194
12	RSpec Test (No devuelve mensajes)	194
13	RSpec Test (Devuelve 200 cuando hay mensajes)	194
14	RSpec Test (Devuelve los mensajes correctos filtrados por tipo y paginados)	194
15	RSpec Test (Devuelve 200)	194
16	RSpec Test (Lista los *topics*)	194
17	RSpec Test (Devuelve 200 cuando no hay mensajes)	194
18	RSpec Test (No devuelve mensajes)	194
19	RSpec Test (Devuelve 200 cuando hay mensajes)	194
20	RSpec Test (Devuelve los mensajes correctos filtrados por *topic* y paginados)	195
21	RSpec Test (Devuelve 200)	195
22	RSpec Test (Lista los clientes)	195
23	RSpec Test (Devuelve 200 cuando no hay mensajes)	195
24	RSpec Test (No devuelve mensajes)	195
25	RSpec Test (Devuelve 200 cuando hay mensajes)	195
26	RSpec Test (Devuelve los mensajes correctos filtrados por cliente y paginados)	195
27	RSpec Test (Ejecuta Puma)	195
28	RSpec Test (Llama `start!` sobre el subscriptor)	195
29	RSpec Test (Ejecuta los tests)	195
30	RSpec Test (Llama `start!` sobre la aplicación)	195
31	RSpec Test (Abre una consola REPL)	196
32	RSpec Test (Inicializa `app_tree`)	196
33	RSpec Test (Encapsula la configuración)	196
34	RSpec Test (Encapsula la configuración del servidor web)	196
35	RSpec Test (Encapsula la configuración de Redis)	196

36 RSpec Test (Encapsula la configuración de la base de datos) 196
37 RSpec Test (Encapsula la configuración del cliente MQTT) 196
38 RSpec Test (Carga la configuración de la base de datos) 196
39 RSpec Test (Crea los índices de la base de datos) 196
40 RSpec Test (Inicializa `model`) 196
41 RSpec Test (Permite persistir una entrada en la base de datos) 196
42 RSpec Test (Devuelve la entrada al persistirla) 196
43 RSpec Test (Delega `all` al modelo) 196
44 RSpec Test (Delega `delete_all` al modelo) 196
45 RSpec Test (Permite filtrar por tipo) 196
46 RSpec Test (Permite filtrar por *topic*) 196
47 RSpec Test (Permite filtrar por cliente) 196
48 RSpec Test (Permite listar tipos) 196
49 RSpec Test (Permite listar *topics*) 196
50 RSpec Test (Permite listar clientes) 196
51 RSpec Test (Devuelve 200 si el estado es correcto) 197
52 RSpec Test (Devuelve 500 si el estado es incorrecto) 197
53 RSpec Test (Publica un mensaje usando el cliente) 197
54 RSpec Test (Permite arrancar el hilo al llamar a `start!`) 197
55 RSpec Test (Inicializa `topic_store`) 197
56 RSpec Test (Construye el esquema esperado) 197
57 RSpec Test (Inicializa `redis`) 198
58 RSpec Test (Se llama a `hkeys` sobre el *hash* de Redis al listar) 198
59 RSpec Test (Se llama a `hget` sobre el *hash* de Redis al tomar un esquema) 198
60 RSpec Test (Se llama a `hset` si no hay esquema) 198
61 RSpec Test (No se llama a `hset` si hay esquema) 198
62 RSpec Test (Inicializa `redis`) 198
63 RSpec Test (Se llama a `smembers` sobre el *set* de Redis al listar) 198
64 RSpec Test (Se llama a `sadd` sobre el *set* de Redis al añadir) 198

Agradecimientos

En este apartado quiero dar las gracias a todos los que me han ayudado en este camino, ya sea durante la carrera o durante la realización del proyecto.

Para empezar, al departamento de telemática de la ETSI, por ayudarme a adquirir conocimientos durante los años que me han ayudado a realizar este proyecto, en especial a mi tutora, Isabel Román.

A mis compañeros de trabajo, por la ayuda a la hora de profundizar en conceptos de programación orientada a objetos e ingeniería del *software*. En especial a Bruno Bossola, Manuel Morales y Francesc Quiñones.

A Juan Manuel Vozmediano, ya que gracias a él tuve esa oportunidad.

A todos los compañeros tras tantos años en la escuela, y por supuesto a mi familia y amigos.

Y por último a todos aquellos a los que haya olvidado mencionar, porque serán muchos.

Resumen

Este proyecto tiene como finalidad el desarrollo de una pasarela de comunicación entre DDS[2] y MQTT[1], dos de los sistemas de mensajería utilizados en el área M2M (*machine-to-machine*). Son dos sistemas muy dispares, por lo que no toda la funcionalidad se puede traducir de uno a otro.

Más concretamente, el proyecto se centrará en diseñar un intermediario DDS/MQTT que permita crear sistemas híbridos con clientes heterogéneos, es decir, de un sistema u otro.

La aplicación principal está escrita en Java, utilizando RTI Connext[16] y Eclipse Paho[23].

Además, la solución incluye una aplicación que permitirá leer y escribir mensajes desde el lado MQTT: una aplicación de Rack[31] en Ruby que proporciona una API que sigue el paradigma REST para ser utilizada a través de una mínima interfaz gráfica en HTML y Javascript. Este servicio comenzó como una ayuda a la hora de desarrollar la aplicación principal y acabó siendo suficientemente maduro para incluirlo como parte del proyecto.

Abstract

The aim of this project is the development of a communication gateway between DDS[2] and MQTT[1], two of the messaging systems used in the M2M (machine-to-machine) paradigm. The two are rather dissimilar, and therefore not all of their functionality can be matched by the other.

More precisely, this project will focus on the design of an intermediary between DDS and MQTT which will allow the deployment of hybrid systems comprising heterogeneous clients. That is, for both systems.

The main application is written in Java, relying on RTI Connext[16] and Eclipse Paho[23].

On top of that, the presented solution includes another application that allows for reading and writing of messages from the MQTT side: a Rack[31] application written in Ruby that exposes a REST API. This API will be called through a minimal graphical interface written in HTML and Javascript. This service started as a development tool for the main application and ended up being mature enough to be bundled as part of the project.

1 Motivación y objetivos

DDS (Data Distribution Service) es un *middleware* estandarizado que implementa el patrón publicación-subscripción y permite la construcción de sistemas centrados en los datos. Uno de los rasgos característicos de DDS es que no depende de elementos centralizados, permitiendo que los sistemas construidos sean totalmente distribuidos. Esto es una ventaja a la hora de trabajar en ámbitos de red local (LAN), pudiendo comunicarse los participantes de manera sencilla. Sin embargo, en una red de área extensa se convierte en un problema, ya que cortafuegos, NAT y demás son obstáculos para su correcto funcionamiento. La motivación es, pues, facilitar el uso de DDS aun cuando las entidades participantes se encuentren en subredes diferentes y sea necesario que los mensajes atraviesen redes de área extensa.

Existen soluciones en el mercado diseñadas para aliviar este problema, como RTI DDS Routing Service[4]. Pero son complejas y propietarias, limitando la interoperabilidad con aplicaciones construidas sobre otras implementaciones.

La solución que se presenta en este documento está centrada en los datos. Es una pasarela que permite que se use MQTT como protocolo en la red extensa. Los sistemas MQTT también siguen el paradigma publicación-suscripción, pero en estos sistemas hay uno o varios elementos encargados de reenviar los mensajes publicados hacia los participantes que hayan mostrado interés en ellos. Estos elementos se suelen llamar *brokers*.

Esto permitirá a los publicadores y suscriptores en ambos sistemas comunicarse dentro de unos casos de uso concretos, dando una mayor flexibilidad.

El reto clave consiste en interpretar la estructura arbitraria de los mensajes provinientes de ambos lados de la comunicación y retransmitirlos por la otra interfaz de forma que se permita comunicación entre participantes DDS con intermediarios MQTT o comunicación entre participantes DDS y clientes MQTT, siempre que los clientes estén preparados para entender el formato de los mensajes enviados.

Ejemplo de este segundo caso es el cliente REST incluido como parte del proyecto, que permite publicar mensajes de los tipos existentes en el dominio hasta el momento.

2 Estado del arte

En este apartado se describen las principales tecnologías de interés en el desarrollo de este proyecto.

2.1 Soluciones de publicación-suscripción con *broker* centralizado

MQTT (*Message Queue Telemetry Transport*) es una solución de mensajería simple y ligera, aunque solamente garantiza una interoperabilidad parcial entre publicadores y subscriptores. Se pueden intercambiar mensajes entre diferentes implementaciones de MQTT, pero ambos extremos de la comunicación deben conocer el formato del cuerpo. Está diseñado para ser fácil de implementar. Esto lo hace ideal para muchas situaciones, incluyendo escenarios con muchas limitaciones, especialmente de ancho de banda.

AMQP (*Advanced Message Queueing Protocol*) es un protocolo orientado a mensajes. Es un protocolo muy opaco, es decir, especificado de forma rígida, lo que hace que las distintas implementaciones puedan interoperar completamente.

Cualquiera de estas opciones necesita de un elemento encargado de enviar los mensajes a los destinatarios que hayan mostrado interés. A este elemento se le suele llamar *broker*. Si usásemos uno de estos protocolos para un caso en el que cierto dispositivo tiene que enviar mensajes a quizá miles de dispositivos, necesitaríamos una arquitectura más compleja, con múltiples *brokers*, para compensar la pérdida de rendimiento.

Otra desventaja de estos sistemas con algún elemento centralizado es que en el momento en que el *broker* cambia de dirección, podría ser necesario reconfigurar los clientes.

Para este proyecto, es de especial interés extenderse en la explicación sobre MQTT:

2.1.1 MQTT

MQTT nació en 1999, de manos del Dr. Andy Stanford-Clark, trabajador de IBM, y Arlen Nipper, de Arcom (ahora Eurotech). Desde entonces, se ha usado en una gran variedad de proyectos, incluyendo monitorización de datos enviados por sensores, sistemas de bases de datos distribuidas (Gaian DB) y aplicaciones móviles tan extendidas como Facebook Messenger.

Es útil para conexiones a distancia, donde la mínima sobrecarga es un requisito y el ancho de banda es caro. Por ejemplo, se ha usado en sensores que se comunican con un *broker* a través de un enlace satélite, conexiones por red telefónica en sistemas de salud y en sistemas domóticos. También es ideal para aplicaciones móviles, donde el ancho de banda y el consumo energético son factores clave.

En octubre de 2014, OASIS, *Organization for the Advancement of Structured Information Standards*, publicó la primera especificación del estándar MQTT 3.1.1[1].

El puerto 1883 está reservado para el uso de MQTT, y el 8883 para MQTT sobre SSL. Es importante destacar que SSL no es un protocolo ligero, así que añade una sobrecarga significativa. Además, el protocolo MQTT soporta autenticación con usuario y contraseña.

En un sistema MQTT, podemos distinguir varios conceptos importantes, que detallaremos a continuación.

Los clientes pueden:

- Publicar mensajes de aplicación que puedan interesar a otros clientes.
- Suscribirse a mensajes de aplicación que les puedan interesar.
- Revocar suscripciones para dejar de recibir ciertos mensajes de aplicación.
- Desconectarse del *broker*.

El *broker* se encarga de:

- Aceptar conexiones de clientes.
- Aceptar mensajes de aplicación enviados por clientes.
- Procesar mensajes de suscripción y revocación de parte de clientes.
- Notificar los mensajes de aplicación que cumplan los criterios de suscripción de los clientes.

Un *topic* es una etiqueta con la que se marcan los mensajes de aplicación, que luego se coteja con las suscripciones que conoce el *broker* para reenviar el mensaje de aplicación a aquellos clientes cuyas suscripciones incluyan el *topic* en cuestión.

Una sesión es una interacción entre cliente y servidor, que puede durar tanto como la conexión de red o puede prolongarse durante más de una.

Una suscripción se compone de filtros de *topic* y una QoS máxima. Cada suscripción pertenece a una única sesión, pero una sesión puede tener más de una suscripción. Cada suscripción dentro de una sesión debe tener un filtro de *topic* diferente.

Un filtro de *topic* es una expresión incluida en una suscripción, que indica el interés en uno o más *topics*. Puede tener comodines (+ o #, correspondientes a uno o varios niveles de la jerarquía, respectivamente).

En el resto de la explicación, "cliente" se refiere a un publicador o suscriptor y "servidor" a un *broker*, para mantener la misma terminología que utiliza el borrador de la norma[1].

2.1.1.1 Protocolo

Un paquete de control MQTT se compone de una parte fija de la cabecera, una parte variable que solamente está presente en algunos tipos de paquetes y la carga.

La parte fija de la cabecera se compone de 4 bits que indican el tipo de paquete, que toma uno de los siguientes valores:

Tabla 2.1 Tipos de paquetes de control.

Nombre	Valor	Dirección	Descripción
Reservado	0	Prohibido	Reservado
CONNECT	1	Cliente a servidor	Petición de conexión del cliente al servidor
CONNACK	2	Servidor a cliente	Asentimiento de la conexión
PUBLISH	3	Ambos	Mensaje de publicación
PUBACK	4	Ambos	Asentimiento de la publicación
PUBREC	5	Ambos	Recepción de la publicación (Envío fiable parte 1)
PUBREL	6	Ambos	Liberación de la publicación (Envío fiable parte 2)
PUBCOMP	7	Ambos	Publicación completa (Envío fiable parte 3)
SUBSCRIBE	8	Cliente a servidor	Petición de suscripción del cliente
SUBACK	9	Servidor a cliente	Asentimiento de suscripción
UNSUBSCRIBE	10	Cliente a servidor	Petición de desuscripción
UNSUBACK	11	Servidor a cliente	Asentimiento de desuscripción
PINGREQ	12	Cliente a servidor	Petición de ping
PINGRESP	13	Servidor a cliente	Respuesta de ping
DISCONNECT	14	Cliente a servidor	Desconexión del cliente
Reservado	15	Prohibido	Reservado

La cabecera fija también contiene 4 bits de banderas, cuyo significado depende del tipo de paquete.

Tabla 2.2 Bits de banderas.

Paquete de control	**Uso**	Bit 3	Bit 2	Bit 1	Bit 0
CONNECT	Reservado	0	0	0	0
CONNACK	Reservado	0	0	0	0
PUBLISH	Usado en MQTT 3.1.1	DUP	QoS	QoS	RETAIN
PUBACK	Reservado	0	0	0	0
PUBREC	Reservado	0	0	0	0
PUBREL	Reservado	0	0	1	0
PUBCOMP	Reservado	0	0	0	0
SUBSCRIBE	Reservado	0	0	1	0
SUBACK	Reservado	0	0	0	0
UNSUBSCRIBE	Reservado	0	0	1	0
UNSUBACK	Reservado	0	0	0	0
PINGREQ	Reservado	0	0	0	0
PINGRESP	Reservado	0	0	0	0
DISCONNECT	Reservado	0	0	0	0

Luego, contiene de uno a cuatro octetos dedicados a codificar la longitud restante del paquete, que se codifican con 7 bits de datos, reservando el más significativo para indicar que hay más octetos en la representación. Así, por ejemplo, 64 se codifica como 0x40 y 1500 como 0x67 0x0B.

Por último puede haber una parte opcional de la cabecera que se encuentra entre la parte fija y el cuerpo del mensaje. Esta parte variable depende del tipo de paquete. Un campo común es el identificador del paquete.

Nombres de *topic*

Un nombre de *topic* es una cadena de texto en la que hay un separador definido (`'/'`) y dos comodines: multinivel (`'#'`) y de un solo nivel (`'+'`).

Un nombre de *topic* que empieza por `'$'` indica un mensaje específico del sistema, los clientes no deberían usarlos para comunicarse entre ellos. Se suele utilizar `'$SYS'` como prefijo para este tipo de topics.

Identificador de paquete

El identificador de paquete tiene un significado local. Es decir, entre un publicador y el *broker* el identificador del paquete no es necesariamente el mismo que aquel asociado al mismo mensaje cuando el *broker* lo envía a los suscriptores interesados.

Los identificadores se generan de forma única para una conexión punto a punto y en un sentido. Así, es posible que entre un cliente y un *broker* se intercambien mensajes PUBLISH con el mismo identificador en sentidos contrarios.

Calidad de servicio

En MQTT, el protocolo de envío funciona de manera similar entre un publicador y un *broker* y entre el broker y un suscriptor.

En las figuras, los mensajes PUBLISH se muestran acompañados de tres números. El primer número corresponde al nivel de QoS, el segundo a la bandera de duplicado, el tercero representa el identificador del paquete. Aunque haya identificadores distintos, hay casos en los que aleatoriamente podrían haberse generado iguales.

MQTT provee tres niveles de QoS. Los tres niveles son los siguientes:

- QoS 0: Como mucho una vez.

  El mensaje PUBLISH se envía según las características de la red que hay por debajo. El receptor no envía ningún tipo de asentimiento ni se hace ningún reenvío. El mensaje llega al *broker* una vez o ninguna, y si llega, llegará a cada uno de los suscriptores una vez o ninguna.

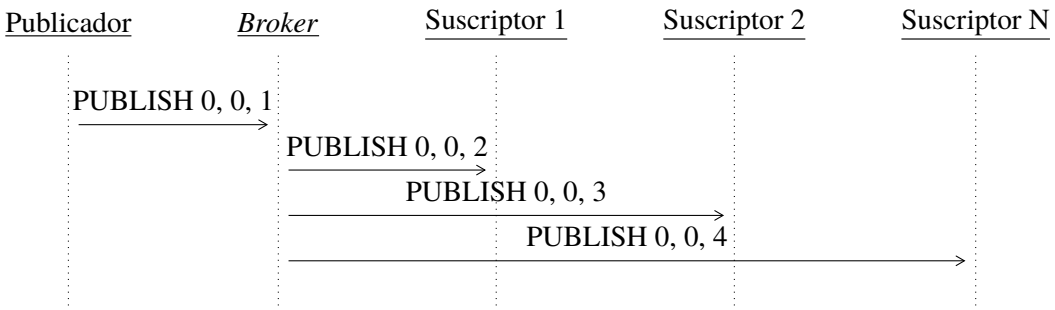

Figura 2.1 Diagrama de secuencia de publicación MQTT con QoS 0.

- QoS 1: Al menos una vez.

  Este nivel de calidad de servicio se asegura de que el mensaje llega al receptor al menos una vez. Un paquete PUBLISH con QoS 1 tiene un identificador de paquete en la cabecera y se debe asentir con un paquete PUBACK.

  El lado que envía debe asignar un identificador de paquete único cada vez que publica un nuevo mensaje de aplicación. Hasta que se reciba el paquete PUBACK del receptor el paquete se trata como “sin asentir”.

Una vez que se envía el PUBACK el receptor debe tratar cualquier otra publicación con el mismo identificador como una nueva publicación, independientemente del valor de la bandera DUP.

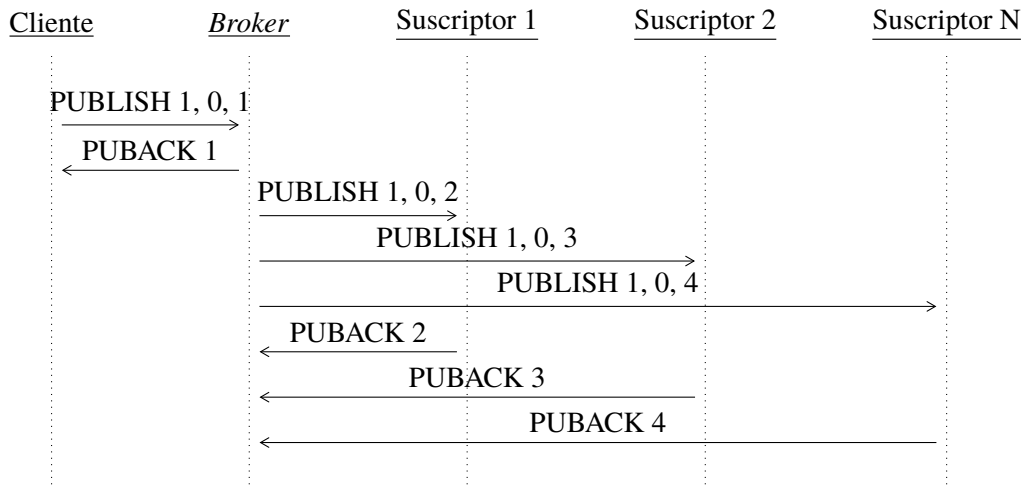

Figura 2.2 Diagrama de secuencia de publicación MQTT con QoS 1.

- QoS 2: Exactamente una vez.

Es la mejor calidad de servicio, cuando no se quieren pérdidas ni duplicación.

En el nivel 2 de QoS, el lado emisor debe asignar también un identificador de paquete único, tratando como “sin asentir” un paquete si no ha recibido un paquete PUBREC con el mismo identificador desde el receptor.

También debe enviar un paquete PUBREL que contenga el mismo identificador de paquete una vez recibe el PUBREC, tratando a este como “sin asentir” hasta la recepción del paquete PUBCOMP correspondiente (con el mismo identificador) del lado receptor.

Una vez recibe el paquete PUBREL no debe reenviar el paquete PUBLISH original.

Por otro lado, el lado receptor debe asentir cada paquete PUBLISH con el mismo identificador con un paquete PUBREC hasta que recibe el paquete PUBREL. Una vez envía el paquete PUBCOMP, el receptor debe tratar cualquier paquete PUBLISH con el mismo identificador como una nueva publicación.

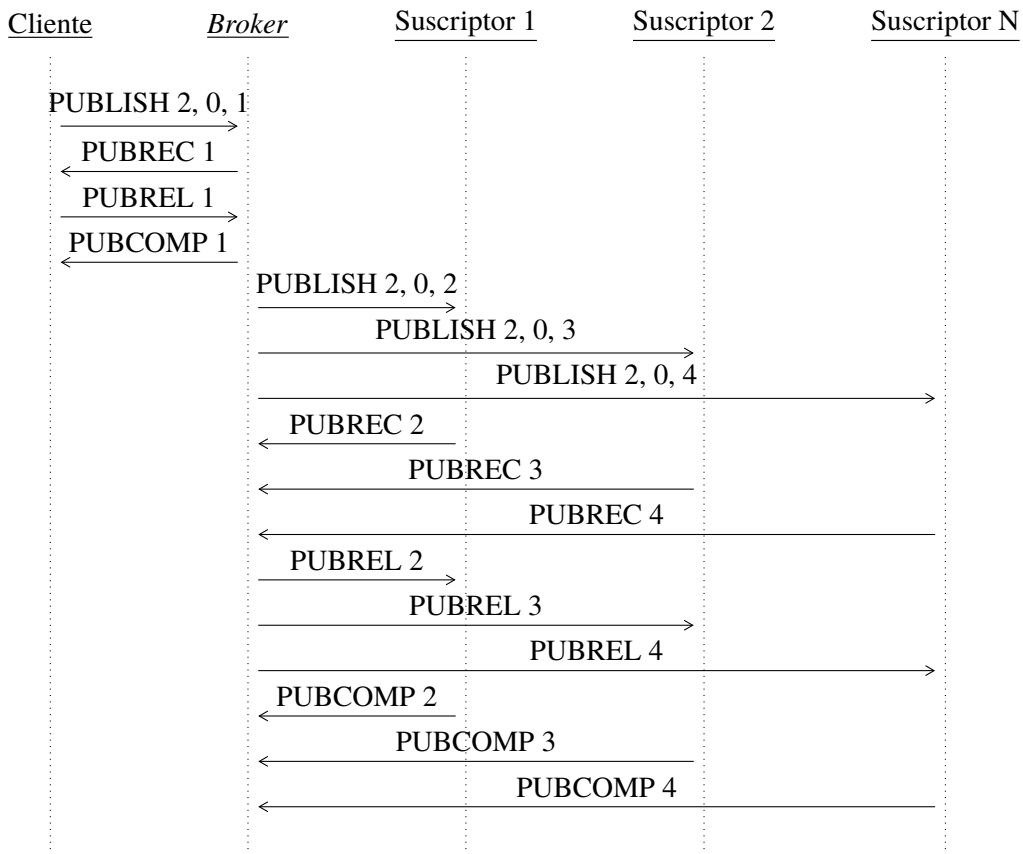

Figura 2.3 Diagrama de secuencia de publicación MQTT con QoS 2.

2.1.1.2 Implementaciones

Brokers

- IBM Websphere MQ Telemetry
- Mosquitto
- RabbitMQ (MQTT plugin)
- eMQTT

Clientes

- Eclipse Paho
- Mosquitto, en muchos lenguajes

2.2 Soluciones de publicación-suscripción distribuidas

Por otro lado, DDS (*Data Distribution Service*) es un sistema más distribuido y centrado en los datos, orientado a comunicación en tiempo real entre dispositivos.

2.2.1 DDS

DDS es un *middleware* para comunicación M2M (*machine-to-machine*) estandarizado por el OMG (*Object Management Group*).

DDS define un paradigma publicación/suscripción, en el que los publicadores crean *topics* y publican *samples*. DDS se encarga de enviar los *samples* a los destinatarios que hayan mostrado interés en cada *topic*.

La principal ventaja de utilizar un *middleware* como DDS es que ahorra al programador tratar con el problema de la serialización y deserialización, enrutamiento y envío de mensajes, control de flujo o reenvíos. Además, ya que implementa el paradigma publicación/subscripción, proporciona desacoplo temporal y espacial entre los participantes.

La especificación describe dos niveles de interfaces:

- Una capa baja, **DCPS** (*Data Centric Publish-Subscribe*), que se centra en envío eficiente de la información a sus respectivos receptores.

- Una capa alta, opcional, **DLRL** (*Data Local Reconstruction Layer*), que ayuda a integrar DDS con la capa de aplicación de forma simple.

DDS se utiliza en aplicaciones con *Big Data*, *trading* y control aéreo, por ejemplo. También es útil en comunicaciones móviles y en entornos sanitarios.

2.2.1.1 Arquitectura

Las entidades más importantes en DDS son las siguientes:

- `DomainParticipant`: Es el elemento principal de la comunicación con un dominio concreto. Representa la participación de la aplicación en el dominio y es una *factory* de instancias de `Publisher`, `Subscriber` y `Topic`, entre otros. Actúa también como contenedor de `Entities`.

- `DomainParticipantFactory`: Una *singleton*[1] *factory*[2], es el principal punto de inicio de las aplicaciones. Se encarga de instanciar los `DomainParticipant`.

  Al ser un *singleton*, se consigue la instancia mediante el método `get_instance()`. Una aplicación puede participar en más de un dominio instanciando múltiples `DomainParticipant`. También se pueden instanciar varios en el mismo dominio, pero hay que tener en cuenta que intercambian datos de la misma forma estén en la misma aplicación o en distintas.

- `TopicDescription`: Es la superclase de `Topic`, `MultiTopic` y `ContentFilteredTopic`.

- `Topic`: La descripción más básica de datos que pueden ser publicados y notificados. Se identifica por su nombre, que debe ser único en el dominio.

- `ContentFilteredTopic`: Es otra especialización de `TopicDescription` que permite subscribirse en función del contenido. Es una subscripción más sofisticada que la de `Topic`, que indica que un `DataReader` no quiere ver todos los valores que se publican bajo un `Topic`, sino solamente los que cumplen con ciertos criterios. Para ello emplea una `filter_expression` y unos `expression_parameters`, que describen dicho filtro sobre el contenido del `Topic`.

- `MultiTopic`: Es una especialización de `TopicDescription` que permite subscribirse a una combinación de datos que vienen de distintos *topics*. Permite combinar datos de los distintos *topics* en un único tipo. Los datos serán filtrados y reordenados según una `subscription_expression` y unos `expression_parameters`.

- `Publisher`: Es el objeto responsable de publicar mensajes. Actúa junto a uno o varios `DataWriter` que le pertenecen. Decide cuándo debe enviar el mensaje teniendo en cuenta más información, como el tiempo y su QoS.

- `DataWriter`: Permite que la aplicación escriba los valores de los datos que se publican bajo cierto `Topic`.

- `Subscriber`: Es el objeto responsable de recibir datos relacionados con una subscripción. Actúa junto a uno o varios `DataReader` que indican a la aplicación que hay datos disponibles mediante un *listener*.

- `DataReader`: Permite que la aplicación declare los datos que quiere recibir (subscripción) y acceder a los datos recibidos por el `Subscriber` al que está asociado.

[1] *Singleton* es un patrón de diseño orientado a objetos que previene la creación de más de una instancia de una clase. Esto puede ser de utilidad cuando es importante que haya únicamente un objeto para coordinar el resto del sistema. Su uso se debe limitar a casos muy concretos, como aquellos en los que esta característica sea crítica.[5]

[2] *Factory* es un objeto cuya responsabilidad es crear otros objetos, una especie de abstracción o capa de indirección sobre el constructor de una clase.

- `DynamicData`: Propuesta como parte de Extensible and Dynamic Topic Types[3], es una API que permite definir y recibir tipos de forma dinámica. Un objeto DynamicData define una muestra de cualquier tipo complejo, que puede ser inspeccionada y manipulada con reflexión.

- `TypeCode`: Es la definición de un tipo concreto de datos, permite inspeccionar el nombre, miembros y otras propiedades de dichos tipos. Se pueden crear usando la `TypeCodeFactory`. Está basada en una clase similar de CORBA.

2.2.1.2 *Built-in topics*

Además de todas estas entidades, DDS provee distintos topics predefinidos (*built-in topics*):

- **Participant Built-in Topic**: Permite acceder a información sobre los `Domain Participant` descubiertos.

- **Publication Built-in Topic**: Permite acceder a información sobre las publicaciones descubiertas.

- **Subscription Built-in Topic**: Permite acceder a información sobre las subscripciones descubiertas.

- **Topic Built-in Topic**: Permite acceder a información sobre los *topics* descubiertos.

2.2.1.3 Implementaciones

Hay tres implementaciones destacadas de DDS.

- **RTI Connext**: Implementación comercial disponible en C, C++, .NET y Java. De Real Time Innovations.

- **OpenSplice DDS**: Implementación comercial de código abierto. También disponible para usar con C, C++, C# y Java.

- **OpenDDS**: Implementación de código abierto de DDS en C++ que también incluye *bindings* para Java.

De estas implementaciones para el propósito de este proyecto la más apropiada es RTI, a pesar de ser la más cerrada de las tres. Como queremos una pasarela lo más transparente posible, poder usar Java (que permite reflexión[3], a diferencia de C++) y soporte de la API `DynamicData` definida

[3] La reflexión es la capacidad de un lenguaje para poder examinar y modificar la estructura y el comportamiento de un programa en tiempo de ejecución. En el contexto de este proyecto es importante poder conocer los tipos en tiempo de ejecución, ya que no se conocerá a priori qué tipos serán enviados. Con Java hay que tener la precaución de que en tiempo de ejecución, cuando se trata con genéricos, el código compilado utiliza `java.lang.Object` en lugar del

en la especificación de X-Types de DDS [3], ni OpenSplice ni OpenDDS permiten hacerlo de forma tan sencilla.

tipo en cuestión. Por lo tanto, `ArrayList<Integer>` y `ArrayList<String>` son idénticas en tiempo de ejecución, ya que la información adicional sobre los tipos ha sido borrada por el compilador.

2.3 Soluciones fuera del patrón publicación-suscripción

REST (*Representational State Transfer*) es un estilo de comunicación basado en peticiones y respuestas. Es simple de implementar pero no permite una comunicación de publicación-subscripción asíncrona. Además, el hecho de ser un sistema sin estado como HTTP hace que en cada petición haya que incluir información adicional que puede hacer el procesado muy ineficiente.

Es muy utilizado en aplicaciones web. La aplicación Ruby parte del proyecto es un cliente REST que permite publicar mensajes a través de MQTT.

2.4 Serialización

Hay otro tema a tratar antes de poder tener una idea global del estado del arte, que es el tema de serialización.

Serializar consiste en tomar un objeto y representarlo de forma que se pueda transmitir y permitiendo ser convertido de vuelta a un objeto en el extremo opuesto de la comunicación.

Existen muchos formatos de representación de datos. Entre ellos:

- **XML**: Lenguaje de marcas extensible, que define normas para codificar de forma que es entendible por máquinas y humanos al mismo tiempo. La principal desventaja es que es demasiado verboso.

- **JSON**: Notación de objetos de Javascript. Es la forma en que los objetos se representan en el lenguaje Javascript. Un objeto es un conjunto de propiedades definidas por una clave y un valor. Es un formato muy compacto y extendido. Tiene la ventaja de tener una gran cantidad de librerías ligeras y bien mantenidas que permiten trabajar con él en muchos lenguajes.

- **CDR**: Representación común de datos. Es un formato para encapsular datos. Describe representaciones para todos los tipos de datos de IDL[15].

En el contexto de este proyecto conviene utilizar distintos formatos en distintas partes del programa. Por ejemplo, para permitir interoperabilidad con clientes MQTT sin que estos tengan que utilizar los tipos de datos de IDL, o quizá ni siquiera estar implementados en el mismo lenguaje, conviene utilizar algo como JSON, ligero y ampliamente soportado.

Sin embargo, para serializar tipos que se usen en el lado DDS conviene utilizar CDR, ya que ha sido específicamente diseñado para serializar tipos IDL.

2.5 Almacenamiento

Para mantener un conocimiento común de las definiciones de tipos existentes (instancias de `TypeCode`) en el sistema conviene compartir dichas definiciones (serializadas) mediante una base de datos o almacén de algún tipo. Para cada caso concreto existe una solución más indicada que el resto de opciones posibles y uno de los pasos claves a la hora de diseñar una arquitectura es encontrar esa solución[14].

- **Base de datos relacional**: El tipo de base de datos más extendido, pero el carácter relacional en este caso no nos ofrece ninguna ventaja, ya que lo que queremos guardar es simplemente una definición.

- **Base de datos NoSQL**: Se podría utilizar alguna base de datos basada en documentos como MongoDB, en grafos como OrientDB o Postgres como NoSQL, pero estos sistemas son complejos para muchos de los casos. Sin embargo, sistemas que necesiten almacenar JSON, o, en general, objetos con un esquema arbitrario, opciones como MongoDB resultan muy útiles.

- **Almacén clave-valor**: Este tipo de almacenes permiten guardar pares clave-valor, funcionalidad que se ajusta bien a lo que se necesita. Se escoge Redis para este propósito, por su facilidad para persistir distintos tipos de datos, incluidos *hashes*.

Para almacenar las definiciones de tipos se utilizará Redis, que es una *cache*/almacén de clave-valor que permite compartirlas de forma sencilla y rápida, confinando toda la información a un *Hash*, en el que podemos comprobar si el tipo está definido ya o no, entre otras operaciones.

2.6 Conceptos de diseño

Para entender el funcionamiento del código que compone este proyecto es necesario entender varios conceptos teóricos relacionados, siendo el primero de estos el de inversión de control (normalmente abreviado como IoC, de *inversion of control*).

Este apartado pretende ser una introducción a la inyección de dependencias, una forma de desarrollo utilizada a lo largo del proyecto. Esta forma de desarrollar condiciona el diseño de la aplicación, así que se hace necesario entenderla antes de enfrentarse al código.

2.6.1 Inversión de control

La primera mención conocida del término aparece en 1988, en un artículo *Designing Reusable Classes*, de Johnson y Foote[6]. El diseño basado en inversión de control es un enfoque opuesto al que se suele utilizar en la programación imperativa.

Tradicionalmente, el código de una aplicación depende de librerías genéricas a las que llama para ejecutar tareas genéricas. Sin embargo, en inversión de control, el objetivo es hacer que el código más genérico llame al código concreto.

En este concepto se basan patrones de diseño como *frameworks*, *callbacks*, planificadores e inyección de dependencias.

Martin Fowler expone un ejemplo de este fenómeno en su artículo del mismo título[8]. En él presenta dos casos con un código con objetivo similar: interacción con el usuario, pero donde la dirección del control es completamente opuesta.

La forma tradicional es la siguiente:

Listado 2.1 Interacción con el usuario sin IoC.

```ruby
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

Introduciendo inversión de control a través de un sistema de ventanas:

Listado 2.2 Interacción con el usuario usando IoC.

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Hay una gran diferencia en la dirección del control de los programas. En el código tradicional, hay control total sobre cuándo se llama a los métodos `process_name` y `process_quest`. En el código con inversión de control el sistema de ventanas tiene el control sobre cuándo llamar a los métodos.

2.6.2 Principio de inversión de dependencia

Parte de los conocidos como principios SOLID de diseño orientado a objetos. Este principio se refiere a una forma concreta de desacoplar módulos del *software*[7].

Por este principio se intenta que los módulos de alto nivel no dependan de los de bajo nivel, sino de abstracciones.

En el artículo *DIP in the Wild*, de Brett L. Schuchert[9], se describen diferentes aplicaciones prácticas de este principio.

2.6.3 Inyección de dependencias

La inyección de dependencias es un patrón de diseño *software* que implementa inversión de control. Este patrón también ayuda a seguir el principio de inversión de dependencia explicado anteriormente. Apareció por primera vez en el artículo *Inversion of Control Containers and the Dependency Injection pattern*, de Martin Fowler[10].

Una inyección consiste en pasar una dependencia (o servicio) a un objeto que depende de ella (o cliente). La dependencia pasa a ser parte del estado del dependiente.

La inyección de dependencias se puede conseguir de más de una forma, pero las más comunes son los llamados contenedores, o *frameworks* (como Spring o Guice).

En el desarrollo de este proyecto se usarán tanto Guice para la aplicación principal escrita en Java; como contenedores, para el cliente REST escrito en Ruby.

Esta forma de desarrollar permitirá escribir el código de la aplicación de forma más rápida y limpia, teniendo también el beneficio de ser mucho más fácil de probar, incluso cuando las dependencias no existan aún.

Originalmente, la aplicación en Java no seguía este enfoque, y eso resultaba en clases más acopladas a sus dependencias, grandes dificultades a la hora de hacer cambios y en general más código.

2.6.3.1 Guice

Guice es la solución de Google para inyección de dependencias. Es un *framework* ligero y muy extendido que permite definir las reglas a utilizar programáticamente, a diferencia de otros *frameworks* como Spring[12].

El objetivo de Guice es evitar la necesidad de usar *factories* en el código, evitar usar `new` en el código. Aún habrá casos en los que es necesario usarlos, pero serán pocos.

La principal ventaja es hacer el código fácil de cambiar, probar y reusar. Combinado con *frameworks* para *mocking* como Mockito, es una herramienta extremadamente poderosa para escribir código bien probado y modular. Por el contrario, el simple hecho de usar `new MyClass()` en el código hace imposible sustituir la instancia de `MyClass` con un objeto falso que asegure no causar efectos secundarios, no permitiendo probar el código de forma aislada, aun si estos `new` están dentro de *factories* u se usan otros patrones de creación.

Guice se beneficia de la naturaleza estrictamente tipada de Java, haciendo uso de herramientas incluidas desde hace varias versiones de Java, como son los genéricos y las anotaciones.

Guice contiene reglas por defecto y se pueden extender creando módulos. Los módulos son instancias de clases que heredan de `AbstractModule` y sobrescriben su método `configure()`. Desde este método `configure()` se tiene acceso a crear reglas que se cargarán cuando se instancie el inyector.

Cuando se trabaja con Guice:

- Se crean uno o más módulos y se le pasan a `Guice.createInjector()`.
- Guice creará un `Binder` y se lo pasará al módulo o módulos.
- Cada módulo usará el `Binder` para definir sus *bindings*.
- Basándose en esos *bindings*, Guice crea un `Injector` y lo devuelve.

- Se usa el inyector para crear instancias.

Para entender bien el funcionamiento de Guice, hay que entender bien el modelo de inyección que usa. En la guía de usuario de Guice aparece la figura 2.4 como esquema conceptual.

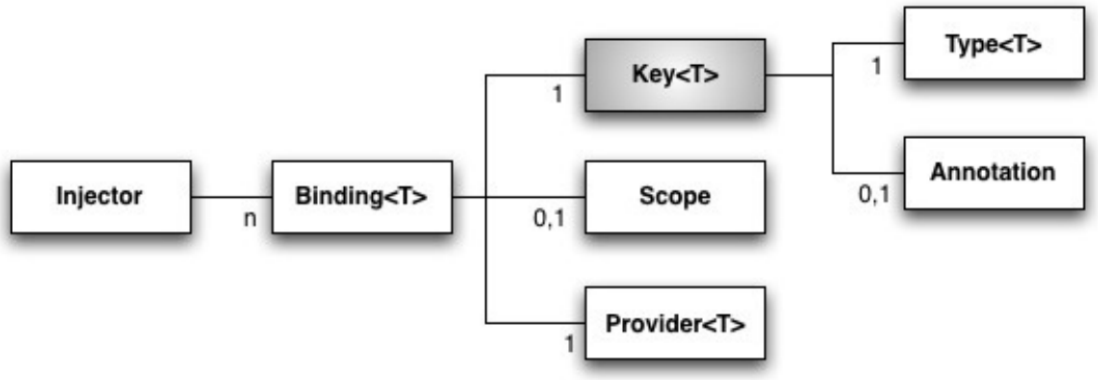

Figura 2.4 Modelo de inyección de Guice.

En esta figura, se ve como un `Injector` contiene varios `Binding`, identificados unívocamente por una clave, que se compone de un tipo y una anotación opcional. Esto permite asociar varios *bindings* a un único tipo.

Cada `Binding` también tiene un `Provider` que provee instancias del tipo necesario. Dada una clase, Guice es capaz de crear instancias de ella mientras tenga un `Binding` asociado.

Además, cada `Binding` tiene un ámbito asociado. Por defecto, Guice crea una instancia nueva cada vez que se le pide una instancia de una clase, pero esto se puede controlar y por ejemplo limitar a una única instancia para toda la aplicación.

La idea detrás de Guice es usar el inyector el mínimo número posible de veces (idealmente una), para crear un objeto raíz. Guice a su vez inyectará las debidas dependencias en ese objeto raíz, y así recursivamente.

Hay diferentes formas de declarar un `Binding`:

- Usando el método `bind` provisto si se hereda de `AbstractModule`, siendo la llamada al `Binder` implícita.

- Creando métodos anotados con `@Provides` en un módulo.

- Creando clases que implementan `Provider<T>` y creando un `Binding` con estos proveedores.

Guice también incluye *bindings* por defecto, como por ejemplo para `java.util.logging.Logger`.

Si se quiere inyectar un tipo parametrizado se puede hacer uso de `TypeLiteral<T>`.

Si no hay un `Binding` asociado a la clase, de la que se depende, Guice intentará inyectarla en cualquier caso, buscando un constructor anotado con `@Inject`, una anotación `@ImplementedBy` en el caso de una interfaz o llamando al constructor por defecto.

Por cada tipo que Guice conoce, también puede inyectar un proveedor para ese tipo.

2.7 Otros elementos a considerar

- **JUnit**: es un *framework* para realizar *tests* unitarios en Java. Define una serie de anotaciones orientadas a crear *suites* de pruebas.

- **Maven**: Es una herramienta para la gestión de proyectos en Java. Permite resolver las dependencias que se hayan configurado, centralizando la configuración en el POM (*project object model*).

- **Mockito**: es un *framework* para crear *mocks* (objetos simulados). Estos objetos son útiles a la hora de probar código, ya que permite cambiar el valor retornado por métodos y verificar los métodos que se han llamado en la ejecución de un *test*. Una ventaja clave de usar *mocks* es poder probar una clase cuando depende de código aún no implementado.

- **Rack**: es una interfaz mínima entre servidor web y *frameworks* de Ruby. Como su nombre indica, provee una estructura sobre la que montar aplicaciones por capas.

- **RSpec**: es una librería para probar el comportamiento de código en Ruby. Permite definir grupos de ejemplos y aporta forma de verificar el comportamiento de las clases que forman la aplicación.

- **Bundler**: al igual que Maven en Java, Bundler permite resolver las librerías (gemas) necesarias en un proyecto de Ruby.

- **Git**: Sistema de control de versiones utilizado para el desarrollo del proyecto, permite utilizar varias ramas de desarrollo y combinarlas formando versiones estables. Se han usado repositorios privados de Bitbucket como remoto.

3 Desarrollo

3.1 Programa principal

El programa principal es capaz de conectar un dominio DDS con un *broker* MQTT. Escucha mensajes en ambos lados y retransmite hacia el otro. Idealmente, su presencia sería totalmente transparente al resto del sistema.

La figura 3.1 muestra un posible sistema en el que hay dos dominios DDS independientes entre los que queremos intercambiar mensajes y a su vez también clientes MQTT.

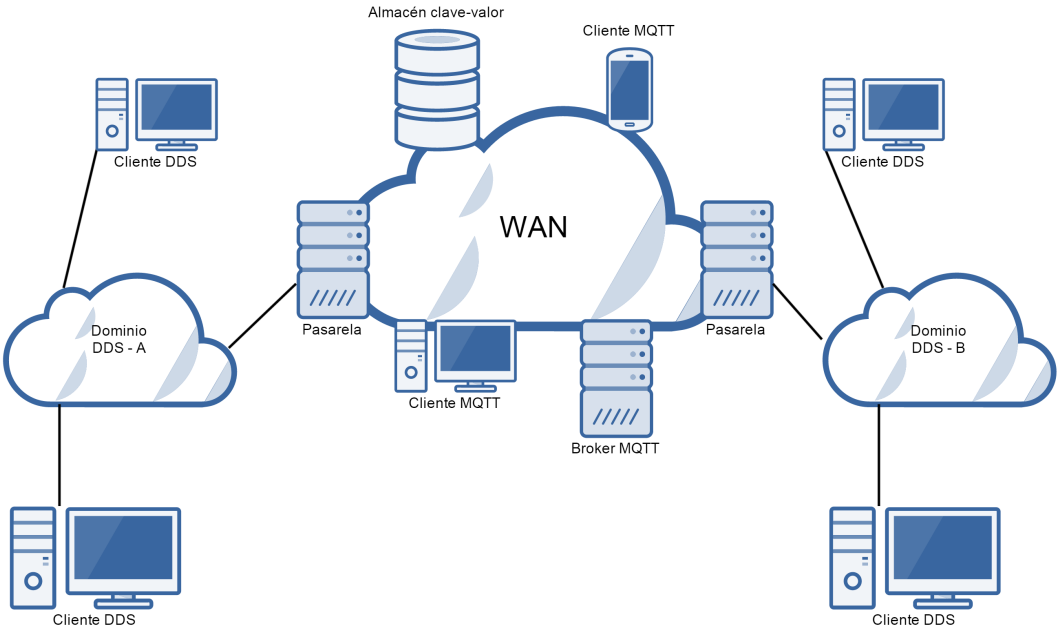

Figura 3.1 Ejemplo de un sistema con la aplicación desplegada.

Sería necesaria una instancia de la pasarela por cada dominio DDS, haciendo que ambas hablen con el mismo *broker* y almacén clave-valor.

A nivel programático, el reto clave consiste en interpretar la estructura arbitraria de los mensajes provenientes de ambos lados de la comunicación y retransmitirlos por la otra interfaz de forma que se permita comunicación DDS a DDS con intermediarios MQTT o comunicación DDS a MQTT con un cliente preparado para entender el formato de los mensajes que se envían.

La complejidad de la aplicación viene en su mayoría de la necesidad de interpretar dichos tipos arbitrarios en el lado DDS y de crear una estructura que permita representar los mensajes serializados en el extremo MQTT. De igual manera, también debe generar los mensajes en el lado DDS a partir de las representaciones JSON provenientes del *broker* MQTT.

La solución desarrollada permite interpretar:

- Estructuras de tipos simples (booleanos, bytes, caracteres, *doubles*, flotantes, enteros, *shorts* y `String`.
- Estructuras con *arrays* de dichos tipos.
- Estructuras anidadas.
- Estructuras con *arrays* de otras estructuras.

El formato de los mensajes MQTT será el siguiente:

Listado 3.1 Estructura JSON de los mensajes MQTT.

```
{
  "msg":{
    "json":"representation",
    "of":"message",
    "sent":"by",
    "dds":"publisher"
  }
  "type":"nameoftype",
  "cid":"mqttclientid"
}
```

Los tipos serializados de los mensajes se guardarán en un almacén clave-valor centralizado en Redis, compartido por todas las instancias de la aplicación, la primera vez que se descubran.

Es necesario guardar estos tipos para que la pasarela sea capaz de publicar los mensajes recibidos del lado MQTT en el dominio DDS acorde al `TypeCode`.

3.1.1 Estructura

La aplicación de Java está compuesta de varios paquetes, estructurados de la siguiente forma:

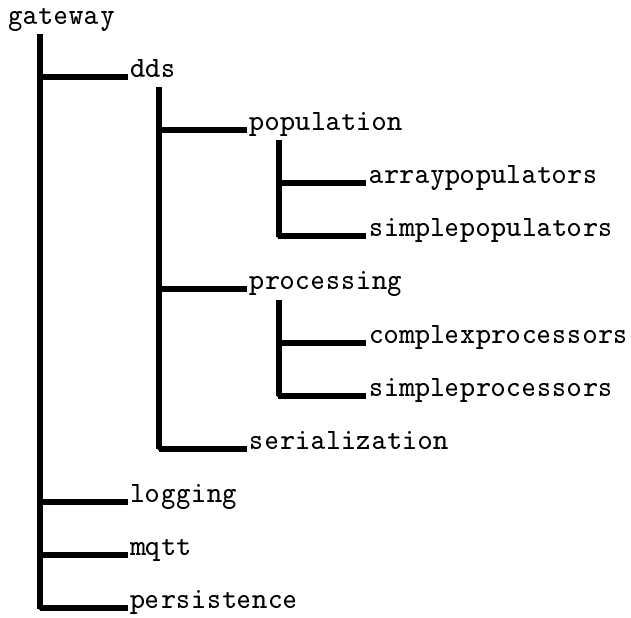

Figura 3.2 Estructura de paquetes de la aplicación Java.

Este árbol de paquetes está repetido en el entorno de *tests*, para así no tener que cambiar la visibilidad para poder probar las clases.

A continuación se expondrán algunos de los criterios de diseño seguidos para el desarrollo de esta aplicación, para luego pasar a explicar el contenido de cada paquete.

El contenido del paquete `gateway` se describirá primero, seguido del resto de paquetes.

Para describir una clase se listarán los siguientes elementos, cuando procedan:

- **Responsabilidad**[1]
- **Visibilidad**
- **Interfaces**
- **Superclases**

[1] Normalmente definida como la razón de una clase para cambiar, según el SRP (*Single Responsibility Principle*) o principio de responsabilidad única, parte de los conocidos como principios SOLID[7] de diseño de las clases.

- **Atributos**[2]
  - **Públicos**
  - **Protegidos**
  - **Paquete**
  - **Privados**
- **Métodos**
  - **Públicos**
  - **Protegidos**
  - **Paquete**
  - **Privados**
- **Dependencias**
  - **Internas**
  - **Externas**
- **Detalles de interés**

Para describir una interfaz se listarán los siguientes elementos, si proceden:

- **Responsabilidad**
- **Visibilidad**
- **Métodos**
  - **Públicos**
  - **Protegidos**
  - **Paquete**
- **Dependencias**

[2] Como norma general, se intenta que las clases sean lo más robustas posible. Es decir, que una vez se construye la instancia, no se pueda alterar su estado. Eso hace que las clases sean más fáciles de probar y previene problemas asociados al uso en una aplicación multihilo. Sin embargo, esto no siempre es posible por completo.

  - **Internas**
  - **Externas**

- **Clases que la implementan**
- **Detalles de interés**

3.1.2 Criterios de diseño

Se intentará mantener un criterio único a la hora de diseñar las clases, siempre que sea viable. Entre los aspectos a tener en cuenta están:

- Mantener una única responsabilidad para cada clase.
- Si una clase implementa una interfaz y otra clase depende de ella, es preferible que dependa de la interfaz. Lo mismo se puede aplicar a subclases y superclases mientras no se necesite parte de la interfaz pública propia de la subclase.
- Si una clase recibe dependencias en el constructor que deben ser inmutables, se guardarán en atributos privados con modificador `final`.
- Si una clase no debe usarse fuera de su paquete, su visibilidad será de paquete (por defecto).
- La interfaz pública de las clases debe ser la mínima posible para su uso, siempre que interfaces y visibilidad lo permitan.
- Todas las clases deben permitir que sus instancias sean inyectadas usando Guice.

3.1.3 Paquete Gateway

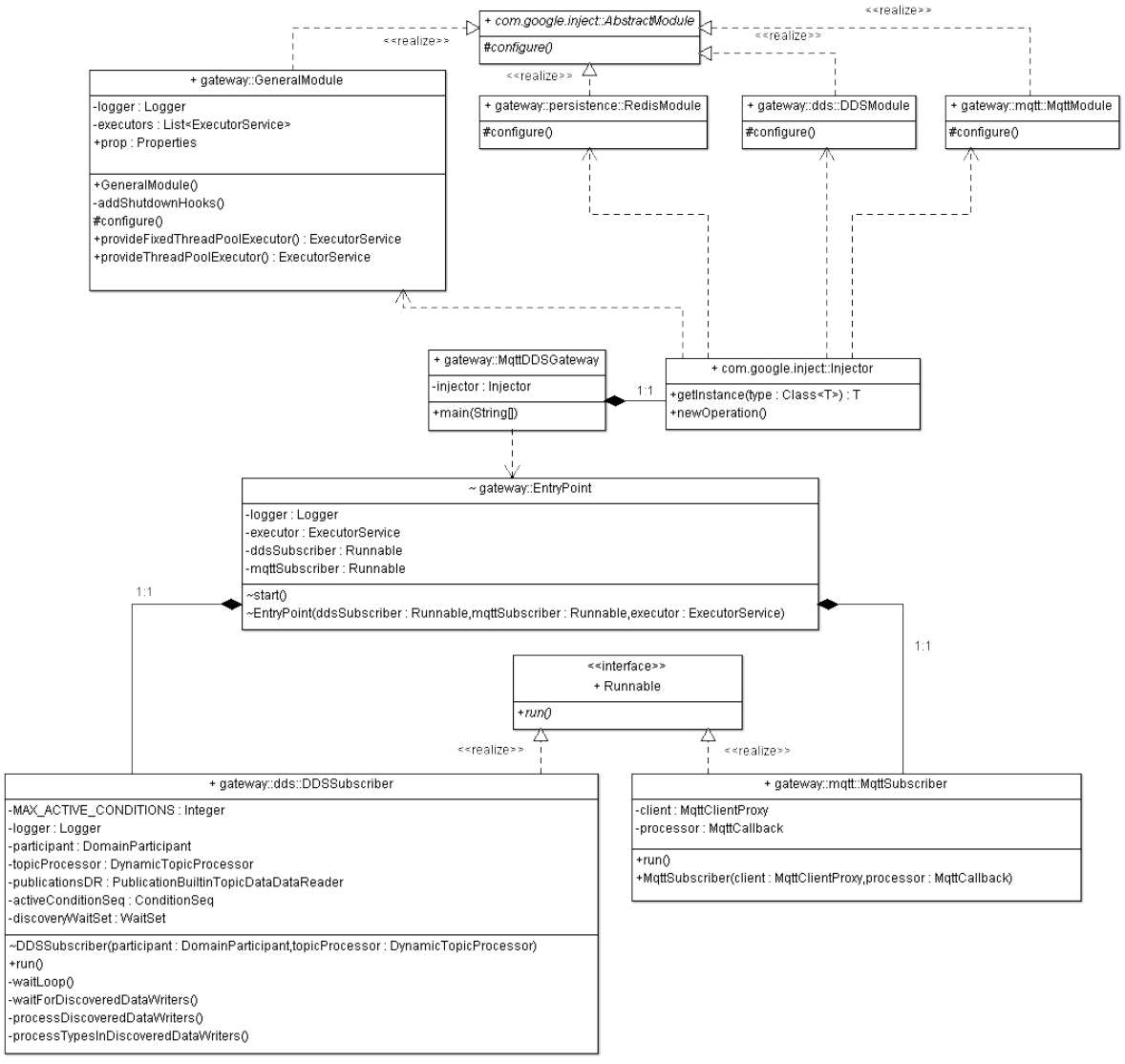

Figura 3.3 Diagrama UML del paquete Gateway.

3.1.3.1 MqttDDSGateway

Responsabilidad

Es la clase principal del programa, en el paquete principal, `gateway`. Como se está usando Guice para inyección de dependencias, conviene que sea lo más simple posible[12]. Se encarga de instanciar el inyector y utilizarlo para crear la primera instancia de `EntryPoint`, clase que se describirá más adelante.

Visibilidad

Pública

Atributos

Privados

- `injector`: Instancia de `Injector`, perteneciente a la librería Guice. Se encargará de inyectar las dependencias necesarias para instanciar `EntryPoint`. Se inicializa llamando a `Guice.createInjector`, pasando como parámetros los módulos `GeneralModule`, `DDSModule`, `RedisModule` y `MqttModule`. Tiene los modificadores `static` y `final`.

Métodos

Públicos

- `main()`: Método principal, utiliza el `injector` para instanciar `EntryPoint` y luego llama a su método `start()`. No devuelve nada.

Dependencias

Internas

- `EntryPoint`
- **Módulos de Guice:** `GeneralModule`, `DDSModule`, `RedisModule` y `MqttModule`.

Externas

- **Guice**: `Injector`.

Detalles de interés

En sus orígenes, esta clase contenía la responsabilidad de `EntryPoint`. Cuando se introdujo la inyección de dependencias se vio claramente el beneficio de extraer todo lo posible de `MqttDDSGateway` para así usar el inyector para resolver todas las dependencias una única vez.

En ese momento, se creó `EntryPoint` con la responsabilidad de iniciar los dos hilos principales de los subscriptores DDS y MQTT.

3.1.3.2 EntryPoint

Responsabilidad

Es la clase responsable de lanzar los dos hilos principales (los subscriptores DDS y MQTT). Se instancia en la clase principal usando el inyector de Guice, por lo tanto, todas sus dependencias son inyectadas nada más empezar el programa.

Visibilidad

Paquete

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `executor`: De tipo `ExecutorService`, corresponde con un `ThreadPoolExecutor` de dos hilos que se inyectará.
- `ddsSubscriber`: Un `Runnable` inyectado. Se espera una instancia de `DDSSubscriber`.
- `mqttSubscriber`: Un `Runnable` inyectado. Se espera una instancia de `MqttSubscriber`.

Métodos

Paquete

- `EntryPoint(Runnable, Runnable, ExecutorService)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos. El parámetro `ddsSubscriber` está

anotado con `@Named("ddsSubscriber")`, `mqttSubscriber` con `@Named ("mqttSubscriber")` y `executor` con `@Named("fixedThreadPool")`.

- `start()`: Ejecuta ambos `Runnables` usando `executor` y espera a su finalización. No devuelve nada.

Dependencias

Internas

- `ApplicationLoggerFactory`

Externas

- **Guice**: `Inject` y `Named`.
- **Log4j**: `Logger`

Detalles de interés

Originalmente, ambos `Runnable` eran parte del paquete Gateway, ya que contenían código conectado a ambos extremos de la comunicación. Además, se instanciaban directamente en la clase principal. Cuando se empezaron a extraer responsabilidades, uno quedó enmarcado en el lado DDS y otro en el lado MQTT, por lo que se movieron a sus paquetes correspondientes.

Para entender el funcionamiento básico de la aplicación es necesario el siguiente diagrama:

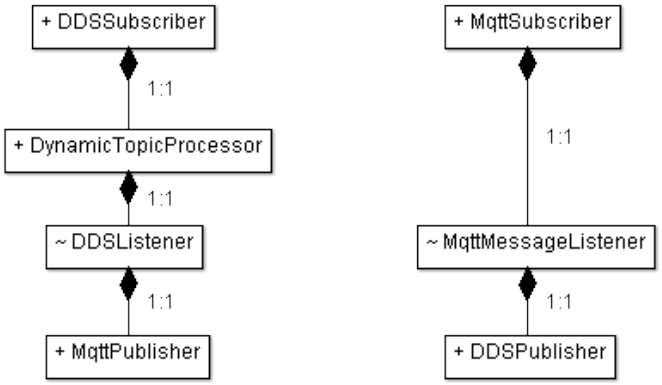

Figura 3.4 Diagrama UML de los componentes principales.

Así, se ve que mediante inyección, el publicador es un componente de los *listeners* de cada sentido de la comunicación, que a su vez son componentes de los suscriptores. Por lo tanto, al instanciar los suscriptores y ejecutarlos se están creando los publicadores necesarios para el funcionamiento de la aplicación.

3.1.3.3 GeneralModule

Responsabilidad

Uno de los módulos de Guice que permite inyectar dependencias a las clases del programa. Se encarga de atar las dependencias al proveedor que debe construirlas[12].

Visibilidad

Pública

Superclases

- `AbstractModule`: Es una clase abstracta de la que se puede extender y sobrescribir `configure()` para escribir las reglas del `Binder`[3].

Atributos

Públicos

- `prop`: Es un objeto `Properties` que se carga utilizando `PropertyReader`. Tiene los modificadores `static` y `final`.

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `executors`: Es una lista de `ExecutorService` que contiene aquellos que existan para que se puedan acabar antes de terminar la ejecución del programa. Tiene los modificadores `static` y `final`.

Métodos

Públicos

- `GeneralModule()`: Agrega un *shutdown hook* a la ejecución, que terminará los ejecutores que haya en `executors`.

Protegidos

- `loadProperties()`: Carga las propiedades establecidas en `config.properties` usando `PropertyReader`. Devuelve un objeto `Properties`.
- `configure()`: Sobrescribe el método `configure()` de `AbstractModule`. No devuelve nada.
- `provideFixedThreadPoolExecutor()`: Anotado con `@Provides` y `@Named("fixed-ThreadPool")`. Instancia un `ExecutorService` de tipo `ThreadPoolExecutor` con un número fijo de hilos (2) y lo agrega a `executors`. Devuelve el `ExecutorService`.

[3] El `Binder` reúne las reglas o *bindings* que indican al inyector cómo construir cada dependencia.

- `provideThreadPoolExecutor()`: Anotado con `@Provides` y `@Named("threadPool")`. Instancia un `ThreadPoolExecutor` y lo agrega a `executors`. Devuelve el `ExecutorService`.

  Se establecen los siguientes *bindings*:

Tabla 3.1 *Bindings* de Guice en `GeneralModule`.

Dependencia	Proveedor	Ámbito	Nombre
`ExecutorService`	`provideThreadPoolExecutor()`	Instancia	threadPool
`ExecutorService`	`provideFixedThreadPoolExecutor()`	Instancia	fixedThreadPool

  Para crear el *binding* usando tipos genéricos se utiliza `TypeLiteral`, también proporcionado por Guice.

Privados

- `addShutdownHooks()`: Agrega un `shutdown hook` a la ejecución para llamar `shutdown()` sobre cada ejecutor almacenado en `executors`.

Dependencias

Internas

- `ApplicationLoggerFactory`

Externas

- **RTI DDS**: `TypeCode`.
- **Guice**: `AbstractModule`, `Named` y `Provides`.
- **Log4j**: `Logger`

Detalles de interés

La documentación de Guice[12] es muy extensa, tanto que aunque sea muy completa resulta difícil encontrar qué se necesita para cada caso de uso. Por lo tanto, hay detalles o errores para los que cuesta encontrar una solución satisfactoria.

Algunos de los errores surgidos trabajando con Guice son los siguientes:

- Originalmente se había creado un único módulo para Guice, pero al estar la aplicación estructurada en paquetes, esto obligaba a cambiar la visibilidad de algunas clases. Por lo tanto, se crearon distintos en base al que existía.

- Dificultades para crear una regla que inyecte un objeto con tipos genéricos: Java aún no permite representar tipos genéricos. Por eso, la solución pasa por usar `TypeLiteral<T>`[13], también parte de Google Inject, que permite representar dichos tipos.

- Guice inyecta el mismo `ExecutorService` en más de una clase: Cuando se empieza a usar Guice, se tiende a escribir reglas para lo que no se necesitan. Si se crea un método anotado con `@Provides` y se crea la regla utilizando `.toInstance(providesMethod())`, esta regla siempre devolverá la misma instancia. Si por el contrario, simplemente se omite la regla, Guice sabrá cómo instanciar el objeto cada vez que se necesite.

3.1.3.4 PropertyReader

Responsabilidad

Es la clase encargada de leer el fichero de propiedades y encapsularlo en un objeto `Properties`.

El programa recibe parámetros de configuración definidos en el fichero `config.properties`. Luego, se utiliza la librería Guice para inyectar dependencias y evitar instanciar clases manualmente.

Un ejemplo del fichero de propiedades es:

Listado 3.2 Listado de propiedades.

```
1  redis.hostname=localhost
2  redis.port=6379
3  redis.timeout=10
4  #redis.password=
5  redis.key=typecodes
6  mqtt.hostname=localhost
7  mqtt.port=1883
8  mqtt.prefix=Gateway/
9  mqtt.clientId=Gateway
10 domain.id=0
```

Visibilidad

Paquete

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

Métodos

Paquete

- `getProperties(String)`: Lee el archivo de propiedades cuyo nombre se le ha pasado y lo encapsula en el objeto `Properties`, que devuelve. Si no encuentra el fichero de propiedades, lanza una `FileNotFoundException`.

Dependencias

Internas

- **Fichero de propiedades**
- `ApplicationLoggerFactory`

Externas

- **Log4j**: `Logger`

3.1.4 Paquete DDS

Este paquete enmarca todas las clases que trabajan directamente con el lado DDS de la comunicación, ya sean las entidades de RTI DDS o el tratamiento de los datos utilizando `DynamicData`.

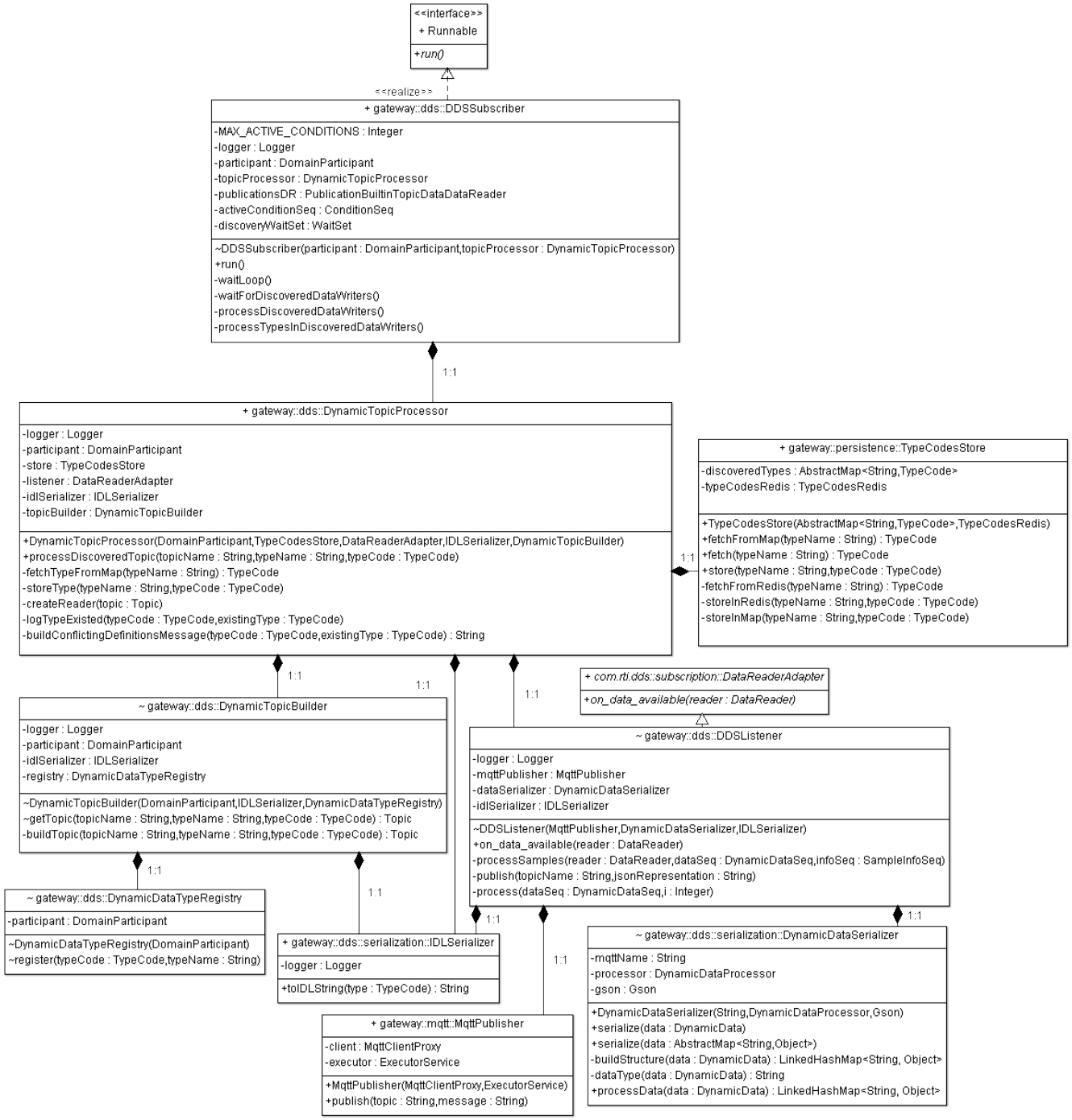

Figura 3.5 Diagrama UML del paquete DDS.

3.1.4.1 ApplicationParticipantFactory

Responsabilidad

Es una *factory* encargada de instanciar el `DomainParticipant` que se usará en la aplicación. Necesita el identificador del dominio.

Visibilidad

Paquete

Métodos

Paquete

- `build(DomainContainer)`: Usa `TheParticipantFactory.create_participant(int, DomainParticipantQos, DomainParticipantListener, int)` para instanciar el `DomainParticipant`. Es un método estático y devuelve el `DomainParticipant`.

Dependencias

Externas

- **RTI DDS**: `DomainParticipant`, `DomainParticipantFactory` y `StatusKind`.

3.1.4.2 DDSListener

Responsabilidad

Es la clase encargada de leer del `DataReader` que se cree para cada `Topic` y llamar al `MqttPublisher` para que publique cada mensaje recibido por la interfaz MQTT.

Visibilidad

Paquete

Superclases

- `DataReaderAdapter`: Es un *listener* que sigue el espíritu de los adaptadores de AWT (implementaciones vacías de los *callbacks*). En particular, se usará `on_data_available(DataReader)`.

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `mqttPublisher`: Instancia de `MqttPublisher`.
- `dataSerializer`: Instancia de `DynamicDataSerializer`.
- `idlSerializer`: Instancia de `IDLSerializer`.

Métodos

Públicos

- `on_data_available(DataReader)`: Define el *callback* que se llama cuando el `DataReader` tiene datos disponibles. Se hace un *cast* a `DynamicDataReader` y se toma la secuencia de datos disponibles. Entonces, se llama a `processSamples(DataReader)` con el `DataReader` en cuestión. No devuelve nada.

Paquete

- `DDSListener(MqttPublisher, DynamicDataSerializer, IDLSerializer)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Privados

- `processSamples(DataReader, DynamicDataSeq, SampleInfoSeq)`: Itera sobre los elementos contenidos en `dataSeq` y si estos son válidos los procesa, obteniendo su representación JSON. Luego, llama a `publish(String, String)` con el nombre del topic y la representación JSON. No devuelve nada.
- `process(DynamicDataSeq, int)`: Toma el i-ésimo elemento de la secuencia. Escribe en el `logger` la representación IDL del tipo como ayuda para el desarrollo, usando `idlSerializer`. Luego, utiliza `dataSerializer` para serializar el mensaje y devolver dicha representación.
- `publish(String, String)`: Delega la publicación sobre `mqttPublisher`. No devuelve nada.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `DynamicDataSerializer`
- `IDLSerializer`
- `MqttPublisher`

Externas

- **RTI DDS**: `DynamicData`, `DynamicDataReader`, `DynamicDataSeq`, `DataReader`, `DataReaderAdapter`, `Subscriber`, `StatusKind`, `InstanceStateKind`, `SampleInfo`, `SampleInfoSeq`, `SampleStateKind`, `ViewStateKind`, `RETCODE_NO_DATA` y `ResourceLimitsQosPolicy`.
- **Log4j**: `Logger`
- **Guice**: `Inject`.

Detalles de interés

Esta clase está basada en una clase interna contenida en el ejemplo DataSpy DDS[18] de RTI Connext DDS, llamada `PrintDynamicDataListener`. Su propósito era leer los datos recibidos e imprimir información de ellos. Siguiendo la estructura de esta clase, ahora se imprime información sobre la estructura de los mensajes y se da la orden de enviarlos hacia la interfaz MQTT utilizando a `MqttPublisher`.

Respecto a la clase original, se ha intentado que el estado de la clase sea mínimo. Eso implica hacer que `dataSeq` e `infoSeq` dejen de ser atributos y sean variables locales. Esto podría evitar problemas de concurrencia.

3.1.4.3 DDSModule

Responsabilidad

Uno de los módulos de Guice que permite inyectar dependencias relacionadas con DDS a las clases del programa. Se encarga de atar las dependencias al proveedor que debe construirlas.

Visibilidad

Pública

Superclases

- `AbstractModule`: Es una clase abstracta de la que se puede extender y sobrescribir `configure()` para escribir las reglas del `Binder`.

Métodos

Protegidos

- `configure()`: Sobrescribe el método `configure()` de `AbstractModule`. No devuelve nada. Establece los siguientes *bindings*:

Tabla 3.2 *Bindings* de Guice en `DDSModule`.

Dependencia	Proveedor	Ámbito	Nombre
`DomainContainer`	`DomainContainerProvider`	Instancia	-
`DomainParticipant`	`DomainParticipantProvider`	*Eager Singleton*[4]	-
`DynamicDataSerializer`	`DynamicDataSerializerProvider`	Instancia	-
`Runnable`	`DDSSubscriber`	Instancia	ddsSubscriber
`DataReaderAdapter`	`DDSListener`	Instancia	-

Dependencias

Internas

- `DDSListener`
- `DDSSubscriber`
- `DomainContainer`
- `DynamicDataSerializer`
- **Proveedores**: `DomainContainerProvider`, `DynamicDataSerializerProvider` y `DomainParticipantProvider`.

[4] Debe ser *eager singleton* para garantizar que hay un único `DomainParticipant` en la aplicación.

Externas

- **RTI DDS**: `DomainParticipant` y `DataReaderAdapter`.
- **Guice**: `AbstractModule` y `Names`.

3.1.4.4 DDSPublisher

Responsabilidad

Es la clase encargada de publicar un mensaje DDS de forma asíncrona. Recibe nombre del *topic*, tipo, nombre del tipo y el mensaje en un `AbstractMap<String,Object>`.

Visibilidad

Pública

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `factory`: Instancia de `DynamicDataPopulator`.
- `participant`: Instancia de `DomainParticipant`.
- `executor`: instancia de `ExecutorService`.
- `topicBuilder`: instancia de `DynamicTopicBuilder`. Se encarga de buscar o crear el *topic* y registrar el tipo.

Métodos

Públicos

- `DynamicDataPublisher(DynamicDataPopulator, DomainParticipant, ExecutorService, DynamicTopicBuilder)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos. El parámetro `executor` está anotado con `@Named("threadPool")`.

- `publish(String, AbstractMap<String,Object>, String, TypeCode)`: Hace que `executor` ejecute el `Runnable` resultante de llamar a `getPublication(String, AbstractMap<String,Object>, String, TypeCode, DynamicTopicBuilder)` con los mismos parámetros que recibe y `topicBuilder`. No devuelve nada.

Privados

- `getPublication(String, AbstractMap<String,Object>, String, TypeCode, DynamicTopicBuilder)`: Crea un `Runnable` cuyo método `run()` contiene el proceso que lleva a publicar un mensaje DDS. Llama a `topicBuilder.getTopic(String, String, TypeCode)` y, si consigue un *topic*, usa `factory.populate(TypeCode, AbstractMap<String,Object>)` para rellenar `DynamicData`. Luego, llama a `write (AbstractMap<String,Object>, Topic, DynamicData)`. Devuelve el `Runnable`.

- `write(AbstractMap<String,Object>, Topic, DynamicData)`: Crea un `DynamicDataWriter` llamando a `buildDataWriter(Topic)`. Si consigue crearlo, publica el mensaje. No devuelve nada.

- `buildDataWriter(Topic)`: Crea un `DynamicDataWriter` llamando a `participant.create_datawriter(Topic, DataWriterQos, DataWriterListener, int)`. Devuelve el `DynamicDataWriter`.

Dependencias

Internas

- `ApplicationLoggerFactory`

- `DynamicDataPopulator`

- `DynamicTopicBuilder`

Externas

- **RTI DDS**: `DynamicData`, `DynamicDataWriter`, `InstanceHandle_t`, `StatusKind`, `DomainParticipant`, `Publisher`, `Topic` y `TypeCode`.

- **Guice**: `Inject` y `Named`.

- **Log4j**: `Logger`

Detalles de interés

El comportamiento asociado a buscar o crear un `Topic` estaba duplicado en esta clase y en `DynamicTopicProcessor`, por lo tanto se extrajo en una clase llamada `DynamicTopicBuilder`, que se puede reusar en ambas.

3.1.4.5 DDSSubscriber

Responsabilidad

Es la clase principal del subscriptor DDS. Se encarga de instanciar un subscriptor DDS y de pedir al `DynamicTopicProcessor` que procese un nuevo *topic* cuando se descubre. El código de esta clase está basado en el ejemplo DataSpy DDS que proporciona RTI.

Visibilidad

Pública

Interfaces

- `Runnable`: Se quiere ejecutar el contenido del método `run()` en un hilo.

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `MAX_ACTIVE_CONDITIONS`. Tiene los modificadores `static` y `final`. Define el número máximo de condiciones a las que hay que atender en el `WaitSet`.
- `participant`: Instancia de `DomainParticipant`. Entidad DDS que permite instanciar publicadores o subscriptores.
- `publicationsDR`: Instancia de `PublicationBuiltinTopicDataDataReader`, `DataReader` para uno de los *Built-in topics* de DDS.
- `activeConditionSeq`: Instancia de `ConditionSeq`.
- `discoveryWaitSet`: Instancia de `WaitSet`[17].

- `topicProcessor`: Instancia de `DynamicTopicProcessor`.

Métodos

Públicos

- `DDSSubscriber(DomainParticipant, DynamicTopicProcessor)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

- `run()`: Sobrescribe el método `run()` de `Runnable`. Instancia un subscriptor utilizando el atributo `participant`. Si no lo consigue escribe en el *log* un error severo. Después, asigna el subscriptor del *built-in* topic de publicaciones al atributo `publicationsDR`. Es también encargado de instanciar `discoveryWaitSet` y asignarle la condición para que despierte cuando hay datos disponibles en el *built-in topic*. Para terminar, llama a `waitLoop()`. Si en algún momento en este proceso ocurre algún error, destruye las entidades del participante y al participante, ya que no se puede continuar con la ejecución correcta del programa.

Privados

- `waitLoop()`: Bucle infinito que llama a `waitForDiscoveredDataWriters()` y `processDiscoveredDataWriters()` repetidamente. No devuelve nada.

- `waitForDiscoveredDataWriters()`: Espera indefinidamente a que se cumplan las condiciones definidas para el `WaitSet`. No devuelve nada.

- `processDiscoveredDataWriters()`: Comprueba que haya datos disponibles en el `DataReader` de publicaciones y llama a `processTypesInDiscoveredDataWriters()`. No devuelve nada.

- `processTypesInDiscoveredDataWriters()`: Toma muestras de `publicationsDR` mientras las haya, y si son nuevas llama a `topicProcessor.processDiscoveredTopic` con el nombre del *topic*, nombre del tipo y `TypeCode`. No devuelve nada.

Dependencias

Internas

- `ApplicationLoggerFactory`

- `DynamicTopicProcessor`

Externas

- **RTI DDS**: `DomainParticipant`, `DomainParticipantFactory`, `PublicationBuiltinTopicDataDataReader`, `ConditionSeq`, `WaitSet`, `Subscriber`, `StatusKind`, `Duration_t`, `PublicationBuiltinTopicData`, `SampleInfo`, `ViewStateKind`, `TypeCode`, `RETCODE_NO_DATA` y `RETCODE_TIMEOUT`.
- **Log4j**: `Logger`
- **Guice**: `Inject`.

Detalles de interés

Está basada en DataSpy DDS[18], uno de los ejemplos provistos por RTI para Connext DDS. Esta clase y `DDSListener` están extraídas y adaptadas al propósito del proyecto. La idea de emplear `Waitsets`[17] también surge de ahí. Sin embargo, aunque originalmente el código estaba destinado a descubrir tanto participantes como publicaciones, en el ámbito cubierto por este proyecto, la aplicación es totalmente agnóstica a nivel de participantes, centrándose en los datos y su estructura.

Como con cualquier fragmento de código no familiar, el proceso de entenderlo y adaptarlo al objetivo que se le quiere dar pasa por horas de lectura de documentación[16], uso del depurador y algo de prueba y error hasta que se consigue entender el funcionamiento.

3.1.4.6 DomainContainer

Responsabilidad

Encapsula el identificador del dominio DDS en un objeto inyectable. Así, se permiten más parámetros configurables si se necesitasen en un futuro.

Visibilidad

Paquete

Métodos

Públicos

- `DomainContainer(int)`: Instancia el objeto y asigna el identificador. `getId()`: Devuelve el identificador. `setId(int)`: Asigna el identificador.

3.1.4.7 DomainContainerProvider

Responsabilidad

Provee instancias de `DomainContainer`.

Visibilidad

Paquete

Interfaces

- `Provider<DomainContainer>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `DomainContainer`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `DomainContainer` pasándole el identificador del dominio. Devuelve el `DomainContainer`.

Dependencias

Internas

- `DomainContainer`
- `GeneralModule`: Propiedades (`domain.id`).

Externas

- **Guice**: `Provider`

3.1.4.8 DomainParticipantProvider

Responsabilidad

Provee instancias de `DomainParticipant`.

Visibilidad

Paquete

Interfaces

- `Provider<DomainParticipant>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `DomainParticipant`.

Atributos

Privados

- `domain`: Instancia de `DomainContainer`. Encapsula el identificador de dominio y cualquier otro parámetro que se quisiera introducir.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `DomainParticipant` utilizando `ApplicationParticipantFactory.build(DomainContainer)`. Luego llama `enable()` sobre él para permitirle crear entidades DDS. Devuelve el `DomainParticipant`.

Paquete

- `DomainParticipantProvider(DomainContainer)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `ApplicationParticipantFactory`
- `DomainContainer`

Externas

- **RTI DDS**: `DomainParticipant`

- **Guice**: `Inject` y `Provider`.

3.1.4.9 DynamicDataSerializerProvider

Responsabilidad

Provee instancias de `DynamicDataSerializer`.

Visibilidad

Paquete

Interfaces

- `Provider<DynamicDataSerializer>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `DynamicDataSerializer`.

Atributos

Privados

- `processor`: Instancia de `DynamicDataProcessor`.
- `gson`: Instancia de `Gson`, parte de `com.google.gson`, para la serialización/deserialización JSON.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `DynamicDataSerializer`, pasándole el identificador de cliente MQTT, `processor` y `gson`. Devuelve el `DynamicDataSerializer`.

Paquete

- `DynamicDataSerializerProvider(DynamicDataProcessor, Gson)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `DynamicDataProcessor`
- `DynamicDataSerializer`
- `GeneralModule`: Propiedades (`mqtt.clientId`).

Externas

- **RTI DDS**: `DomainParticipant`
- **Guice**: `Inject` y `Provider`.
- **Gson**: `Gson`

Detalles de interés

Durante el desarrollo, existía un paquete `gateway.serialization`, pero se eliminó, ya que por problemas de visibilidad tenía más sentido que cada clase de ese paquete estuviese en el paquete apropiado según sus dependencias y las clases que la usasen. En este caso, el paquete `gateway.dds`.

3.1.4.10 DynamicDataTypeRegistry

Responsabilidad

Es una pequeña clase auxiliar que surgió de la necesidad de probar el comportamiento de otras clases. Tener que registrar un tipo las acoplaba con `DynamicDataTypeSupport`, por lo que era difícil crear un *test* completo. Al extraer la clase, sus instancias se pueden sustituir por un objeto falso que permite probar su funcionalidad.

Visibilidad

Paquete

Atributos

Privados

- `participant`: Instancia de `DomainParticipant`.

Métodos

Paquete

- `DynamicDataTypeRegistry(DomainParticipant)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `register(TypeCode, String)`: Registra el tipo con el nombre dado. No devuelve nada.

Dependencias

Externas

- **RTI DDS**: `DomainParticipant`, `DynamicDataTypeSupport` y `TypeCode`.
- **Guice**: `Inject`.

Detalles de interés

La usa `DynamicTopicBuilder`, aunque existe desde antes de que la responsabilidad de esta fuese extraída. Anteriormente era usada por `DynamicTopicProcessor` y `DynamicDataPublisher`.

3.1.4.11 DynamicTopicBuilder

Responsabilidad

Otra clase auxiliar que surgió de la necesidad de probar el comportamiento de otras clases. Obtener un `Topic` es una responsabilidad en sí misma y por lo tanto corresponde a una clase separada.

Visibilidad

Paquete

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

- `registry`: Instancia de `DynamicDataTypeRegistry`.
- `idlSerializer`: Instancia de `IDLSerializer`.
- `participant`: Instancia de `DomainParticipant`.

Métodos

Paquete

- `DynamicTopicBuilder(DomainParticipant, IDLSerializer, DynamicDataTypeRegistry)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `getTopic(String, String, TypeCode)`: Comprueba si hay una definición del *topic* llamando a `participant.lookup_topicdescription(topicName)`. Si la hay la devuelve, si no imprime en `logger` informando de que es un *topic* nuevo y llama a `buildTopic(String, String, TypeCode)` con los mismos parámetros que recibe. Devuelve el `Topic`.

Privados

- `buildTopic(String, String, TypeCode)`: Construye un objeto `Topic` llamando a `participant.create_topic(String, String, TopicQos, TopicListener, int)`. Si no lo consigue crear, lo escribe en el `logger`. Devuelve el `Topic`.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `IDLSerializer`
- `DynamicDataTypeRegistry`

Externas

- **RTI DDS**: `DomainParticipant`, `Topic` `DynamicDataTypeSupport` y `TypeCode`.
- **Log4j**: `Logger`.
- **Guice**: `Inject`.

Detalles de interés

Es usada por `DynamicTopicProcessor` y `DynamicDataPublisher`. Antes, ambas clases contenían este comportamiento, por lo que fue extraído. Además, se trasladó la dependencia sobre `DynamicDataTypeRegistry` hacia esta clase.

3.1.4.12 DynamicTopicProcessor

Responsabilidad

Es la clase encargada de procesar un mensaje recibido por la interfaz DDS y crear un `DataReader` acoplado con `DDSListener`, que se encargará de leer de él cuando se reciba un mensaje.

Visibilidad

Pública

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `participant`: Instancia de `DomainParticipant`.
- `store`: Instancia de `TypeCodesStore`.
- `listener`: Instancia de `DDSListener`.
- `idlSerializer`: instancia de `IDLSerializer`.
- `topicBuilder`: instancia de `DynamicTopicBuilder`. Se encarga de buscar o crear el *topic* y registrar el tipo.

Métodos

Públicos

- `DynamicTopicProcessor(DomainParticipant, TypeCodesStore, DDSListener, IDLSerializer, DynamicTopicBuilder)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `processDiscoveredTopic(String, String, TypeCode)`: Escribe en el `logger` información sobre el topic y el tipo. Después, busca la definición del tipo en memoria. Si no está en memoria se almacenará en memoria y en Redis usando `store` y se usará `topicBuilder.getTopic(String, String, TypeCode)` para obtener una instancia de `Topic`. Luego, se creará el `DataReader` correspondiente. Si ya existe, se llama a `logTypeExisted(TypeCode, TypeCode)` con ambos tipos. No devuelve nada.

Privados

- `fetchTypeFromMap(String)`: Delega sobre `store` para extraer un tipo de memoria. Devuelve el `TypeCode`.
- `storeType(String, TypeCode)`: Delega sobre `store` para almacenar un tipo en memoria y en Redis. No devuelve nada.
- `createReader(Topic)`: Crea un `DataReader` para el *topic* en cuestión y lo acopla a `listener`. No devuelve nada.
- `logTypeExisted(TypeCode, TypeCode)`: Recibe el tipo ya existente y el nuevo y los compara. Escribe en el `logger` el resultado de dicha comparación, usando `buildConflictingDefinitionsMessage(TypeCode, TypeCode)` en caso de que no sean iguales. No devuelve nada.
- `buildConflictingDefinitionsMessage(TypeCode, TypeCode)`: Devuelve un `String` conteniendo las representaciones IDL de los tipos en conflicto para que pueda ser escrita en el `logger`.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `TypeCodesStore`

- `IDLSerializer`
- `DynamicTopicBuilder`

Externas

- **RTI DDS**: `DomainParticipant`, `Subscriber`, `Topic`, `TypeCode` y `StatusKind`.
- **Log4j**: `Logger`.
- **Guice**: `Inject`.

Detalles de interés

Al principio el comportamiento asociado a la serialización y persistencia de `TypeCodes` estaba en esta clase. Luego se extrajeron a `TypeCodesSerializer` y `TypeCodesStore` respectivamente. Esta clase sólo usa directamente `TypeCodesStore`.

Por otro lado, el `ConcurrentSkipListMap` que almacenaba los tipos era una dependencia de esta clase. Ahora, está contenida dentro de `TypeCodesStore`, lo que ayuda a no mezclar responsabilidades.

3.1.4.13 TypeResolver

Responsabilidad

Clase auxiliar que resuelve los distintos `TCKind` a `String` para no hacer a las demás clases depender de `TCKind` o depender de un `switch` con `TCKind`.

Visibilidad

Pública

Atributos

Privados

- `TYPE_MAP`: Instancia de `ImmutableMap<TCKind, String>`, parte de las librerías de Guava, o más concretamente `com.google.common.collect`. Se utiliza un `builder` para construir el *map*. Tiene los modificadores `static` y `final`.

  Contiene los siguientes pares clave-valor:

Tabla 3.3 Pares clave-valor de `TypeResolver`.

Clave (`TCKind`)	**Valor** (`String`)
`TCKind.TK_SHORT`	`short`
`TCKind.TK_LONG`	`int`
`TCKind.TK_USHORT`	`short`
`TCKind.TK_ULONG`	`int`
`TCKind.TK_FLOAT`	`float`
`TCKind.TK_DOUBLE`	`double`
`TCKind.TK_BOOLEAN`	`boolean`
`TCKind.TK_CHAR`	`char`
`TCKind.TK_OCTET`	`byte`
`TCKind.TK_STRING`	`string`
`TCKind.TK_ARRAY`	`array`
`TCKind.TK_STRUCT`	`complex`
`TCKind.TK_UNION`	`complex`

Métodos

Públicos

- `get(TCKind)`: Busca un valor para esa clave en `TYPE_MAP`. Devuelve dicho valor.

Dependencias

Externas

- **RTI DDS**: `TCKind`.
- **Guava**: `ImmutableMap`.

Detalles de interés

Se usa `ImmutableMap` por la seguridad de no poder modificar el valor, ya que aunque se utilizase el modificador `final`, sería posible llamar a `put(TCKind, String)` sobre él y sus valores se modificarían[26] .

A la hora de introducir esta clase se hace una concesión importante: el precio a pagar por no acoplar las *factories* de los objetos que extraerán o ingresarán datos a RTI DDS es en rendimiento, sin embargo no es tal diferencia como para plantear otra solución.

Para que los tipos sean legibles se está traduciendo el `Enum` original a `String`. El problema introducido es que los `switch` que antes hacían uso de enteros ahora utilizan una funcionalidad de Java 7 que permite hacer `switch` con `String`. Cuando el compilador traduce esto a *bytecode* se divide en dos etapas de `lookupswitch`, haciendo la ejecución más lenta[11].

Se podría crear un `Enum` propio y traducir de uno a otro o utilizar enteros directamente, pero se ha considerado que el impacto no es tanto como para obligar a sacrificar la simplicidad.

3.1.5 Ingreso de datos para DDS

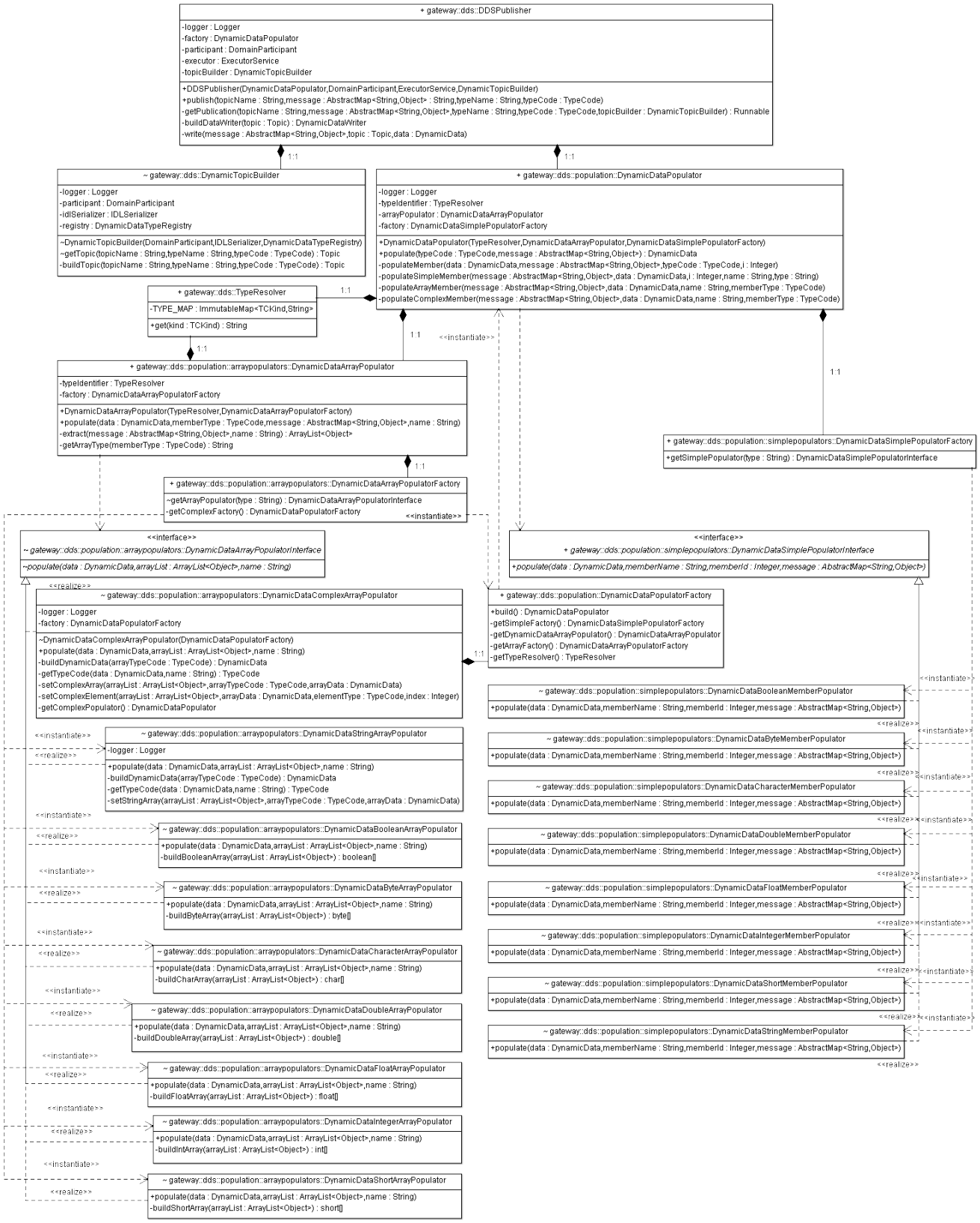

Figura 3.6 Diagrama UML del paquete de ingreso de datos DDS.

El propósito de este paquete es rellenar un objeto `DynamicData` en base a un *map* con los datos y un `TypeCode`.

3.1.5.1 DynamicDataPopulator

Responsabilidad

Es la clase principal que contiene la lógica para rellenar estructuras `DynamicData` con datos de un *map*. Tiene el control sobre cuál de las demás clases se encarga de rellenar qué miembros.

Visibilidad

Pública

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `typeIdentifier`: Instancia de `TypeResolver`.
- `arrayPopulator`: Instancia de `DynamicDataArrayPopulator`.
- `factory`: Instancia de `DynamicDataSimplePopulatorFactory`.

Métodos

Públicos

- `DynamicDataPopulator(TypeResolver, DynamicDataArrayPopulator, DynamicDataSimplePopulatorFactory)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `populate(TypeCode, AbstractMap<String, Object>)`: Instancia un nuevo objeto `DynamicData` con el `TypeCode` que se le pasa. Luego, itera por sus miembros y llama a `populateMember(DynamicData, AbstractMap<String, Object>, TypeCode, String, int)` pasándole el objeto `DynamicData`, el mensaje, el `TypeCode` obtenido para el miembro, el nombre del miembro y el índice. Devuelve el objeto `DynamicData` relleno.

Privados

- `populateMember(DynamicData, AbstractMap<String, Object>, TypeCode, String, int)`: Contiene la lógica de decisión sobre cómo rellenar cada uno de los tipos de miembros. Primero usa `typeIdentifier` para conseguir un `String` que represente al tipo y luego decide el método a llamar tal que:

Tabla 3.4 Métodos para ingresar datos en miembros de `DynamicData`.

Tipo(s)	Método
`complex`	`populateComplexMember(AbstractMap<String, Object>, DynamicData, String, TypeCode)`
`array`	`populateArrayMember(AbstractMap<String, Object>, DynamicData, String, TypeCode)`
Por defecto	`populateSimpleMember(AbstractMap<String, Object>, DynamicData, int, String, TypeCode)`

  Puede lanzar una excepción de tipo `BadKind`. No devuelve nada.

- `populateSimpleMember(AbstractMap<String, Object>, DynamicData, int, String, String)`: Llama a `factory.getSimplePopulator(String)` con el tipo para conseguir un `DynamicDataSimplePopulatorInterface`. Luego llama a `populate(DynamicData, String, int, AbstractMap<String, Object>` sobre él. No devuelve nada.

- `populateArrayMember(AbstractMap<String, Object>, DynamicData, String, TypeCode)`: Llama a `arrayPopulator.populate(DynamicData, TypeCode, AbstractMap<String, Object>, String`. Puede lanzar una excepción de tipo `BadKind`. No devuelve nada.

- `populateComplexMember(AbstractMap<String, Object>, DynamicData, String, TypeCode)`: Crea un nuevo objeto `DynamicData` para el miembro complejo anidado. Lo inicializa llamando a `populate(TypeCode, AbstractMap<String, Object>` (la clase es recursiva para miembros complejos hasta que los consigue descomponer en simples). Luego llama a `set_complex_member(String, int, DynamicData)` sobre el `DynamicData` original para rellenar el miembro. No devuelve nada.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `DynamicDataArrayPopulator`
- `DynamicDataSimplePopulatorFactory`

- `TypeResolver`

Externas

- **RTI DDS**: `DynamicData`, `BadKind`, `Bounds` y `TypeCode`.
- **Log4j**: `Logger`.
- **Guice**: `Inject`.
- **Gson**: `LinkedTreeMap`.

Detalles de interés

Al principio casi todo el comportamiento del paquete estaba en esta clase y definido mediante reflexión. La API de reflexión de Java es potente y ahorra código, pero hace que sea frágil (pueden surgir muchas excepciones y casos no considerados). La implementación con reflexión utiliza construcciones como esta:

Listado 3.3 Ejemplo con objeto Method.

```
1 Method method = data.getClass().getMethod(buildMethodName(type),new Class[]{String.
      class, int.class, memberClass});
2 element = method.invoke(data, new Object [] {info.member_name, info.member_id, value
      });
```

Que hacen muy probables que haya algún caso no considerado que haga que la aplicación deje de funcionar correctamente. Esta implementación generó muchos problemas hasta que se escribieron *tests* y el código se refactorizó para usar la estructura actual.

Así, progresivamente se fueron extrayendo responsabilidades, lo que resultó en más código pero más estable y más modular.

3.1.5.2 DynamicDataPopulatorFactory

Responsabilidad

Es una *factory* encargada de construir un `DynamicDataPopulator` cuando se necesite. Resuelve las dependencias de este construyéndolas.

Visibilidad

Pública

Métodos

Públicos

- `build()`: Devuelve una nueva instancia de `DynamicDataPopulator`.

Privados

- `getSimpleFactory()`: Devuelve una nueva instancia de `DynamicDataSimplePopulatorFactory`.
- `getDynamicDataArrayPopulator()`: Devuelve una nueva instancia de `DynamicDataArrayPopulator`.
- `getArrayFactory()`: Devuelve una nueva instancia de `DynamicDataArrayPopulatorFactory`.
- `getTypeResolver()`: Devuelve una nueva instancia de `TypeResolver`.

Dependencias

Internas

- `DynamicDataPopulator`
- `DynamicDataArrayPopulator`
- `DynamicDataSimplePopulatorFactory`
- `DynamicDataArrayPopulatorFactory`
- `TypeResolver`

Detalles de interés

Necesaria para rellenar *arrays* de estructuras complejas. Se inyecta para así poder rellenar las estructuras complejas con nuevas instancias de `DynamicDataPopulator`.

3.1.6 Ingreso de datos en *arrays*

La clase `DynamicData` provee una serie de *setters* para rellenar los miembros de tipo *array*[16]:

- `set_int_array`
- `set_short_array`
- `set_float_array`
- `set_double_array`
- `set_boolean_array`
- `set_char_array`
- `set_byte_array`
- `set_long_array`
- `set_complex_member`

Es decir, uno por cada uno de los principales tipos simples y `set_complex_member` para los demás casos.

Todas las clases encargadas de rellenar `arrays` están dentro del paquete `gateway.dds.population.complexpopulators`.

3.1.6.1 DynamicDataArrayPopulator

Responsabilidad

Es la clase encargada de rellenar los elementos de un `array`. Utiliza `DynamicDataArrayPopulatorFactory` para conseguir la clase que lo hará y luego llama `populate(DynamicData, ArrayList<Object>, String)` sobre ella.

Visibilidad

Pública

Atributos

Privados

- `typeIdentifier`: Instancia de `TypeResolver`.
- `factory`: Instancia de `DynamicDataArrayPopulatorFactory`.

Métodos

Públicos

- `DynamicDataArrayPopulator(TypeResolver, DynamicDataArrayPopulatorFactory)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `populate(DynamicData, TypeCode, AbstractMap<String, Object>, String)`: Llama a `extract(AbstractMap<String, Object>, String)` para extraer un `ArrayList<Object>` con los elementos del mensaje. Luego, usa `getArrayType(TypeCode)` para conseguir el tipo de los elementos del *array* en forma de `String`. Por último, consigue una instancia que implementa `DynamicDataArrayPopulatorInterface` sobre la que llama `populate(DynamicData, ArrayList<Object>, String)`. No devuelve nada.

Privados

- `extract(AbstractMap<String, Object>, String)`: Extrae el miembro del mensaje como un `ArrayList<Object>` y lo devuelve.
- `getArrayType(TypeCode)`: Llama a `typeIdentifier(TCKind)` con el `content_type()` extraído del *array*. Devuelve el tipo en forma de `String`.

Dependencias

Internas

- `DynamicDataArrayPopulatorFactory`
- `TypeResolver`

Externas

- **RTI DDS**: `DynamicData`, `BadKind` y `TypeCode`.

- **Guice**: `Inject`.

Detalles de interés

Antes de refactorizar, el comportamiento de todas las clases que implementan la interfaz `DynamicDataArrayPopulatorInterface` estaba en esta clase. Se extrajo la interfaz para poder separar la responsabilidad de rellenar cada tipo en distintas clases.

3.1.6.2 DynamicDataArrayPopulatorInterface

Responsabilidad

Es una interfaz que comparten todas las clases que se encargarán de rellenar los miembros de tipo *array*.

Visibilidad

Paquete

Métodos

Paquete

- `populate(DynamicData, ArrayList<Object>, String)`: Será el método que se llame para rellenar el campo del `DynamicData` proporcionado y el nombre dado con los datos presentes en el `ArrayList<Object>`. No devuelve nada.

Dependencias

Externas

- **RTI DDS**: `DynamicData`.

Clases que la implementan

- `DynamicDataBooleanArrayPopulator`
- `DynamicDataByteArrayPopulator`
- `DynamicDataCharacterArrayPopulator`
- `DynamicDataComplexArrayPopulator`

- DynamicDataDoubleArrayPopulator
- DynamicDataFloatArrayPopulator
- DynamicDataIntegerArrayPopulator
- DynamicDataShortArrayPopulator
- DynamicDataStringArrayPopulator

3.1.6.3 DynamicDataArrayPopulatorFactory

Responsabilidad

Se encarga de instanciar las distintas clases que implementan DynamicDataArrayPopulator-Interface.

Visibilidad

Pública

Métodos

Paquete

- getArrayPopulator(String): Instancia una de las implementaciones de DynamicDataArray-PopulatorInterface. En función del parámetro que recibe, instancia:

Tabla 3.5 Clases que permite instanciar DynamicDataArrayPopulatorFactory.

Tipo	Clase
int	DynamicDataIntegerArrayPopulator
short	DynamicDataShortArrayPopulator
float	DynamicDataFloatArrayPopulator
double	DynamicDataDoubleArrayPopulator
boolean	DynamicDataBooleanArrayPopulator
char	DynamicDataCharacterArrayPopulator
byte	DynamicDataByteArrayPopulator
string	DynamicDataStringArrayPopulator
complex	DynamicDataComplexArrayPopulator

Devuelve la instancia de la implementación concreta de `DynamicDataArrayPopulatorInterface`.

Privados

- `getComplexFactory()`: Instancia una nueva `DynamicDataPopulatorFactory` y la devuelve. Se necesita para instanciar `DynamicDataComplexArrayPopulator`.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`
- `DynamicDataBooleanArrayPopulator`
- `DynamicDataByteArrayPopulator`
- `DynamicDataCharacterArrayPopulator`
- `DynamicDataComplexArrayPopulator`
- `DynamicDataDoubleArrayPopulator`
- `DynamicDataFloatArrayPopulator`
- `DynamicDataIntegerArrayPopulator`
- `DynamicDataShortArrayPopulator`
- `DynamicDataStringArrayPopulator`
- `DynamicDataPopulatorFactory`

Detalles de interés

Al tener una interfaz común (`DynamicDataArrayPopulatorInterface`) se puede instanciar y devolver uno de muchos tipos dependiendo de un parámetro.

3.1.6.4 DynamicDataBooleanArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de booleanos.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `boolean[]` llamando a `buildBooleanArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_boolean_array(String, int, boolean[])`. No devuelve nada.

Privados

- `buildBooleanArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `boolean[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.5 DynamicDataByteArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de bytes.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `byte[]` llamando a `buildByteArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_byte_array(String, int, byte[])`. No devuelve nada.

Privados

- `buildByteArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `byte[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.6 DynamicDataCharacterArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de caracteres.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `char[]` llamando a `buildCharArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_char_array(String, int, char[])`. No devuelve nada.

Privados

- `buildCharArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `char[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.7 DynamicDataComplexArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de tipos complejos.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

- `factory`: Instancia de `DynamicDataPopulatorFactory`.

Métodos

Públicos

- `DynamicDataComplexArrayPopulator(DynamicDataPopulatorFactory)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

- `populate(DynamicData, ArrayList<Object>, String)`: Llama a `getTypeCode (DynamicData, String)` para conseguir el `TypeCode`. Luego llama a `buildDynamicData (TypeCode)` con el `TypeCode` en cuestión, que devuelve un `DynamicData` que representa el miembro de tipo *array* de `String`. Por último, llama a `setComplexArray(ArrayList <Object>, TypeCode, DynamicData)` y llama a `data.set_complex_member(String, int, Dy- namicData)` con el `DynamicData` que devuelve.

Privados

- `getTypeCode(DynamicData, String)`: Llama a `get_member_type(String, int)` sobre el `DynamicData`. Devuelve el `TypeCode`.

- `buildDynamicData(TypeCode)`: Instancia un nuevo `DynamicData` con el `TypeCode` que recibe.

- `setComplexArray(ArrayList<Object>, TypeCode, DynamicData)`: Consigue el `Type- Code` del contenido. Luego itera por los elementos del `DynamicData` llamando a `setComplex- Element(ArrayList<Object>, DynamicData, TypeCode, int)`, que rellenará cada elemento. Puede lanzar una excepción de tipo `BadKind`. No devuelve nada.

- `setComplexElement(ArrayList<Object>, DynamicData, TypeCode, int)`: Consigue un `DynamicDataPopulator` llamando a `getComplexPopulator()`. Luego, instancia un nuevo `LinkedTreeMap<String, Object>` en el que guarda el contenido que hay que introducir en el elemento en cuestión. Popula el elemento llamando a `populate(TypeCode,`

`AbstractMap<String, Object>`) sobre el objeto de tipo `DynamicDataPopulator` instanciado anteriormente, para a continuación llamar a `set_complex_member` para rellenar el miembro con los datos del `DynamicData` que esa llamada retorna. No devuelve nada.

- `getComplexPopulator()`: Llama a `factory.build()`. Devuelve el `DynamicDataPopulator`.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `DynamicDataPopulator`
- `DynamicDataPopulatorFactory`

Externas

- **RTI DDS**: `DynamicData`, `BadKind` y `TypeCode`.
- **Log4j**: `Logger`.
- **Guice**: `Inject`.
- **Gson**: `LinkedTreeMap`.

3.1.6.8 DynamicDataDoubleArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de `doubles`.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `double[]` llamando a `buildDoubleArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_double_array(String, int, double[])`. No devuelve nada.

Privados

- `buildDoubleArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `double[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.9 DynamicDataFloatArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de flotantes.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `float[]` llamando a `buildFloatArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_float_array(String, int, float[])`. No devuelve nada.

Privados

- `buildFloatArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `float[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.10 DynamicDataIntegerArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de flotantes.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `int[]` llamando a `buildIntArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_int_array(String, int, int[])`. No devuelve nada.

Privados

- `buildIntArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `int[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.11 DynamicDataShortArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de `shorts`.

Interfaces

- `DynamicDataArrayPopulatorInterface`

Visibilidad

Paquete

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Sobrescribe al método `populate` definido en `DynamicDataArrayPopulatorInterface`. Consigue un `short[]` llamando a `buildShortArray(ArrayList<Object>)` y a continuación popula el miembro llamando a `data.set_short_array(String, int, short[])`. No devuelve nada.

Privados

- `buildShortArray(ArrayList<Object>)`: Itera sobre el objeto `ArrayList<Object>` y rellena los elementos de un `short[]`, que devuelve.

Dependencias

Internas

- `DynamicDataArrayPopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.6.12 DynamicDataStringArrayPopulator

Responsabilidad

Se encarga de rellenar *arrays* de `String`.

Visibilidad

Paquete

Interfaces

- `DynamicDataArrayPopulatorInterface`

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

Métodos

Públicos

- `populate(DynamicData, ArrayList<Object>, String)`: Llama a `getTypeCode (DynamicData, String)` para conseguir el `TypeCode`. Luego llama a `buildDynamicData(TypeCode)` con el `TypeCode` en cuestión, que devuelve un `DynamicData` que representa el miembro de tipo *array* de `String`. Por último, llama a `setStringArray(ArrayList <Object>, TypeCode, DynamicData)` y llama a `data.set_complex_member(String, int, DynamicData)` con el `DynamicData` que devuelve.

Privados

- `getTypeCode(DynamicData, String)`: Llama a `get_member_type(String, int)` sobre el `DynamicData`. Devuelve el `TypeCode`.
- `buildDynamicData(TypeCode)`: Instancia un nuevo `DynamicData` con el `TypeCode` que recibe.
- `setStringArray(ArrayList<Object>, TypeCode, DynamicData)`: Itera por los elementos del objeto `DynamicData` y llama a `set_string(String, int, String)` con los elementos del `ArrayList<Object>`. Puede lanzar una excepción de tipo `BadKind`.

Dependencias

Internas

- `ApplicationLoggerFactory`

Externas

- **RTI DDS**: `DynamicData`, `BadKind` y `TypeCode`.

- **Log4j**: `Logger`.

3.1.7 Ingreso de datos en miembros simples

La clase `DynamicData` provee una serie de *setters* para rellenar los miembros de tipo simple[16]:

- `set_int`
- `set_short`
- `set_float`
- `set_double`
- `set_boolean`
- `set_char`
- `set_byte`
- `set_long`
- `set_string`

Todas las clases encargadas de rellenar miembros simples están dentro del paquete `gateway.dds.population.simplepopulators`.

3.1.7.1 DynamicDataSimplePopulatorInterface

Responsabilidad

Es una interfaz que comparten todas las clases que se encargarán de rellenar miembros simples.

Visibilidad

Pública

Métodos

Paquete

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Será el método que se llame para rellenar el campo del `DynamicData` proporcionado, el nombre dado y el identificador dado. No devuelve nada.

Dependencias

Externas

- **RTI DDS**: `DynamicData`.

Clases que la implementan

- `DynamicDataBooleanMemberPopulator`
- `DynamicDataByteMemberPopulator`
- `DynamicDataCharacterMemberPopulator`
- `DynamicDataDoubleMemberPopulator`
- `DynamicDataFloatMemberPopulator`
- `DynamicDataIntegerMemberPopulator`
- `DynamicDataShortMemberPopulator`
- `DynamicDataStringMemberPopulator`

3.1.7.2 DynamicDataSimplePopulatorFactory

Responsabilidad

Se encarga de instanciar las distintas clases que implementan `DynamicDataSimplePopulatorInterface`.

Visibilidad

Pública

Métodos

Paquete

- `getSimplePopulator(String)`: Instancia una de las implementaciones de `DynamicDataSimplePopulatorInterface`. En función del parámetro que recibe, instancia:

Tabla 3.6 Clases que permite instanciar `DynamicDataSimplePopulatorFactory`.

Tipo	Clase
`int`	`DynamicDataIntegerMemberPopulator`
`short`	`DynamicDataShortMemberPopulator`
`float`	`DynamicDataFloatMemberPopulator`
`double`	`DynamicDataDoubleMemberPopulator`
`boolean`	`DynamicDataBooleanMemberPopulator`
`char`	`DynamicDataCharacterMemberPopulator`
`byte`	`DynamicDataByteMemberPopulator`
`string`	`DynamicDataStringMemberPopulator`

Devuelve la instancia de la implementación concreta de `DynamicDataSimplePopulatorInterface`.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`
- `DynamicDataBooleanMemberPopulator`
- `DynamicDataByteMemberPopulator`
- `DynamicDataCharacterMemberPopulator`
- `DynamicDataDoubleMemberPopulator`
- `DynamicDataFloatMemberPopulator`
- `DynamicDataIntegerMemberPopulator`
- `DynamicDataShortMemberPopulator`
- `DynamicDataStringMemberPopulator`

Detalles de interés

Al tener una interfaz común (`DynamicDataSimplePopulatorInterface`) se puede instanciar y devolver uno de muchos tipos dependiendo de un parámetro.

3.1.7.3 DynamicDataBooleanMemberPopulator

Responsabilidad

Se encarga de rellenar booleanos.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `boolean` llamando a `booleanValue()` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_boolean(String, int, boolean)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.4 DynamicDataByteMemberPopulator

Responsabilidad

Se encarga de rellenar bytes.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `byte` llamando a `byteValue()` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_byte(String, int, byte)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.5 DynamicDataCharacterMemberPopulator

Responsabilidad

Se encarga de rellenar caracteres.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `char` llamando a `charAt(0)` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_char(String, int, char)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.6 DynamicDataDoubleMemberPopulator

Responsabilidad

Se encarga de rellenar `doubles`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `double` llamando a `doubleValue()` sobre el miembro del mensaje y a continuación

popula el miembro llamando a `data.set_double(String, int, double)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.7 DynamicDataFloatMemberPopulator

Responsabilidad

Se encarga de rellenar flotantes.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `float` llamando a `floatValue()` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_float(String, int, float)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.8 DynamicDataIntegerMemberPopulator

Responsabilidad

Se encarga de rellenar enteros.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `int` llamando a `intValue()` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_int(String, int, int)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.9 DynamicDataShortMemberPopulator

Responsabilidad

Se encarga de rellenar `shorts`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Consigue un `short` llamando a `shortValue()` sobre el miembro del mensaje y a continuación popula el miembro llamando a `data.set_short(String, int, short)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.7.10 DynamicDataStringMemberPopulator

Responsabilidad

Se encarga de rellenar `String`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimplePopulatorInterface`

Métodos

Públicos

- `populate(DynamicData, String, int, AbstractMap<String, Object>)`: Sobrescribe al método `populate` definido en `DynamicDataSimplePopulatorInterface`. Popula el miembro llamando a `data.set_string(String, int, String)`. No devuelve nada.

Dependencias

Internas

- `DynamicDataSimplePopulatorInterface`

Externas

- **RTI DDS**: `DynamicData`

3.1.8 Procesamiento DDS

El propósito de este paquete es devolver un *map* con los datos contenidos en un objeto `DynamicData`.

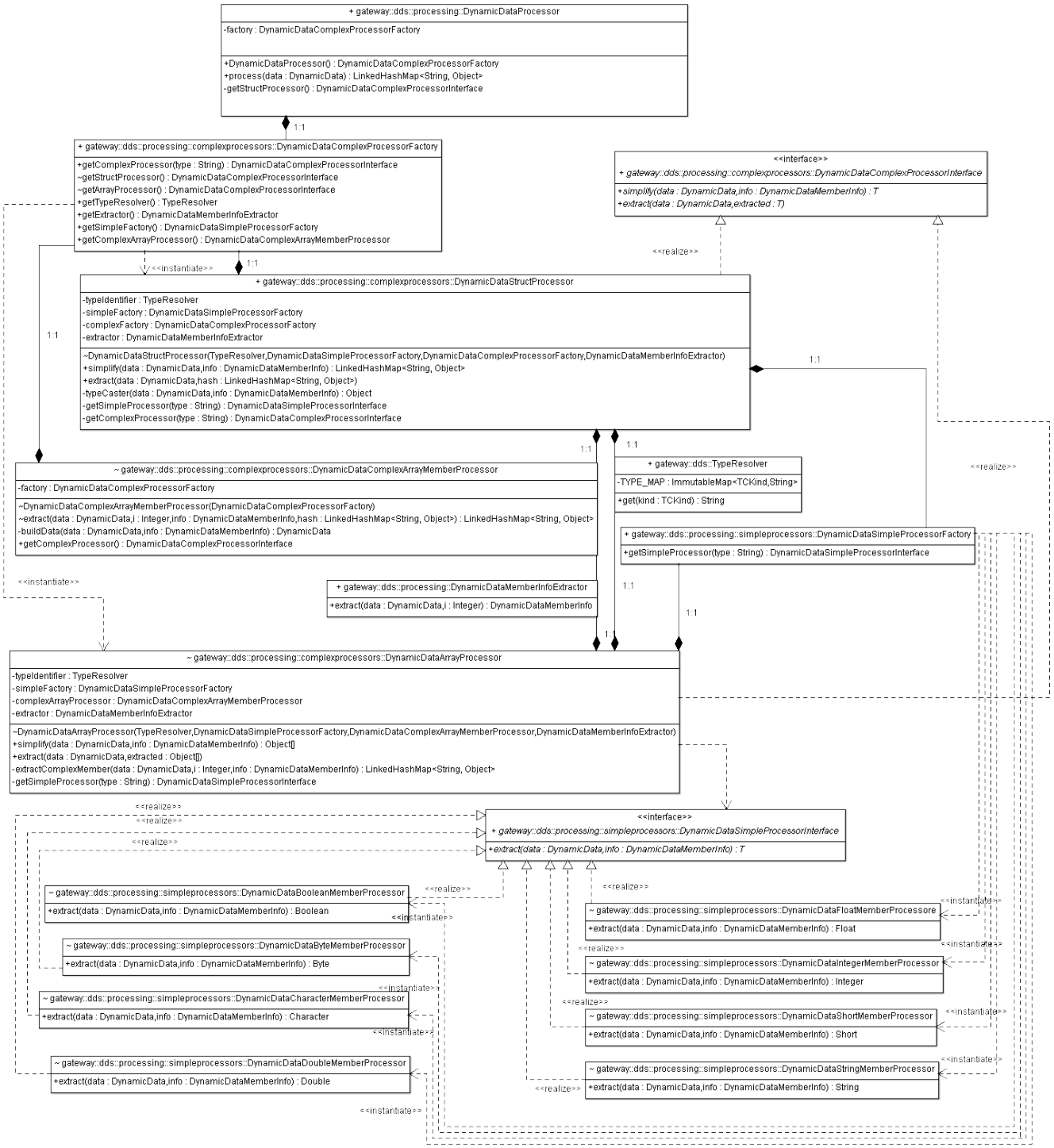

Figura 3.7 Diagrama UML del paquete de Procesamiento DDS.

3.1.8.1 DynamicDataProcessor

Responsabilidad

Es la clase principal que contiene la lógica para procesar estructuras `DynamicData` y extraer los datos a un *map*. Tiene el control sobre cuál de las demás clases se encarga de procesar qué miembros.

Visibilidad

Pública

Atributos

Privados

- `factory`: Instancia de `DynamicDataComplexProcessorFactory`.

Métodos

Públicos

- `DynamicDataProcessor(DynamicDataComplexProcessorFactory)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `process(DynamicData)`: Instancia un nuevo objeto `LinkedHashMap<String, Object>`. Luego, llama a `getStructProcessor()` para conseguir un procesador y llama `extract(DynamicData, LinkedHashMap<String, Object>)` sobre él. Devuelve el objeto `LinkedHashMap<String, Object>` populado.

Privados

- `getStructProcessor()`: Llama a `factory.getComplexProcessor(“complex”)` y devuelve una instancia que implementa `DynamicDataComplexProcessorInterface<LinkedHashMap<String, Object>>`.

Dependencias

Internas

- `DynamicDataComplexProcessorFactory`

- `DynamicDataComplexProcessorInterface`

Externas

- **RTI DDS**: `DynamicData`.
- **Guice**: `Inject`.

Detalles de interés

Al principio casi todo el comportamiento del paquete estaba en esta clase y definido mediante reflexión.

Al igual que en el paquete de población, progresivamente se fueron extrayendo responsabilidades.

3.1.8.2 DynamicDataMemberInfoExtractor

Responsabilidad

Una clase auxiliar que surgió de la necesidad de probar código de otras clases. Permite extraer un objeto `DynamicDataMemberInfo` a partir de un `DynamicData` y un índice entero.

Visibilidad

Pública

Métodos

Públicos

- `extract(DynamicData, int)`: Instancia un nuevo objeto `DynamicDataMemberInfo`. Luego, llama a `data.get_member_info_by_index(DynamicDataMemberInfo, int)` con esa instancia y el índice. Devuelve el `DynamicDataMemberInfo`.

Dependencias

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.9 Procesamiento de miembros complejos

En el paquete `gateway.dds.processing.complexprocessors` se reúnen todas las clases que permiten procesar los datos de miembros complejos (*structs* y *arrays*).

3.1.9.1 DynamicDataComplexProcessorInterface<T>

Responsabilidad

Es una interfaz que comparten algunas clases que se encargarán de procesar miembros complejos.

Visibilidad

Pública

Métodos

Paquete

- `simplify(DynamicData, DynamicDataMemberInfo`: Será el método que se llame para simplificar el miembro complejo. Devuelve la representación, de tipo `T`.
- `extract(DynamicData, T`: Será el método que se llame para extraer los datos y crear la representación. No devuelve nada.

Dependencias

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

Clases que la implementan

- `DynamicDataArrayProcessor`
- `DynamicDataStructProcessor`

3.1.9.2 DynamicDataComplexProcessorFactory

Responsabilidad

Se encarga de instanciar las distintas clases que implementan `DynamicDataComplexProcessorInterface<T>`.

Visibilidad

Pública

Métodos

Pública

- `getComplexProcessor(String)`: Instancia una de las implementaciones de `DynamicDataComplexProcessorInterface<T>`. En función del parámetro que recibe, llama a:

  Tabla 3.7 Clases que permite instanciar `DynamicDataComplexProcessorFactory`.

Tipo	Clase
`array`	`getArrayProcessor()`
`complex`	`getStructProcessor()`

  Devuelve la instancia de la implementación concreta de `DynamicDataComplexProcessorInterface<T>`.

Paquete

- `getArrayProcessor()`: Instancia un nuevo `DynamicDataArrayProcessor` y lo devuelve como `DynamicDataComplexProcessorInterface<Object[]>`.
- `getStructProcessor()`: Instancia un nuevo `DynamicDataStructProcessor` y lo devuelve como `DynamicDataComplexProcessorInterface<LinkedHashMap<String, Object>>`.

Privados

- `getTypeResolver()`: Instancia un nuevo `TypeResolver` y lo devuelve.
- `getExtractor()`: Instancia un nuevo `DynamicDataMemberInfoExtractor` y lo devuelve.

- `getSimpleFactory()`: Instancia un nuevo `DynamicDataSimpleProcessorFactory` y lo devuelve.
- `getComplexArrayProcessor()`: Instancia un nuevo `DynamicDataComplexArrayMemberProcessor` y lo devuelve.

Dependencias

Internas

- `DynamicDataComplexProcessorInterface<T>`
- `TypeResolver`
- `DynamicDataMemberInfoExtractor`
- `DynamicDataSimpleProcessorFactory`
- `DynamicDataComplexArrayMemberProcessor`
- `DynamicDataStructProcessor`
- `DynamicDataArrayProcessor`

Detalles de interés

Al tener una interfaz común (`DynamicDataComplexProcessorInterface<T>`) se puede instanciar y devolver uno de varios tipos dependiendo de un parámetro.

3.1.9.3 DynamicDataStructProcessor

Responsabilidad

Se encarga de rellenar estructuras complejas.

Visibilidad

Paquete

Interfaces

- `DynamicDataComplexProcessorInterface<LinkedHashMap<String,Object>>`.

Atributos

Privados

- `typeIdentifier`: Instancia de `TypeResolver`.
- `simpleFactory`: Instancia de `DynamicDataSimpleProcessorFactory`.
- `complexFactory`: instancia de `DynamicDataComplexProcessorFactory`.
- `extractor`: instancia de `DynamicDataMemberInfoExtractor`.

Métodos

Públicos

- `extract(DynamicData, LinkedHashMap<String, Object>)`: Sobrescribe al método `extract` definido en `DynamicDataComplexProcessorInterface<T>`. Itera sobre los miembros del `DynamicData`, luego instancia un nuevo `DynamicDataMemberInfo` usando `extractor` y usa `typeCaster(DynamicData, DynamicDataMemberInfo)` para procesar los miembros. Por último, lo coloca en el *map* que se le ha pasado. No devuelve nada.
- `simplify(DynamicData, DynamicDataMemberInfo)`: Crea un nuevo `DynamicData` y lo rellena con el miembro en cuestión. Luego instancia un `LinkedHashMap<String, Object>` y llama a `extract(DynamicData, LinkedHashMap<String, Object>)`. Devuelve el `LinkedHashMap<String, Object>`.

Paquete

- `DynamicDataStructProcessor(TypeResolver, DynamicDataSimpleProcessorFactory, DynamicDataComplexProcessorFactory, DynamicDataMemberInfoExtractor)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Privados

- `typeCaster(DynamicData, DynamicDataMemberInfo)`: Crea un nuevo `Object` llamado `member` y lo inicializa. Luego, si el miembro es complejo o un *array*, llama a `getComplexProcessor(String)` y luego llama a `simplify(DynamicData, DynamicDataMemberInfo)` sobre él y luego lo asigna a `member`. En otro caso llama a `getSimpleProcessor(String)` y luego a `extract(DynamicData, DynamicDataMemberInfo)` sobre él, asignando lo que devuelva a `member`. Devuelve `member`.

- `getSimpleProcessor(String)`: Llama a `simpleFactory.getSimpleProcessor(String)` y devuelve la instancia que implementa `DynamicDataSimpleProcessorInterface<T>`.
- `getComplexProcessor(String)`: Llama a `complexFactory.getComplexProcessor(String)` y devuelve la instancia que implementa `DynamicDataComplexProcessorInterface<T>`.

Dependencias

Internas

- `DynamicDataSimpleProcessorFactory`
- `DynamicDataComplexProcessorFactory`
- `DynamicDataSimpleProcessorInterface`
- `DynamicDataComplexProcessorInterface`
- `DynamicDataMemberInfoExtractor`
- `TypeResolver`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.
- **Guice**: `Inject`.

3.1.9.4 DynamicDataArrayProcessor

Responsabilidad

Se encarga de rellenar estructuras complejas.

Visibilidad

Paquete

Interfaces

- `DynamicDataComplexProcessorInterface<Object[]>`.

Atributos

Privados

- `typeIdentifier`: Instancia de `TypeResolver`.
- `simpleFactory`: Instancia de `DynamicDataSimpleProcessorFactory`.
- `complexArrayProcessor`: instancia de `DynamicDataComplexArrayMemberProcessor`.
- `extractor`: instancia de `DynamicDataMemberInfoExtractor`.

Métodos

Públicos

- `extract(DynamicData, Object[])`: Sobrescribe al método `extract` definido en `DynamicDataComplexProcessorInterface<T>`. Itera sobre los miembros del `DynamicData`, luego instancia un nuevo `DynamicDataMemberInfo` usando `extractor` y usa `typeIdentifier` para conseguir el tipo. Crea una instancia de `Object` llamada `element`, que se asignará al i-ésimo elemento del *array*. Si el tipo no es complejo, llama a `getSimpleProcessor(String)` y a `extract(DynamicData, DynamicDataMemberInfo)` sobre él, asignando el valor retornado a `element`. Si el tipo es complejo, se llama a `extractComplexMember(DynamicData, int, DynamicDataMemberInfo)`, asignando el valor retornado a `element`.
- `simplify(DynamicData, DynamicDataMemberInfo)`: Crea un nuevo `DynamicData` y lo popula con el miembro en cuestión. Luego instancia un `Object[]` y llama a `extract(DynamicData, Object[])`. Devuelve el `Object[]`.

Paquete

- `DynamicDataArrayProcessor(TypeResolver, DynamicDataSimpleProcessorFactory, DynamicDataComplexArrayMemberProcessor, DynamicDataMemberInfoExtractor)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Privados

- `extractComplexMember(DynamicData, int, DynamicDataMemberInfo)`: Crea una nueva instancia de `LinkedHashMap<String, Object>` y acto seguido llama a `complexArrayProcessor.extract(DynamicData, int, DynamicDataMemberInfo, LinkedHashMap<String, Object>)`, luego devuelve el `LinkedHashMap<String, Object>` poblado.

- `getSimpleProcessor(String)`: Llama a `simpleFactory.getSimpleProcessor(type)`, que devuelve una instancia que implementa `DynamicDataSimpleProcessorInterface<T>`.

Dependencias

Internas

- `DynamicDataSimpleProcessorFactory`
- `DynamicDataComplexProcessorInterface`
- `DynamicDataSimpleProcessorInterface`
- `DynamicDataComplexArrayMemberProcessor`
- `DynamicDataMemberInfoExtractor`
- `TypeResolver`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.
- **Guice**: `Inject`.

3.1.9.5 DynamicDataComplexArrayMemberProcessor

Responsabilidad

Se encarga de rellenar estructuras complejas miembros de *arrays*.

Visibilidad

Paquete

Atributos

Privados

- `factory`: Instancia de `DynamicDataComplexProcessorFactory`.

Métodos

Paquete

- `DynamicDataComplexArrayMemberProcessor(DynamicDataComplexProcessorFactory)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `extract(DynamicData, int, DynamicDataMemberInfo, LinkedHashMap<String, Object>)`: Crea un nuevo objeto `DynamicData` llamando a `buildData(DynamicData, DynamicDataMemberInfo)`. Luego llama a `data.get_complex_member(DynamicData, String, int)`, que popula la nueva instancia de `DynamicData`. Luego, llama a `getComplexProcessor()` para conseguir un procesador para tipos complejos, sobre el que se llamará a `extract(DynamicData, LinkedHashMap<String, Object>`. Devuelve el `LinkedHashMap<String, Object>`.

Privados

- `buildData(DynamicData, DynamicDataMemberInfo)`: Llama a `data.get_member_type(String, int)`, que devuelve un `TypeCode`. Luego usa el `TypeCode` para instanciar un nuevo `DynamicData`, que se devuelve.
- `getComplexProcessor()`: Llama a `factory.getStructProcessor()` y devuelve el `DynamicDataComplexProcessorInterface<LinkedHashMap<String, Object>>` resultante.

Dependencias

Internas

- `DynamicDataComplexProcessorFactory`
- `DynamicDataComplexProcessorInterface`

Externas

- **RTI DDS**: `DynamicData`, `TypeCode` y `DynamicDataMemberInfo`.
- **Guice**: `Inject`.

3.1.10 Procesamiento de miembros simples

En el paquete `gateway.dds.processing.simpleprocessors` se reúnen todas las clases que permiten procesar los datos de miembros simples.

La clase `DynamicData` provee una serie de *setters* para conseguir valores de los miembros de tipo simple[16]:

- `get_int`
- `get_short`
- `get_float`
- `get_double`
- `get_boolean`
- `get_char`
- `get_byte`
- `get_long`
- `get_string`

3.1.10.1 DynamicDataSimpleProcessorInterface<T>

Responsabilidad

Es una interfaz que comparten todas las clases que se encargarán de tomar los valores de los miembros simples.

Visibilidad

Pública

Métodos

Paquete

- `extract(DynamicData, DynamicDataMemberInfo)`: Será el método que se llame para extraer el campo del `DynamicData` proporcionado. Devuelve el campo de tipo `T`.

Dependencias

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

Clases que la implementan

- `DynamicDataBooleanMemberProcessor`
- `DynamicDataByteMemberProcessor`
- `DynamicDataCharacterMemberProcessor`
- `DynamicDataDoubleMemberProcessor`
- `DynamicDataFloatMemberProcessor`
- `DynamicDataIntegerMemberProcessor`
- `DynamicDataShortMemberProcessor`
- `DynamicDataStringMemberProcessor`

3.1.10.2 DynamicDataSimpleProcessorFactory

Responsabilidad

Se encarga de instanciar las distintas clases que implementan `DynamicDataSimpleProcessorInterface<T>`.

Visibilidad

Pública

Métodos

Paquete

- `getSimpleProcessor(String)`: Instancia una de las implementaciones de `DynamicDataSimpleProcessorInterface<T>`. En función del parámetro que recibe, instancia:

Tabla 3.8 Clases que permite instanciar `DynamicDataSimpleProcessorFactory`.

Tipo	Clase
`int`	`DynamicDataIntegerMemberProcessor`
`short`	`DynamicDataShortMemberProcessor`
`float`	`DynamicDataFloatMemberProcessor`
`double`	`DynamicDataDoubleMemberProcessor`
`boolean`	`DynamicDataBooleanMemberProcessor`
`char`	`DynamicDataCharacterMemberProcessor`
`byte`	`DynamicDataByteMemberProcessor`
`string`	`DynamicDataStringMemberProcessor`

Devuelve la instancia de la implementación concreta de `DynamicDataSimpleProcessorInterface<T>`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface<T>`
- `DynamicDataBooleanMemberProcessor`
- `DynamicDataByteMemberProcessor`
- `DynamicDataCharacterMemberProcessor`
- `DynamicDataDoubleMemberProcessor`
- `DynamicDataFloatMemberProcessor`
- `DynamicDataIntegerMemberProcessor`
- `DynamicDataShortMemberProcessor`
- `DynamicDataStringMemberProcessor`

Detalles de interés

Al tener una interfaz común (`DynamicDataSimpleProcessorInterface<T>`) se puede instanciar y devolver uno de muchos tipos dependiendo de un parámetro.

3.1.10.3 DynamicDataBooleanMemberProcessor

Responsabilidad

Se encarga de extraer booleanos.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Boolean>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un booleano llamando a `data.get_boolean(String, int)`. Devuelve el booleano.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.4 DynamicDataByteMemberProcessor

Responsabilidad

Se encarga de extraer bytes.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Byte>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un byte llamando a `data.get_byte(String, int)`. Devuelve el byte.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.5 DynamicDataCharacterMemberProcessor

Responsabilidad

Se encarga de extraer caracteres.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Character>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_char(String, int)`. Devuelve el carácter.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.6 DynamicDataDoubleMemberProcessor

Responsabilidad

Se encarga de extraer `doubles`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Double>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_double(String, int)`. Devuelve el `Double`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.7 DynamicDataFloatMemberProcessor

Responsabilidad

Se encarga de extraer flotantes.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Float>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_float(String, int)`. Devuelve el `Float`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.8 DynamicDataIntegerMemberProcessor

Responsabilidad

Se encarga de extraer enteros.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Integer>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_int(String, int)`. Devuelve el `Integer`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.9 DynamicDataShortMemberProcessor

Responsabilidad

Se encarga de extraer `shorts`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<Short>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_short(String, int)`. Devuelve el `Short`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.10.10 DynamicDataStringMemberProcessor

Responsabilidad

Se encarga de extraer `String`.

Visibilidad

Paquete

Interfaces

- `DynamicDataSimpleProcessorInterface<String>`

Métodos

Públicos

- `extract(DynamicData, DynamicDataMemberInfo)`: Sobrescribe al método `extract` definido en `DynamicDataSimpleProcessorInterface<T>`. Consigue un carácter llamando a `data.get_string(String, int)`. Devuelve el `String`.

Dependencias

Internas

- `DynamicDataSimpleProcessorInterface`

Externas

- **RTI DDS**: `DynamicData` y `DynamicDataMemberInfo`.

3.1.11 Serialización

En el paquete `gateway.dds.serialization` están contenidas las clases que serializan datos en el lado DDS.

3.1.11.1 DynamicDataSerializer

Responsabilidad

Permite serializar `DynamicData` o `AbstractMap<String, Object>` a JSON.

Visibilidad

Pública

Atributos

Privados

- `mqttName`: `String` que contiene el nombre del cliente MQTT.
- `processor`: Instancia de `DynamicDataProcessor`.
- `gson`: instancia de `Gson`.

Métodos

Públicos

- `DynamicDataProcessor(String, DynamicDataProcessor, Gson)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `serialize(DynamicData)`: Llama a `serialize(buildStructure(DynamicData))`.
- `serialize(AbstractMap<String, Object>)`: Llama a `gson.toJson(DynamicData)` y devuelve el valor.

Privados

- `buildStructure(DynamicData)`: Instancia un nuevo `LinkedHashMap<String, Object>`. Agrega los siguientes pares clave-valor:

Tabla 3.9 Pares clave-valor del mensaje MQTT.

Clave	Valor
`msg`	`processData(DynamicData)`
`cid`	`mqttName`
`type`	`dataType(DynamicData)`

Devuelve el `LinkedHashMap<String, Object>`.

- `dataType(DynamicData)`: Llama a `get_type().name()` sobre el objeto `DynamicData`. Captura las excepciones de tipo `BadKind` si ocurriesen.
- `processData(DynamicData)`: Llama a `processor.process(DynamicData)` y devuelve el resultado, que son los datos almacenados en un `LinkedHashMap<String, Object>`.

Dependencias

Internas

- **DynamicDataProcessor**

Externas

- **RTI DDS**: `DynamicData` y `BadKind`.
- **Guice**: `Inject`.
- **Gson**: `Gson`.

3.1.11.2 IDLSerializer

Responsabilidad

Devuelve la representación IDL de un `TypeCode` como `String`.

Visibilidad

Pública

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

Métodos

Públicos

- `toIDLString(TypeCode)`: Inicializa un nuevo `StringWriter`. Acto seguido, llama a `printIDL(int, StringWriter)` sobre el `TypeCode`. Por último, devuelve el resultado de lanzar `toString()` sobre el `StringWriter`.

Dependencias

Internas

- **ApplicationLoggerFactory**

Externas

- **RTI DDS**: `TypeCode`.
- **Log4j**: `Logger`.

3.1.12 *Logging*

En el paquete `gateway.logging` están las clases auxiliares para *logging*.

3.1.12.1 ApplicationLoggerFactory

Responsabilidad

Encapsula la instanciación de `Logger`.

Visibilidad

Pública

Métodos

Públicos

- `getLogger(String)`: Delega sobre `LogManager` para instanciar el `Logger` de nombre dado y lo devuelve. Tiene el modificador `static`.

Dependencias

Externas

- **Log4j**: `Logger` y `LogManager`.

Detalles de interés

La instanciación de los `Logger` está encapsulada en caso de que se decidiera cambiar de librería de *logging*.

3.1.13 Paquete MQTT

El paquete `gateway.mqtt` contiene las clases encargadas del lado MQTT de la comunicación. La mayoría de la información necesaria se encuentra en la documentación de Paho[23].

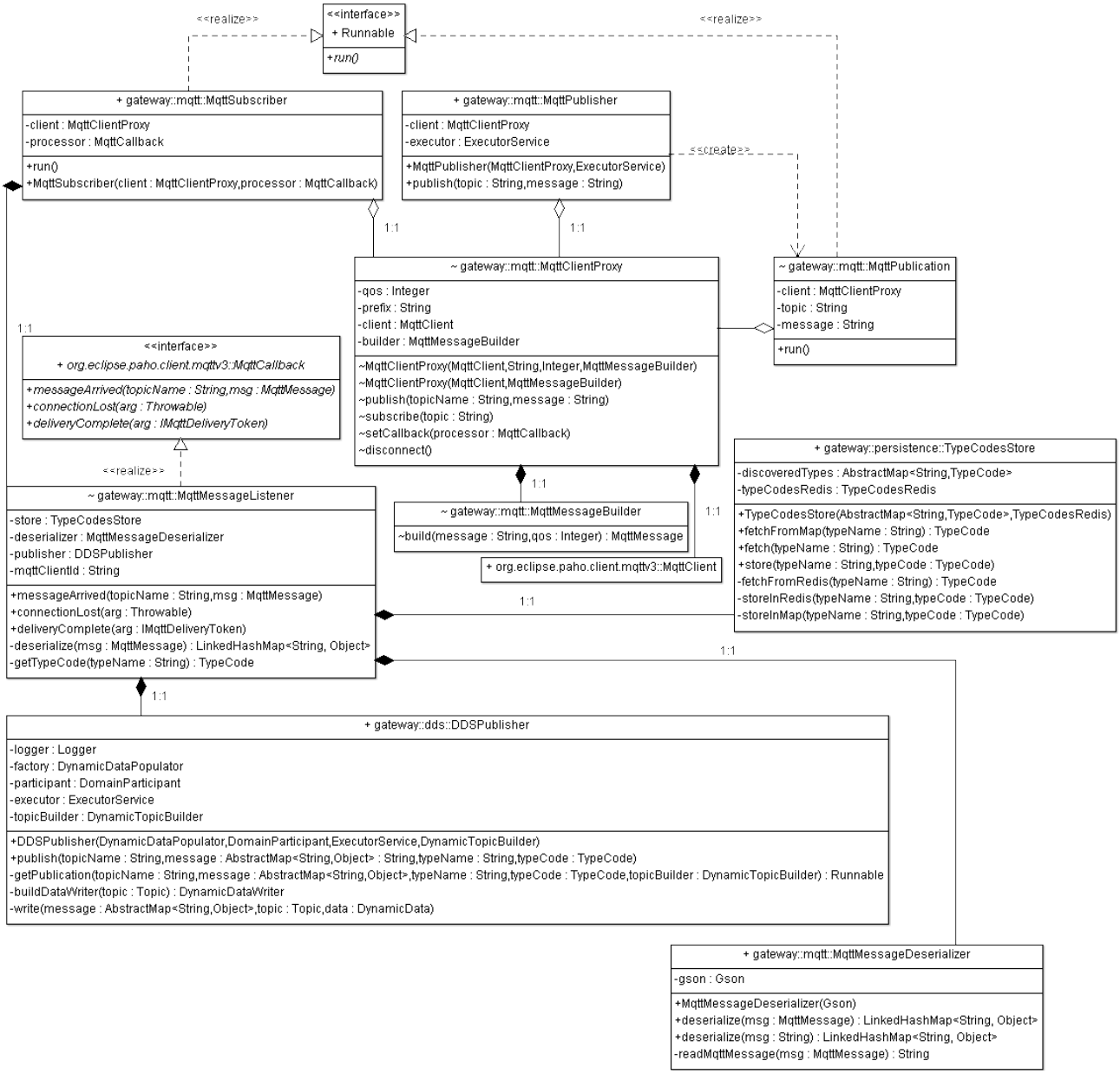

Figura 3.8 Diagrama UML del paquete MQTT.

3.1.13.1 MqttClientProvider

Responsabilidad

Provee instancias de `MqttClient`.

Visibilidad

Paquete

Interfaces

- `Provider<MqttClient>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `MqttClient`.

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.
- `persistence`: Instancia de `MemoryPersistence`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `MqttClient` utilizando `buildClient(String, String)` con la dirección construida con las propiedades obtenidas del fichero de configuración.

Paquete

- `MqttClientProvider(MemoryPersistence)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Privados

- `buildClient(String, String)`: Instancia un nuevo cliente utilizando el constructor `MqttClient(String, String, Persistence)`. Luego, instancia un objeto `MqttConnectOptions`, crea una sesión limpia y establece el nombre de usuario. Luego, conecta el cliente y lo devuelve.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `GeneralModule`: Propiedades (`mqtt.hostname`, `mqtt.port` y `mqtt.clientId`).

Externas

- **Paho MQTT**: `MqttClient`, `MqttConnectOptions`, `MqttException` y `MemoryPersistence`.
- **Guice**: `Inject` y `Provider`.
- **Log4j**: `Logger`.

3.1.13.2 MqttClientProxy

Responsabilidad

Encapsula un `MqttClient` y permite publicar, suscribir, anclar *callbacks* y desconectar. También permite establecer un prefijo.

Visibilidad

Paquete

Atributos

Privados

- `logger`: Instancia de `Logger`, perteneciente a `org.apache.logging.log4j`. Se inicializa pidiendo un `Logger` a `ApplicationLoggerFactory` con el nombre de la clase actual. Tiene los modificadores `static` y `final`.

- `qos`: Entero entre 0 y 2 que contiene el nivel de QoS.
- `prefix`: `String` que contiene el prefijo.
- `client`: Instancia de `MqttClient`.
- `builder`: Instancia de `MqttMessageBuilder`.

Métodos

Paquete

- `MqttClientProxy(MqttClient, String, int, MqttMessageBuilder)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `MqttClientProxy(MqttClient, MqttMessageBuilder)`: Llama al otro constructor con prefijo definido como `String` vacía y QoS 2.
- `publish(String, String)`: Usa `builder` para construir un objeto `MqttMessage`. Luego delega la publicación sobre el cliente añadiendo el prefijo.
- `subscribe(String)`: Delega la suscripción sobre el cliente añadiendo el prefijo.
- `setCallback(MqttCallback)`: Delega la llamada sobre el cliente.
- `disconnect()`: Delega la desconexión sobre el cliente.

Dependencias

Internas

- `ApplicationLoggerFactory`
- `MqttMessageBuilder`

Externas

- **Paho MQTT**: `MqttClient`, `MqttCallback`, `MqttException` y `MqttMessage`.
- **Guice**: `Inject`.
- **Log4j**: `Logger`.

3.1.13.3 MqttClientProxyProvider

Responsabilidad

Provee instancias de `MqttClientProxy`.

Visibilidad

Paquete

Interfaces

- `Provider<MqttClientProxy>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `MqttClientProxy`.

Atributos

Privados

- `client`: Instancia de `MqttClient`.
- `builder`: Instancia de `MqttMessageBuilder`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `MqttClientProxy` con los parámetros inyectados y las propiedades obtenidas del fichero de configuración.

Paquete

- `MqttClientProxyProvider(MqttClient, MqttMessageBuilder)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `MqttMessageBuilder`

- `MqttClientProxy`
- `GeneralModule`: Propiedades (`mqtt.prefix` y `mqtt.Qos`).

Externas

- **Paho MQTT**: `MqttClient`.
- **Guice**: `Inject` y `Provider`.

3.1.13.4 MqttMessageBuilder

Responsabilidad

Instancia un nuevo `MqttMessage` en base a un mensaje dado como `String` y un valor para QoS.

Visibilidad

Paquete

Métodos

Paquete

- `build(String, int)`: Inicializa un nuevo `MqttMessage`. Luego, llama a `setPayload(byte[])` con el contenido del mensaje en bytes sobre él y a `setQos(int)` con el valor de QoS. Devuelve el `MqttMessage`.

Dependencias

Externas

- **Paho MQTT**: `MqttMessage`.

3.1.13.5 MqttMessageDeserializer

Responsabilidad

Permite deserializar mensajes MQTT.

Visibilidad

Pública

Atributos

Privados

- `gson`: Instancia de `Gson`.

Métodos

Públicos

- `MqttMessageDeserializer(Gson)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `deserialize(MqttMessage)`: Llama a `readMqttMessage(MqttMessage)` para conseguir la representación en `String`. Luego, llama a `deserialize(String)` con esa representación. Devuelve el resultado de esa llamada.
- `deserialize(String)`: Crea un `LinkedHashMap<String, Object>`. Llama a `gson.fromJson(String, Class)` con el mensaje y la clase `LinkedHashMap`. Devuelve el `LinkedHashMap<String, Object>`.

Privados

- `readMqttMessage(MqttMessage)`: Llama a `toString()` sobre el mensaje, devuelve el `String`.

Dependencias

Externas

- **Paho MQTT**: `MqttMessage`.
- **Gson**: `Gson`.
- **Guice**: `Inject`.

3.1.13.6 MqttMessageListener

Responsabilidad

Es encargado de atender a los mensajes MQTT que llegan y procesarlos para que sean publicados en el lado DDS si fuese necesario.

Visibilidad

Paquete

Interfaces

- `MqttCallback`: Permite que el `MqttClient` llame a los métodos que implementa cuando ocurre cierto evento.

Atributos

Privados

- `store`: Instancia de `TypeCodesStore`.
- `deserializer`: Instancia de `MqttMessageDeserializer`.
- `mqttClientId`: Identificador del cliente MQTT.
- `publisher`: Instancia de `DDSPublisher`.

Métodos

Públicos

- `messageArrived(String, MqttMessage)`: Sobrescribe el método `messageArrived(String, MqttMessage)` definido por `MqttCallback`. Llama a `deserialize(MqttMessage)` para conseguir el mensaje en un `LinkedHashMap<String, Object>`. Luego, comprueba que el mensaje no lo haya enviado la misma aplicación. Busca el nombre del tipo en el mensaje para luego usar `getTypeCode(String)`. Si encuentra el `TypeCode`, usa el `publisher` para publicar el mensaje en el lado DDS. No devuelve nada.
- `connectionLost(Throwable)`: Parte de la interfaz `MqttCallback`. No es necesario implementarlo.

- `deliveryComplete(IMqttDeliveryToken)`: Parte de la interfaz `MqttCallback`. No es necesario implementarlo.

Paquete

- `MqttMessageListener(TypeCodesStore, MqttMessageDeserializer, String, DDSPublisher)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Privados

- `deserialize(MqttMessage)`: Delega sobre `deserializer` y devuelve el `LinkedHashMap<String, Object>`.
- `getTypeCode(String)`: Delega sobre `store` y devuelve el `TypeCode`.

Dependencias

Internas

- `TypeCodesStore`
- `MqttMessageDeserializer`
- `DDSPublisher`

Externas

- **Paho MQTT**: `MqttMessage`, `MqttCallback` y `IMqttDeliveryToken`.
- **RTI DDS**: `TypeCode`.
- **Gson**: `LinkedTreeMap`.
- **Guice**: `Inject`.

Detalles de interés

Implementar la interfaz permitiría cambiar una implementación por otra con facilidad y simplemente registrar el *callback*.

3.1.13.7 MqttMessageListenerProvider

Responsabilidad

Provee instancias de `MqttMessageListener`.

Visibilidad

Paquete

Interfaces

- `Provider<MqttMessageListener>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `MqttMessageListener`.

Atributos

Privados

- `store`: Instancia de `TypeCodesStore`.
- `deserializer`: Instancia de `MqttMessageDeserializer`.
- `publisher`: Instancia de `DDSPublisher`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `MqttMessageListener` con los parámetros inyectados y las propiedades obtenidas del fichero de configuración.

Paquete

- `MqttMessageListenerProvider(TypeCodesStore, MqttMessageDeserializer, DDSPublisher)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `TypeCodesStore`
- `MqttMessageDeserializer`
- `DDSPublisher`
- `MqttMessageListener`
- `GeneralModule`: Propiedades (`mqtt.clientId`).

Externas

- **Guice**: `Inject` y `Provider`.

3.1.13.8 MqttModule

Responsabilidad

Uno de los módulos de Guice que permite inyectar dependencias relacionadas con MQTT a las clases del programa. Se encarga de atar las dependencias al proveedor que debe construirlas.

Superclases

- `AbstractModule`: Es una clase abstracta de la que se puede extender y sobrescribir `configure()` para escribir las reglas del `Binder`.

Métodos

Protegidos

- `configure()`: Sobrescribe el método `configure()` de `AbstractModule`. No devuelve nada. Establece los siguientes *bindings*:

Tabla 3.10 *Bindings* de Guice en `MqttModule`.

Dependencia	Proveedor	Ámbito	Nombre
`MqttClientProxy`	`MqttClientProxyProvider`	*Eager Singleton*[5]	-
`MqttClient`	`MqttClientProvider`	*Eager Singleton*[6]	-
`Runnable`	`MqttSubscriber`	Instancia	mqttSubscriber
`MqttCallback`	`MqttMessageListenerProvider`	Instancia	-

Dependencias

Internas

- **Proveedores**: `MqttClientProxyProvider`, `MqttClientProvider` y `MqttMessageListenerProvider`.
- `MqttClientProxy`.
- `MqttClient`.
- `MqttSubscriber`.

Externas

- **MQTT Paho**: `MqttCallback` y `MqttClient`.
- **Guice**: `AbstractModule` y `Names`.

3.1.13.9 MqttPublication

Responsabilidad

Es una publicación asíncrona de un mensaje MQTT.

Visibilidad

Paquete

[5] Debe ser *eager singleton* para garantizar que hay un único cliente MQTT en la aplicación.
[6] Garantiza que hay un único subscriptor y/o publicador MQTT en la aplicación.

Interfaces

- `Runnable`: Se quiere ejecutar el contenido del método `run()` en un hilo.

Atributos

Privados

- `client`: Instancia de `MqttClientProxy`.
- `topic`: Nombre del *topic*.
- `message`: Mensaje a enviar.

Métodos

Públicos

- `run()`: Sobrescribe el método `run()` de `Runnable`. Llama a `publish(String, String)` sobre `client`.

Paquete

- `MqttPublication(MqttClientProxy, String, String)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `MqttClientProxy`

3.1.13.10 MqttPublisher

Responsabilidad

Permite ejecutar `MqttPublication` asíncronamente.

Visibilidad

Pública

Atributos

Privados

- `client`: Instancia de `MqttClientProxy`.
- `executor`: Instancia de `ExecutorService`, de tipo `ThreadPoolExecutor`.

Métodos

Públicos

- `MqttPublisher(MqttClientProxy, ExecutorService)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos. El parámetro `executor` está anotado con `@Named("fixedThreadPool")`.
- `publish(String, String)`: Utiliza `executor` para ejecutar una nueva instancia de `MqttPublication`. No devuelve nada.

Dependencias

Internas

- `MqttClientProxy`
- `MqttPublication`

Externas

- **Guice**: `Inject` y `Named`.

3.1.13.11 MqttSubscriber

Responsabilidad

Es la clase principal de la subscripción MQTT.

Visibilidad

Pública

Interfaces

- `Runnable`: Se quiere ejecutar el contenido del método `run()` en un hilo.

Atributos

Privados

- `client`: Instancia de `MqttClientProxy`.
- `processor`: Instancia de `MqttCallback`.

Métodos

Públicos

- `MqttSubscriber(MqttClientProxy, MqttCallback)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `run()`: Sobrescribe el método `run()` de `Runnable`. Llama a los métodos `setCallback (processor)` y `subscribe(“#”)`.

Dependencias

Internas

- `MqttClientProxy`

Externas

- **MQTT Paho**: `MqttCallback`.
- **Guice**: `Inject`.

Detalles de interés

Se decidió hacer `Runnable` las clases `DDSSubscriber` y `MqttSubscriber` para así facilitar una interfaz común a la clase que debiera llamarlos.

3.1.14 Paquete de persistencia

El paquete `gateway.persistence` se encarga de la lógica para persistir los `TypeCode` serializados en Redis como almacén clave-valor con una copia en memoria para evitar llamadas innecesarias.

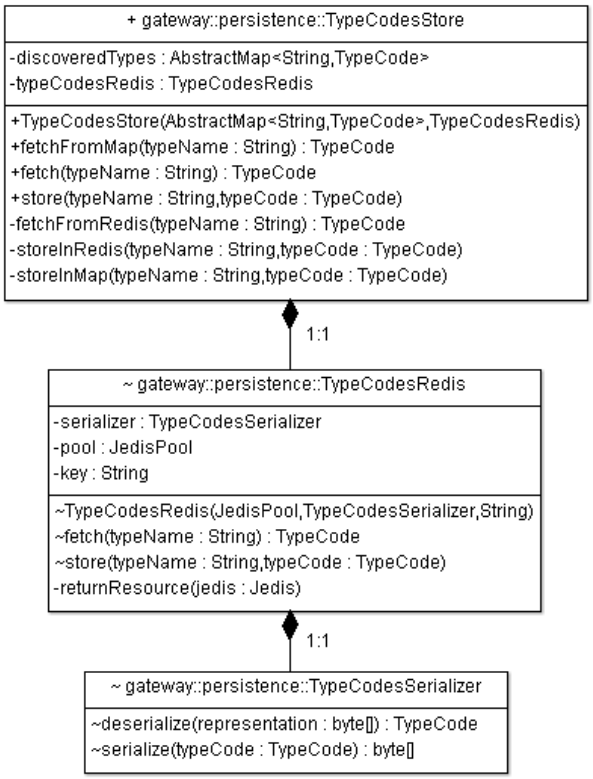

Figura 3.9 Diagrama UML del paquete de persistencia.

3.1.14.1 JedisPoolProvider

Responsabilidad

Provee instancias de `JedisPool`.

Visibilidad

Paquete

Interfaces

- `Provider<JedisPool>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `JedisPool`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `JedisPool` con opciones establecidas en el fichero de propiedades.

Dependencias

Internas

- `GeneralModule`: Propiedades (`redis.hostname`, `redis.port` y `redis.timeout`).

Externas

- **Jedis**: `JedisPool` y `JedisPoolConfig`.
- **Guice**: `Provider`.

3.1.14.2 RedisModule

Responsabilidad

Uno de los módulos de Guice que permite inyectar dependencias relacionadas con Redis a las clases del programa. Se encarga de atar las dependencias al proveedor que debe construirlas.

Visibilidad

Pública

Superclases

- `AbstractModule`: Es una clase abstracta de la que se puede extender y sobrescribir `configure()` para escribir las reglas del `Binder`.

Métodos

Protegidos

- `configure()`: Sobrescribe el método `configure()` de `AbstractModule`. No devuelve nada. Establece los siguientes *bindings*:

Tabla 3.11 *Bindings* de Guice en `RedisModule`.

Dependencia	Proveedor	Ámbito	Nombre
`JedisPool`	`JedisPoolProvider`	*Eager Singleton*[7]	-
`TypeCodesRedis`	`TypeCodesRedisProvider`	Instancia	-
`AbstractMap<String,TypeCode>`	`ConcurrentSkipListMap<String,TypeCode>`	*Eager Singleton*[8]	concurrent

Dependencias

Internas

- **Proveedores**: `JedisPoolProvider` y `TypeCodesRedisProvider`.
- `TypeCodesRedis`.

Externas

- **Guice**: `AbstractModule`, `Provides`, `TypeLiteral` y `Names`.
- **Jedis**: `JedisPool`.
- **RTI DDS**: `TypeCode`.

3.1.14.3 TypeCodesRedis

Responsabilidad

Es la clase encargada de persistir `TypeCodes` en Redis. Se usa la librería Jedis para ese propósito[21].

Visibilidad

Paquete

[7] Es necesario que exista un único *pool* de conexiones a Redis.
[8] Debe ser *eager singleton* para garantizar que se comparte en ambos sentidos de la comunicación.

Atributos

Privados

- `serializer`: Instancia de `TypeCodesSerializer`.
- `pool`: Instancia de `JedisPool`.
- `key`: Clave del *hash* de Redis donde se almacenarán los tipos.

Métodos

Paquete

- `TypeCodesRedis(JedisPool, TypeCodesSerializer, String)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.
- `fetch(String)`: Dentro de un bloque `try` toma un recurso de la *pool* de Jedis y si la clave correspondiente al tipo existe, extrae el `TypeCode` serializado y lo deserializa. En el bloque `finally` llama a `returnResource(Jedis)` con el recurso. Devuelve el `TypeCode`.
- `store(String, TypeCode)`: Dentro de un bloque `try` toma un recurso de la *pool* de Jedis y si la clave correspondiente al tipo no existe, almacena el `TypeCode` que recibe una vez serializado. En el bloque `finally` llama a `returnResource(Jedis)` con el recurso. No devuelve nada.

Privados

- `returnResource(Jedis)`: Si `Jedis` no es `null`, devuelve el recurso a la *pool*. No devuelve nada.

Dependencias

Internas

- `TypeCodesSerializer`

Externas

- **Jedis**: `Jedis` y `JedisPool`.
- **RTI DDS**: `TypeCode`.

- **Guice**: `Inject`.

Detalles de interés

Originalmente las clases `TypeCodesRedis`, `TypeCodesStore` y `TypeCodesSerializer` formaban parte del código de publicador y subscriptor DDS, pero se fueron extrayendo y dividiendo responsabilidades gradualmente. Primero se extrajo todo el bloque, luego se separó la responsabilidad de la serialización y por último la persistencia en memoria y en red.

Por otro lado, es muy importante tener el bloque `finally` en el que se devuelven los recursos tomados de la *pool*, ya que si no una vez agotados el programa quedaría bloqueado en el próximo intento de conexión esperando la liberación de algún recurso.

Hay otras librerías de Redis, como por ejemplo `redis-rb` en Ruby que hacen esta operación transparente al usuario. Sin embargo, tener control sobre los recursos nos permite más flexibilidad a la hora de trabajar con ellos.

3.1.14.4 TypeCodesRedisProvider

Responsabilidad

Provee instancias de `TypeCodesRedis`.

Visibilidad

Paquete

Interfaces

- `Provider<TypeCodesRedis>`: Definida por Guice. Por lo tanto sobrescribe el método `get()`, que devuelve un objeto del tipo `TypeCodesRedis`.

Atributos

Privados

- `jedisPool`: Instancia de `JedisPool`.
- `serializer`: Instancia de `TypeCodesSerializer`.

Métodos

Públicos

- `get()`: Sobrescribe al método `get()` de `Provider<T>`. Instancia un `TypeCodesRedis` con opciones establecidas en el fichero de propiedades.

Paquete

- `TypeCodesRedisProvider(JedisPool, TypeCodesSerializer)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos.

Dependencias

Internas

- `TypeCodesSerializer`
- `GeneralModule`: Propiedades (`redis.key`).

Externas

- **Jedis**: `JedisPool`.
- **Guice**: `Inject` y `Provider`.

3.1.14.5 TypeCodesSerializer

Responsabilidad

Es la clase encargada de serializar y deserializar `TypeCodes` en CDR para almacenarlos en Redis.

Visibilidad

Paquete

Métodos

Paquete

- `deserialize(byte[])`: Busca el orden de bytes correcto usando `NativeInterface`. Luego, crea un nuevo `CdrInputStream` con el *array* de bytes y el booleano que define el

orden. Por último, se usa `TypeCodeFactory.TheTypeCodeFactory.create_tc_from_stream(CdrInputStream)` para instanciar el `TypeCode` en cuestión y se devuelve.

- `serialize(TypeCode)`: Se busca el tamaño del `TypeCode` serializado llamando `get_serialized_size(0)` sobre él. Luego, se crea un `CdrOutputStream` con ese tamaño de búfer. A continuación se llama a `serialize(OutputStream)` sobre el `TypeCode` y se extrae la representación del flujo para devolverla.

Dependencias

Externas

- **RTI DDS**: `CdrInputStream`, `CdrOutputStream`, `TypeCode`, `TypeCodeFactory` y `NativeInterface`.

3.1.14.6 TypeCodesStore

Responsabilidad

Es la clase encargada de guardar los `TypeCodes` en memoria y llamar a `TypeCodesRedis` para persistir en Redis.

Visibilidad

Pública

Atributos

Privados

- `discoveredTypes`: Instancia de `AbstractMap<String, TypeCode>` donde se almacenan los tipos descubiertos en esta ejecución. Se espera recibir un `ConcurrentSkipListMap<String, TypeCode>`.

- `typeCodesRedis`: Instancia de `TypeCodesRedis`.

Métodos

Públicos

- `TypeCodesStore(AbstractMap<String, TypeCode>, TypeCodesRedis)`: Anotado con `@Inject`. Asigna los parámetros que recibe a sus respectivos atributos. El parámetro `discovered-Types` está anotado con `@Named("concurrent")`.
- `fetchFromMap(String)`: Devuelve el valor asociado al nombre del tipo dado en el mapa en memoria.
- `fetch(String)`: Devuelve el valor asociado al nombre del tipo dado en el mapa en memoria si lo encuentra. Si no, lo busca en Redis usando `fetchFromRedis(String)` y lo guarda en el *map* usando `storeInMap(String, TypeCode)`.
- `store(String, TypeCode)`: Guarda el `TypeCode` en memoria usando `storeInMap(String, TypeCode)` y luego en Redis usando `storeInRedis(String, TypeCode)`. No devuelve nada.

Privados

- `fetchFromRedis(String)`: Devuelve el valor asociado al nombre del tipo dado en el mapa en Redis delegando sobre `typeCodesRedis`.
- `storeInRedis(String, TypeCode)`: Delega sobre `typeCodesRedis` para guardar un `TypeCode` en Redis. No devuelve nada.
- `storeInMap(String, TypeCode)`: Guarda el `TypeCode` en memoria. No devuelve nada.

Dependencias

Internas

- `TypeCodesRedis`

Externas

- **RTI DDS**: `TypeCode`.
- **Guice**: `Inject` y `Named`.

Detalles de interés

Es importante en este caso que todas las instancias de la clase tengan la misma instancia de `ConcurrentSkipListMap<String, TypeCode>` en `discoveredTypes`, ya que se quiere que los tipos sean conocidos por toda la aplicación sin importar quién haya tenido conocimiento primero.

También es clave guardar las definiciones en memoria aunque estén en Redis para evitar llamadas innecesarias.

3.1.15 *Tests*

Para probar el código expuesto anteriormente, se han creado varias baterías de pruebas usando JUnit[27] y Mockito[28].

Para cada fichero se especificará cómo se preparan los *tests* y qué *tests* se realizan. Cada uno está anotado con `@Test`, y antes de ejecutarlos se ejecuta lo necesario para preparar la ejecución, anotado con `@Before`.

Se usará el término *mock* para referirse a objetos sustitutos de clases reales creados usando la librería Mockito.

Mockito también permite realizar verificaciones sobre los *mocks*. Es decir, comprobar que un método del *mock* se ha invocado y con qué parámetros. Eso ayuda a probar el código de forma aislada, sin necesidad de que el resto de clases tengan que estar configuradas correctamente.

3.1.15.1 TestSuite

Define una *suite* de *tests* que lanza todos los definidos en las clases que se declaran dentro de `@Suite.SuiteClasses`.

3.1.15.2 EntryPointTest

Pruebas de la clase `EntryPoint`. Esta clase ejecuta dos `Runnable` usando un `ExecutorService`.

Para preparar las pruebas habrá que crear *mocks* de ambos `Runnable` y del `ExecutorService`. Además, será necesario crear un `ArgumentCaptor<Runnable>`, que permitirá capturar los `Runnable` que lleguen al ejecutor falso y lanzarlos[28].

Pruebas realizadas

JUnit Test 1 (Ejecución de `Runnable`) *Se prueba que el ejecutor se llama pasándole dos veces un `Runnable`.*
JUnit Test 2 (Ejecución de `ddsSubscriber` y `mqttSubscriber`) *Se prueba que son `ddsSubscriber` y `mqttSubscriber` los `Runnable` que se ejecutan.*

3.1.15.3 DDSPublisherTest

Pruebas de la clase `DDSPublisher`. Esta clase recibe mensajes junto a sus tipos y *topics* y los publica en el lado DDS.

Para preparar las pruebas habrá que crear *mocks* de `DomainParticipant`, `Topic`, `DynamicDataWriter`, `DynamicData`, `DynamicDataPopulator`, `IDLSerializer`, `ExecutorService`, `DynamicDataTypeRegistry` y `DynamicTopicBuilder`. Como tiene comportamiento asíncrono, también será necesario crear un `ArgumentCaptor<Runnable>`, que permitirá capturar los `Runnable` que lleguen al ejecutor falso y lanzarlos.

Pruebas realizadas

JUnit Test 3 (Carácter asíncrono) *Se prueba que el ejecutor se llama pasándole un `Runnable` cuando se llama a `publish(String, AbstractMap<String, Object>, String, TypeCode)`.*
JUnit Test 4 (Obtención del `Topic`) *Se verifica que se usa `topicBuilder` para conseguir el topic al que hay que publicar.*
JUnit Test 5 (Obtención del `DataWriter`) *Se verifica que se crea un `DataWriter` para el topic que se había obtenido.*
JUnit Test 6 (Creación del objeto `DynamicData`) *Se verifica que se usa el `DynamicDataPopulator` para construir un objeto `DynamicData` a partir del mensaje.*

3.1.15.4 DynamicTopicBuilderTest

Pruebas de la clase `DynamicTopicBuilder`. Esta clase busca `Topics` existentes o instancia nuevos.

Para preparar las pruebas habrá que crear *mocks* de `DomainParticipant`, `Topic`, `IDLSerializer` y `DynamicDataTypeRegistry`. También se crea un `TypeCode`.

Pruebas realizadas

JUnit Test 7 (Obtención del `Topic`) *Se verifica que se llama `lookup_topicdescription(String)` sobre el participante al llamar a `getTopic(String, String, TypeCode)`.*
JUnit Test 8 (Registra el `Topic`) *Se verifica que se usa `registry` para registrar el `Topic`.*
JUnit Test 9 (Instancia un nuevo `Topic` si no se encuentra) *Se verifica que se usa `create_topic(String, String, TopicQos, TopicListener, int)` para instanciar un nuevo `Topic`.*

3.1.15.5 DynamicTopicProcessorTest

Pruebas de la clase `DynamicTopicProcessor`. Esta clase procesa los nuevos *topics* DDS y crea `DataReaders` para ellos.

Para preparar las pruebas habrá que crear *mocks* de `DomainParticipant`, `TypeCodesStore`, `DDSListener`, `Topic` y `DynamicTopicBuilder`. También se crea un `TypeCode`.

Pruebas realizadas

JUnit Test 10 (Toma el tipo de memoria) *Verifica que se usa* `store` *para extraer el tipo de memoria.*
JUnit Test 11 (Almacena el tipo si no se encuentra en memoria) *Verifica que se usa* `store` *para almacenar el tipo en memoria y Redis.*
JUnit Test 12 (Crea el `Topic`**)** *Verifica que se usa* `topicBuilder` *si no se encuentra el tipo.*
JUnit Test 13 (Crea el `DataReader`**)** *Verifica que se llama a* `create_datareader(TopicDescription, DataReaderQos, DataReaderListener, int)` *sobre el participante.*
JUnit Test 14 (No crea el `Topic` **si está registrado)** *Verifica que no se crea el* `Topic` *si ya se ha registrado previamente.*
JUnit Test 15 (No crea el `DataReader` **si está registrado)** *Verifica que no se crea el* `DataReader` *si el* `Topic` *ya se ha registrado previamente.*

3.1.15.6 DynamicDataPopulatorTest

Pruebas del proceso de población de `DynamicData`.

Para cada prueba habrá que crear un `TypeCode` correspondiente con la estructura de datos que se quiere. Son pruebas de integración, así que las clases son las reales.

Pruebas realizadas

JUnit Test 16 (Población de estructuras) *Verifica que se pueden rellenar estructuras.*

La estructura probada es:

Listado 3.4 JSON de estructura.

```
1 {
2   "im":"WOOHOO",
3   "inside":30
4 }
```

JUnit Test 17 (Población de estructuras anidadas) *Verifica que se pueden rellenar estructuras anidadas.*

La estructura probada es:

Listado 3.5 JSON de estructura anidada.

```
1 {
2   "color":"BLUE",
3   "x":100,
```

```
  "y":100,
  "z":[1,2,3,4],
  "shapesize":{
    "im":"WOOHOO",
    "inside":30
  }
}
```

JUnit Test 18 (Población de miembros simples) *Verifica que se pueden rellenar los distintos tipos de miembros simples.*

La estructura probada es:

Listado 3.6 JSON de estructura con miembros simples.

```
{
  "short":12,
  "long":1231241241,
  "ushort":13,
  "ulong":21312312,
  "float":30.12,
  "double":1231.231321,
  "boolean":true,
  "char":"A",
  "byte":127,
  "string":"blahblahblahblah"
}
```

JUnit Test 19 (Población de miembros *arrays*) *Verifica que se pueden rellenar los distintos tipos de arrays.*

Listado 3.7 JSON de estructura con *arrays*.

```
{
  "short":[12,13],
  "long":[123231,153231],
  "float":[30.1231,61.231],
  "double":[512381.1231,44442.111],
  "boolean":[true,false],
  "char":["A","B"],
  "byte":[127,-128]
}
```

JUnit Test 20 (Población de *arrays* de `String`) *Verifica que se pueden rellenar arrays de cadenas.*

La estructura probada es:

Listado 3.8 JSON de estructura con *arrays* de cadenas.

```
{
  "string":[
    "abc",
    "def",
    "ghi",
    "jkl",
    "mno"
  ]
}
```

JUnit Test 21 (Población de *arrays* de estructuras complejas) *Verifica que se pueden rellenar arrays de estructuras complejas.*

La estructura probada es:

Listado 3.9 JSON de estructura con *arrays* de estructuras complejas.

```
{
  "ca":[
    {
      "im":"WOOHOO",
      "inside":30
    },
    {
      "im":"WOOHOOY",
      "inside":32
    }
  ]
}
```

3.1.15.7 DynamicDataProcessorTest

Pruebas de procesamiento de `DynamicData`.

Para cada prueba habrá que crear un `TypeCode` correspondiente con la estructura de datos que se quiere. Son pruebas de integración, así que las clases son las reales.

Las pruebas realizadas son las siguientes, las mismas que en la población pero en el proceso inverso:

JUnit Test 22 (Procesamiento de estructuras) *Verifica que se pueden procesar estructuras.*

La estructura probada es:

Listado 3.10 JSON de estructura.

```
{
  "im":"WOOHOO",
  "inside":30
}
```

JUnit Test 23 (Procesamiento de estructuras anidadas) *Verifica que se pueden procesar estructuras anidadas.*

La estructura probada es:

Listado 3.11 JSON de estructura anidada.

```
{
  "color":"BLUE",
  "x":100,
  "y":100,
  "z":[1,2,3,4],
  "shapesize":{
    "im":"WOOHOO",
    "inside":30
  }
}
```

JUnit Test 24 (Procesamiento de miembros simples) *Verifica que se pueden procesar los distintos tipos de miembros simples.*

La estructura probada es:

Listado 3.12 JSON de estructura con miembros simples.

```
{
  "short":12,
  "long":1231241241,
  "ushort":13,
  "ulong":21312312,
  "float":30.12,
  "double":1231.231321,
  "boolean":true,
  "char":"A",
  "byte":127,
  "string":"blahblahblahblah"
}
```

JUnit Test 25 (Procesamiento de miembros *arrays*) *Verifica que se pueden procesar los distintos tipos de arrays.*

La estructura probada es:

Listado 3.13 JSON de estructura con *arrays*.

```
{
  "short":[12,13],
  "long":[123231,153231],
  "float":[30.1231,61.231],
  "double":[512381.1231,44442.111],
  "boolean":[true,false],
  "char":["A","B"],
  "byte":[127,-128]
}
```

JUnit Test 26 (Procesamiento de *arrays* de `String`) *Verifica que se pueden procesar arrays de cadenas.*

La estructura probada es:

Listado 3.14 JSON de estructura con *arrays* de cadenas.

```
{
  "string":[
    "abc",
    "def",
    "ghi",
    "jkl",
    "mno"
  ]
}
```

JUnit Test 27 (Procesamiento de *arrays* de estructuras complejas) *Verifica que se pueden procesar arrays de estructuras complejas.*

La estructura probada es:

Listado 3.15 JSON de estructura con *arrays* de estructuras complejas.

```
{
  "ca":[
    {
      "im":"WOOHOO",
      "inside":30
    },
    {
      "im":"WOOHOOY",
      "inside":32
    }
```

```
11   ]
12 }
```

3.1.15.8 DynamicDataSimpleMembersProcessorTest

Pruebas de procesamiento de miembros simples. Se prueban las clases que implementan `DynamicDataSimpleProcessorInterface<T>`.

Para preparar las pruebas habrá que crear *mocks* de `DynamicData` y `DynamicDataMemberInfo`.

Pruebas realizadas

JUnit Test 28 (Extracción de un booleano) *Verifica que se llama a `get_boolean(String, int)` con los parámetros correspondientes para extraer un booleano.*
JUnit Test 29 (Extracción de un byte) *Verifica que se llama a `get_byte(String, int)` con los parámetros correspondientes para extraer un byte.*
JUnit Test 30 (Extracción de un carácter) *Verifica que se llama a `get_char(String, int)` con los parámetros correspondientes para extraer un carácter.*
JUnit Test 31 (Extracción de un `double`) *Verifica que se llama a `get_double(String, int)` con los parámetros correspondientes para extraer un `double`.*
JUnit Test 32 (Extracción de un flotante) *Verifica que se llama a `get_float(String, int)` con los parámetros correspondientes para extraer un flotante.*
JUnit Test 33 (Extracción de un entero) *Verifica que se llama a `get_int(String, int)` con los parámetros correspondientes para extraer un entero.*
JUnit Test 34 (Extracción de un `short`) *Verifica que se llama a `get_short(String, int)` con los parámetros correspondientes para extraer un `short`.*
JUnit Test 35 (Extracción de un `String`) *Verifica que se llama a `get_string(String, int)` con los parámetros correspondientes para extraer un `String`.*

3.1.15.9 DynamicDataStructProcessorTest

Pruebas de procesamiento de estructuras.

Para preparar las pruebas habrá que crear *mocks* de `DynamicData`, `DynamicDataMemberInfo`, `DynamicDataMemberInfoExtractor`, `DynamicDataSimpleProcessorInterface<?>` y `LinkedHashMap`.

Pruebas realizadas

JUnit Test 36 (Extracción de un miembro de la estructura) *Verifica que el map se rellena con el valor en cuestión.*

JUnit Test 37 (Extracción de varios miembros de la estructura) *Verifica que el map se rellena con los valores en cuestión.*

3.1.15.10 DynamicDataArrayProcessorTest

Pruebas de procesamiento de *arrays*.

Para preparar las pruebas habrá que crear *mocks* de `DynamicData`, `DynamicDataMemberInfo`, `DynamicDataMemberInfoExtractor` y `DynamicDataSimpleProcessorInterface<?>`.

Pruebas realizadas

JUnit Test 38 (Extracción de miembros del array) *Verifica que el array resultante se popula con los valores en cuestión.*

3.1.15.11 DynamicDataSerializerTest

Pruebas de serialización de `DynamicData`, se prueba la clase `DynamicDataSerializer`.

Para preparar las pruebas habrá que crear *mocks* de `DynamicData`, `DynamicDataProcessor` y `TypeCode`.

Pruebas realizadas

JUnit Test 39 (Llamada al procesador) *Verifica que se llama al `DynamicDataProcessor` al serializar.*
JUnit Test 40 (Serializa con identificador de cliente y tipo) *Verifica que al serializar están presentes el nombre del tipo y el identificador cliente con el que se ha instanciado la clase.*
JUnit Test 41 (Permite serializar un *map* en lugar de un `DynamicData`) *Verifica que serializa correctamente.*

3.1.15.12 IDLSerializerTest

Pruebas de la clase `IDLSerializer`.

Para preparar las pruebas habrá que crear un *mock* de `TypeCode`.

Pruebas realizadas

JUnit Test 42 (Imprime IDL a `String`) *Verifica que se llama a `print_IDL` del `TypeCode` pasando un `StringWriter`.*

3.1.15.13 MqttClientProxyTest

Pruebas de la clase `MqttClientProxy`.

Para preparar las pruebas habrá que crear *mocks* de `MqttClient`, `MqttMessageBuilder` y `MqttMessageListener`.

Pruebas realizadas

JUnit Test 43 (Publica un mensaje en un *topic*) *Verifica que se delega sobre el cliente agregando un prefijo.*
JUnit Test 44 (Permite desconectar el cliente) *Verifica que se delega sobre el cliente.*
JUnit Test 45 (Permite agregar un *callback*) *Verifica que se delega sobre el cliente.*
JUnit Test 46 (Permite suscribirse a un *topic*) *Verifica que se delega sobre el cliente agregando un prefijo.*

3.1.15.14 MqttMessageBuilderTest

Pruebas de la clase `MqttMessageBuilder`.

Pruebas realizadas

JUnit Test 47 (Construye un mensaje MQTT correctamente)
JUnit Test 48 (Construye un mensaje MQTT con QoS correctamente)

3.1.15.15 MqttMessageDeserializerTest

Pruebas de la clase `MqttMessageBuilder`.

Para preparar las pruebas habrá que crear un *mock* de `MqttMessage`.

Pruebas realizadas

JUnit Test 49 (Deserializa un `MqttMessage`) *Verifica que el resultado coincide con lo esperado.*
JUnit Test 50 (Deserializa un `String`) *Verifica que el resultado coincide con lo esperado.*

3.1.15.16 MqttMessageListenerTest

Pruebas de la clase `MqttMessageListener`.

Para preparar las pruebas habrá que crear *mocks* de `LinkedHashMap`, `TypeCodesStore`, `MqttMessageDeserializer` y `DDSPublisher`.

Pruebas realizadas

JUnit Test 51 (Ignora el mensaje si el identificador del cliente coincide con el propio) *Comprueba que nunca se hace ningún procesado si eso ocurre.*
JUnit Test 52 (Busca el tipo en el almacén de `TypeCodes`) *Si corresponde procesar el mensaje, verifica que busca el tipo en `store`.*
JUnit Test 53 (Toma los distintos campos del *map*) *Verifica que se llama a `get(String)` sobre el mensaje para obtener los distintos elementos.*

3.1.15.17 MqttPublicationTest

Pruebas de la clase `MqttPublication`. Para preparar las pruebas habrá que crear un *mock* de `MqttClientProxy`.

Pruebas realizadas

JUnit Test 54 (Publica el mensaje usando el cliente) *Verifica que se llama a `publish(String, String)` sobre el cliente al invocar al método `run()`.*

3.1.15.18 MqttPublisherTest

Pruebas de la clase `MqttPublisher`.

Para preparar las pruebas habrá que crear *mocks* de `ThreadPoolExecutor` y `MqttClientProxy`.

Pruebas realizadas

JUnit Test 55 (Ejecuta una publicación) *Verifica que se llama a `execute(Runnable)` sobre el ejecutor al invocar al método `publish(String, String)`, siendo el `Runnable` un `MqttPublication`.*

3.1.15.19 MqttSubscriberTest

Pruebas de la clase `MqttPublisher`.

Para preparar las pruebas habrá que crear *mocks* de `MqttMessageListener` y `MqttClientProxy`.

Pruebas realizadas

JUnit Test 56 (Se suscribe a todos los *topics* y escucha los mensajes) *Verifica que se llama a* `setCallback(MqttCallback)` *con el* `MqttMessageListener` *y se suscribe a todos los canales usando el cliente.*

3.1.15.20 TypeCodesRedisTest

Pruebas de la clase `TypeCodesRedis`.

Para preparar las pruebas habrá que crear *mocks* de `JedisPool`, `Jedis`, `TypeCodesSerializer` y `TypeCode`.

Pruebas realizadas

JUnit Test 57 (Toma y liberación de recursos en `fetch`) *Verifica que se llama a* `getResource()` *y* `returnResource(Jedis)` *sobre el pool de Jedis.*
JUnit Test 58 (Toma y liberación de recursos en `store`) *Verifica que se llama a* `getResource()` *y* `returnResource(Jedis)` *sobre el pool de Jedis.*
JUnit Test 59 (Almacenamiento correcto) *Verifica que se llama a* `hset(byte[], byte[], byte[])` *sobre un* `Jedis` *con los parámetros correctos.*
JUnit Test 60 (Almacenamiento incorrecto) *Verifica que no se llama a* `hset(byte[], byte[], byte[])` *sobre un* `Jedis` *cuando la clave ya existe.*
JUnit Test 61 (Extracción correcta (resultado)) *Comprueba que el valor retornado por* `hget(byte[], byte[])` *es el que se devuelve.*
JUnit Test 62 (Extracción correcta (llamada)) *Verifica que se llama a* `hget(byte[], byte[])` *sobre un* `Jedis` *con los parámetros correctos si la clave existe.*
JUnit Test 63 (Extracción incorrecta) *Verifica que nunca se llama a* `hget(byte[], byte[])` *sobre un* `Jedis` *si la no clave existe.*

3.1.15.21 TypeCodesSerializerTest

Pruebas de la clase `TypeCodesSerializer`.

Pruebas realizadas

JUnit Test 64 (Comprueba la simetría de la serialización) *Serializa y deserializa un* `TypeCode` *y comprueba que el resultado es el mismo.*
JUnit Test 65 (Serialización correcta) *Serializa un* `TypeCode` *y comprueba que el resultado es el esperado.*

3.1.15.22 TypeCodesStoreTest

Pruebas de la clase `TypeCodesStore`.

Para preparar las pruebas habrá que crear *mocks* de `ConcurrentSkipListMap`, `TypeCodesRedis` y `TypeCode`.

Pruebas realizadas

JUnit Test 66 (Extracción de memoria (*miss*)) *Verifica que se extrae de memoria y no de `TypeCodesRedis` si se llama a `fetchFromMap(String)`, aun si no se encuentra.*
JUnit Test 67 (Extracción de memoria (*hit*)) *Verifica que se extrae de memoria y no de `TypeCodesRedis` si se llama a `fetchFromMap(String)`, si se encuentra.*
JUnit Test 68 (Extracción de Redis (llamada)) *Verifica que se extrae de `TypeCodesRedis` llamando a `fetch(String)` sobre él si no se encuentra en memoria.*
JUnit Test 69 (Extracción de Redis (resultado)) *Verifica que el resultado extraído de `TypeCodesRedis` se devuelve.*
JUnit Test 70 (Extracción de Redis (almacenamiento en memoria)) *Verifica que el resultado extraído de `TypeCodesRedis` se almacena en memoria.*
JUnit Test 71 (Almacenamiento en memoria) *Verifica que el `TypeCode` se almacena en memoria.*
JUnit Test 72 (Almacenamiento en Redis) *Verifica que el `TypeCode` se almacena en `TypeCodeRedis`.*

3.1.16 Librerías

A continuación se describirá brevemente qué uso se ha hecho de cada librería externa utilizada y de qué servicios dependen (si dependen de alguno).

Las dependencias se resuelven usando Maven cuando estén disponibles (todas menos RTI DDS)[25].

El fichero `pom.xml` describe dichas dependencias:

Listado 3.16 Contenido de `pom.xml`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mqttddsgateway</groupId>
  <artifactId>mqttddsgateway</artifactId>
  <version>0.0.2-SNAPSHOT</version>

  <properties>
    <mockito.version>1.9.5</mockito.version>
    <junit.version>4.11</junit.version>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <scope>test</scope>
      <version>${mockito.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <scope>test</scope>
```

```
35      <version>${junit.version}</version>
36    </dependency>
37    <dependency>
38      <groupId>org.hamcrest</groupId>
39      <artifactId>hamcrest-core</artifactId>
40      <version>1.3</version>
41    </dependency>
42    <dependency>
43      <groupId>com.google.code.gson</groupId>
44      <artifactId>gson</artifactId>
45      <version>2.2.4</version>
46    </dependency>
47    <dependency>
48      <groupId>org.apache.commons</groupId>
49      <artifactId>commons-pool2</artifactId>
50      <version>2.0</version>
51    </dependency>
52    <dependency>
53      <groupId>com.google.inject</groupId>
54      <artifactId>guice</artifactId>
55      <version>3.0</version>
56    </dependency>
57    <dependency>
58      <groupId>aopalliance</groupId>
59      <artifactId>aopalliance</artifactId>
60      <version>1.0</version>
61  </dependency>
62  <dependency>
63    <groupId>org.eclipse.paho</groupId>
64    <artifactId>mqtt-client</artifactId>
65    <version>0.4.0</version>
66  </dependency>
67    <dependency>
68      <groupId>redis.clients</groupId>
69      <artifactId>jedis</artifactId>
70      <version>2.5.2</version>
71    </dependency>
72    <dependency>
73      <groupId>com.google.guava</groupId>
74      <artifactId>guava</artifactId>
75      <version>16.0</version>
76    </dependency>
77    <dependency>
78      <groupId>org.apache.logging.log4j</groupId>
79      <artifactId>log4j-api</artifactId>
80      <version>2.2</version>
81    </dependency>
82    <dependency>
83      <groupId>org.apache.logging.log4j</groupId>
```

```
84      <artifactId>log4j-core</artifactId>
85      <version>2.2</version>
86    </dependency>
87  </dependencies>
88 </project>
```

3.1.16.1 Gson

Existen varias librerías para la serialización y deserialización JSON en Java, debido a lo extendida que está. Gson era la que permitía la serialización de estructuras arbitrarias de forma más cómoda para el propósito de la aplicación[22]. De esta librería se utilizan `Gson`, `GsonBuilder` y `LinkedTreeMap<K,V>`.

Descripción

Clases

- `Gson`: Clase principal para el uso de Gson, provee los metodos `toJson(Object)` y `fromJson(String, Class)`.
- `GsonBuilder`: Permite especificar opciones a la inicialización de `Gson`.
- `LinkedTreeMap<K,V>`: Clase hija de `AbstractMap<K,V>` que usa Gson para almacenar los objetos cuando deserializa de JSON.

3.1.16.2 Guava

Es una librería de utilidades desarrolladas por Google para Java. De esta librería se utiliza `ImmutableMap<K,V>`.

Descripción

Clases

- `ImmutableMap<K,V>`: Clase hija de `AbstractMap<K,V>`. Permite crear un *map* inmutable, de forma que se puede tener en un contexto estático y público y ser usado sin temor a que se sobrescriba su contenido o se agreguen parejas clave-valor.

3.1.16.3 Guice

Se usa Guice (o Google Inject) como *framework* para inyección de dependencias. Provee una manera de inyectar dependencias de forma sencilla para evitar acoplar el código y hacerlo difícil de mantener y de escribir *tests* para probarlo. Para ello, se crean uno o más módulos que definen el comportamiento de un inyector que permite instanciar objetos en la aplicación, resolviendo las dependencias en el proceso[10].

Existen otras alternativas, como Spring, pero Guice tiene toda la funcionalidad necesaria y es sencillo de implementar y mantener, y tiene una documentación extensa y fácil de entender[12].

De Guice se usan las anotaciones `@Inject`, `@Named` y `@Provides`; las clases `AbstractModule`, `TypeLiteral<T>` y `Names`; y la interfaz `Provider<T>`.

Para usar Guice hay que tener disponible las interfaces de la librería AOP Alliance.

Descripción

Anotaciones

- `@Inject`: Indica el constructor que debe utilizar Guice al inyectar una instancia de la clase como dependencia.
- `@Named`: Informa a Guice del nombre con el que está marcada la dependencia en sus *bindings*.
- `@Provides`: Crea un *binding* para el tipo de dato que devuelve el método anotado. Cuando el inyector necesite ese tipo, llamará al método.

Clases

- `AbstractModule`: Clase abstracta de la que se hereda para que la configuración de Guice sea más simple. Sólo hay y que sobrescribir el método `configure()` y añadir los *bindings* y proveedores que se desee.
- `TypeLiteral<T>`: Java no provee una manera de representar tipos genéricos. Esta clase es la forma de hacerlo.
- `Names`: Define métodos auxiliares para trabajar con dependencias anotadas con `@Named`.

Interfaces

- `Provider<T>`: Las clases que la implementan actúan como un proveedor del tipo T, que permite crear un *binding* usando `bind(Class).toProvider(Class)` en el método `configure()` del módulo.

3.1.16.4 Jedis

Se usa Jedis para persistir `TypeCodes` en Redis[21]. Por lo tanto, depende de una instancia de Redis ejecutándose. Se escoge Redis por permitir guardar distintos tipos de datos de forma cómoda, incluyendo *bytearrays*[19].

De Jedis se usan las clases `Jedis`, `JedisPool` y `JedisPoolConfig`.

Para usar `JedisPool` hay que tener disponible la librería Apache Commons Pool 2.

Descripción

Clases

- `Jedis`: Una instancia de esta clase permite ejecutar comandos de Redis. No es seguro usarlos en un entorno multihilo, así que se debe usar a través de un `JedisPool`, que sí se seguro.
- `JedisPool`: Un *pool* de recursos Jedis. Debe ser único en la aplicación y expone los métodos `getResource()` y `returnResource(Jedis)` para tomar y dejar recursos.
- `JedisPoolConfig`: Encapsula la configuración de Jedis.

3.1.16.5 JUnit

Se usa JUnit para *unit testing* o pruebas unitarias. Provee una serie de anotaciones y métodos que ayudan a probar el código[27].

De JUnit se usan las anotaciones `@Before`, `@Test`, `@Suite.SuiteClasses` y `@RunWith` y las clases `Assert` y `Suite`.

Descripción

Anotaciones

- `@Before`: Se ejecuta antes que una batería de pruebas para prepararlas.

- `@Test`: Marca una prueba.
- `@Suite.SuiteClasses`: Las clases que abarca serán parte de la *suite* de pruebas.
- `@RunWith`: Denota quién ejecutará las pruebas. El parámetro debe ser un `Runner`.

Clases

- `Assert`: Una colección de métodos que permiten asegurarse de que una condición se cumple en una prueba. Un ejemplo es `assertEquals(Object, Object)`.
- `Suite`: Una subclase de `Runner` que ejecuta las pruebas indicadas.

3.1.16.6 Log4j

Para *logging* se usa la librería Log4j[24] en lugar de `java.util.logging.Logger`, que es el *logger* por defecto de Java.

La configuración de *logging* se define en `log4j2.xml`. La estructura es la siguiente:

Listado 3.17 Configuración de logging.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="ALL">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="all">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

En ese fichero se definen:

- Los niveles de *logging*.
- Los *appenders*, o distintos lugares donde se escriben los *logs*.
- El patrón con el que se escriben los mensajes.
- Los niveles de *loggers* concretos.

En el entorno de *tests*, el nivel está definido como "off".

De esta librería se usan `Logger` y `LogManager`.

Descripción

Clases

- `Logger`: Ofrece una API cómoda para *logging*, con métodos como `trace(String)`, `debug(String)`, `info(String)`, `warn(String)`, `error(String)` y `fatal(String)`. Estos métodos permiten escribir en distintos niveles de gravedad, pudiendo así filtrar lo que se escribe en la configuración.

- `LogManager`: Expone el método `getLogger(String)`, que permite instanciar un nuevo `Logger` dado el nombre de este.

3.1.16.7 Mockito

Es una librería para la creación de *mocks*, su verificación y reemplazo de sus llamadas[28]. El uso de *mocks* hace que probar el código sea mucho más sencillo, ya que desacopla la prueba de un gran número de factores externos causados por el resto de clases de la aplicación.

De Mockito se usa las clases `Mockito`, `Mock` y `ArgumentCaptor<T>`, junto a las anotaciones `@Mock` y la interfaz `OngoingStubbing<T>` entre otras.

Necesita tener la librería Objenesis para funcionar.

Descripción

Anotaciones

- `@Mock`: Denota que el objeto es un sustituto del objeto original.

Clases

- `Mockito`: Hereda de `Matchers`. Es una colección de métodos que permiten crear *mocks*, verificar llamadas a métodos, falsificar el valor que devuelven, etc. Entre ellos están: `mock(Class)`, `verify(T)`, `never()`, `when(T)` y `times(int)`.

- `Mock`: Es un objeto falso cuyas llamadas a métodos se pueden verificar y falsificar.

- `ArgumentCaptor<T>`: Es una clase que permite almacenar uno o más argumentos de un tipo y luego retornarlos para hacer verificaciones u otras operaciones con ellos.

Interfaces

- `OngoingStubbing<T>`: El método `when(T)` de `Mockito` devuelve esta clase. Denota que se está falsificando la llamada al método de un `Mock`. Expone métodos como `thenReturn(T)`, que permite establecer el valor que devolverá el método cuando sea llamado. Así, el ejemplo completo sería `when(mock.toString()).thenReturn ("Representation")`, siendo mock un `Mock`.

3.1.16.8 Paho MQTT

Es una librería para comunicación usando MQTT[23]. Se usan las clases `MqttMessage`, `MqttClient`, `MqttConnectOptions`, `MemoryPersistence` y `MqttException`; y la interfaz `MqttCallback`. Para una explicación más detallada sobre MQTT, ver 2.1.1.1.

Descripción

Clases

- `MqttMessage`: Encapsula un mensaje MQTT, es decir, contenido, configuración de QoS, etc.
- `MqttClient`: Cliente MQTT, permite comunicarse con un *broker*.
- `MqttConnectOptions`: Opciones del cliente MQTT, por ejemplo sesión limpia activada o desactivada.
- `MemoryPersistence`: Persiste los mensajes MQTT en memoria, para evitar escrituras a disco cuando persiste en disco.
- `MqttException`: Excepción genérica cuando ocurre algún problema en MQTT.

Interfaces

- `MqttCallback`: Ofrece una interfaz común con *callbacks* que el cliente llamará cuando ocurra cierto evento, por ejemplo en la llegada de un mensaje llamará a `messageArrived(String, MqttMessage)`.

3.1.16.9 RTI Connext DDS

Es la librería elegida como implementación de DDS. Cuenta con una extensa documentación[16] y algunos ejemplos útiles[18]. Para una descripción de la gran mayoría de entidades usadas, ver 2.2.1.1.

3.2 Cliente REST (Kabuki)

Además de la aplicación principal, también se ha incluido un pequeño cliente REST escrito en Ruby, con un mínimo *frontend* en Javascript.

Esta aplicación expone cinco páginas accesibles desde el navegador, que son:

- `/app/index.html`
- `/admin/healthcheck`
- `/app/show_by_client.html`
- `/app/show_by_topic.html`
- `/app/show_by_type.html`

La primera permite enviar mensajes que correspondan con los tipos que ha recibido.

La segunda muestra el estado de los servicios necesarios para el correcto funcionamiento del cliente.

El resto muestran tablas paginadas con los mensajes recibidos, pero permiten filtrar por distintos campos (identificador del cliente, *topic* y tipo). Estos mensajes recibidos se almacenan en MongoDB.

Para esto se sirven de una API creada usando la librería Grape y un *frontend* creado usando JQuery, Dynatable y JSON Editor.

El reto principal en esta aplicación es el de guardar el esquema de los mensajes para poder reproducir la estructura al enviar un mensaje de dicho tipo usando el cliente. Esto se consigue creando un JSON *schema* a partir de nuevos tipos, almacenándolo y utilizándolo para definir qué estructura se debe enviar. Además de eso, se almacenarán un listado de *topics* asociados a cada tipo de mensaje.

3.2.1 Estructura

Su estructura en carpetas es la siguiente para el código en Ruby:

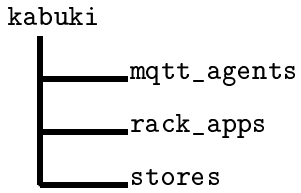

Figura 3.10 Estructura de directorios de la aplicación Ruby.

La aplicación está diseñada para hacer una inyección de dependencias simple con contenedores. Hay 5 contenedores: `AppTree`, `Config`, `MqttAgents`, `RackApps` y `Stores`.

Cada uno de estos contenedores define métodos que permiten instanciar objetos las distintas clases aplicación componiéndolos con sus dependencias.

Para cada clase en la que proceda se describirán:

- **Responsabilidad**
- **Atributos**
- **Superclases**
- **Módulos**[9]
- **Métodos en la *singleton class***[10]
  - **Públicos**
  - **Protegidos**
  - **Privados**
- **Métodos**

[9] En Ruby, un módulo permite incluir comportamiento en varias clases de forma simple. Esto crea una dependencia mucho más débil que la herencia. Es lo más parecido a una interfaz en Java, con la diferencia de que los módulos no son abstractos.

[10] Ruby permite añadir comportamiento a instancias de a clases. Así, también se puede extender el comportamiento de instancias de la clase `Class`. A esto se le llama *singleton class* o *eigenclass*.

  - **Públicos**
  - **Protegidos**
  - **Privados**

- **Dependencias**[11]
- **Detalles de interés**

[11] En lenguajes con tipos dinámicos, las dependencias no son tan vinculantes como en lenguajes de tipado estricto, ya que al no requerir tipos puede haber clases con interfaces parecidas que sean intercambiables. Esto se conoce como *duck typing*, siguiendo un razonamiento tal que: *Si es un pájaro que anda como un pato, nada como un pato y hace cuac como un pato, entonces para mí ese pájaro es un pato.*

3.2.2 Fichero de configuración de Rack

Ejecuta `Kabuki::App.app_tree.rack_apps.server`.

3.2.3 Módulo Kabuki

El fichero principal del módulo se encarga de permitir el autocargado de las clases pertenecientes al módulo.

Dependencias

- `ActiveSupport`
- `ActiveSupport::Inflector`

3.2.4 App

Responsabilidad

Iniciar la aplicación y el servidor web.

Métodos

Públicos

- `self.start!`: Método de clase que lanza el subscriptor MQTT y el servidor web.
- `self.app_tree`: Instancia memoizada de `AppTree`.

Privados

- `self.run`: Requiere Puma::CLI e inicia el servidor web. Esto bloquea la ejecución.

Dependencias

- `AppTree`
- `Puma::CLI`

3.2.5 AppTree

Responsabilidad

Contenedor raíz de la aplicación.

Métodos

Públicos

- `config`: Instancia el contenedor `Config` pasando a la instancia actual de este contenedor raíz.
- `stores`: Instancia el contenedor `Stores` pasando a la instancia actual de este contenedor raíz.
- `rack_apps`: Instancia el contenedor `RackApps` pasando a la instancia actual de este contenedor raíz.
- `mqtt`: Instancia el contenedor `MqttAgents` pasando a la instancia actual de este contenedor raíz.
- `mongo_session`: Se asegura de que la base de datos en Mongo está configurada usando el contenedor de configuración y luego pide la sesión a `Mongoid` llamando a `default_session`.
- `schema_builder`: Instancia `SchemaBuilder` pasando la instancia de `TopicStore` almacenada en el contenedor de `Stores`.

Dependencias

- `Config`
- `Stores`
- `RackApps`
- `MqttAgents`
- `Mongoid`
- `SchemaBuilder`

Detalles de interés

El sistema de contenedores no es una forma ideal de inyección de dependencias, pero debido a que Ruby tiene tipos dinámicos, el proceso es más complejo.

3.2.6 Cli

Responsabilidad

Ofrece una interfaz de línea de comandos que permite ejecutar los *tests*, iniciar el servidor web o una consola REPL[12].

Superclase

`Thor`: Parte de la gema `Thor`.

Métodos

Públicos

- `test`: Ejecuta los *tests* de RSpec.
- `start`: Ejecuta `App.start!`.
- `test`: Requiere `pry` y llama a `binding.pry`, que carga una consola REPL en el contexto actual.

Dependencias

- `App`
- `RSpec::Core::Runner`
- `Pry`
- `Thor`

[12]REPL son las iniciales de *read-eval-print-loop*, una consola interactiva que permite ejecutar código introducido por el usuario.

3.2.7 Config

Responsabilidad

Contenedor para configuración.

Atributos

- `app_tree`: Instancia del controlador raíz que se le pasa al inicializar.

Métodos

Públicos

- `initialize(app_tree)`: Instancia el contenedor y almacena el contenedor raíz en una variable de instancia.
- `config_file`: Dependiendo del entorno en que se ejecute la aplicación define el nombre del fichero de configuración.
- `config`: Toma la configuración del fichero de configuración obtenido de `config_file`.
- `server`: Parte de la configuración referente al servidor web.
- `redis`: Parte de la configuración referente a Redis.
- `database`: Parte de la configuración referente a MongoDB.
- `mqtt`: Parte de la configuración referente a MQTT.
- `ensure_database_configured!`: Requiere `Mongoid` y carga la configuración definida en `database`, luego crea índices[13] en la colección de mensajes y devuelve el estado de la configuración.

Dependencias

- `YAML`
- `Hashie::Mash`

[13] Un índice en bases de datos es una copia de una columna o columnas de la base de datos que permite buscar en ella de forma eficiente. Cuando una tabla o colección crece mucho, es necesario tener índices, o las consultas serán más costosas.

- Ficheros de configuración
- Mongoid
- Message
- AppTree

3.2.8 Initializable

Responsabilidad

Interfaz que define un constructor por defecto que asigna los atributos pasados como clave-valor de un *hash*.

Métodos

Públicos

- `initialize(args={})`: Llama a `set_attributes(args)`.

Privados

- `set_attributes(args)`: Por cada uno de los argumentos llama al *setter* correspondiente con el valor adecuado.

3.2.9 SchemaBuilder

Responsabilidad

Construye un esquema JSON a partir de un tipo y el esquema del mensaje recibido.

Módulos

- `Initializable`

Atributos

- `topic_store`: Instancia de `TopicStore` que se inyectará desde el contenedor.

Métodos

Públicos

- `self.build(type, msg_schema)`: Método de clase que extrae la clave "`$schema`" del esquema recibido. Construye un esquema preparado para el *frontend*, que lista los *topics* asociados al tipo usando `topic_store` y muestra los campos del mensaje.

3.2.10 Aplicaciones de Rack

3.2.10.1 Api

Responsabilidad

Define los diferentes *endpoints* que la API proveerá.

Superclases

`Grape::API`: Define un DSL[14]. Esta clase estará descrita de una forma distinta debido a este DSL.

Atributos

- `schema_store`: Atributo de clase. Instancia de `SchemaStore`.
- `publisher`: Atributo de clase. Instancia de `MqttPublisher`.
- `topic_store`: Atributo de clase. Instancia de `TopicStore`.
- `schema_builder`: Atributo de clase. Instancia de `SchemaBuilder`.

Métodos en la *singleton class*

Públicos

- `configure(&block)`: La clase se devuelve a sí misma al bloque que se le pasa.
- `attributes`: Devuelve todos los nombres de los atributos inyectables en la clase.

Helpers

Se define un *helper* por cada atributo para poder acceder a los atributos desde los *endpoints*.

Endpoints

GET `/api/v1.0/schemas.json`

No recibe parámetros. Usa `schema_store` para obtener un listado de los distintos esquemas de tipos. Devuelve 200.

[14] *Domain-specific language*, un lenguaje específico orientado a una aplicación concreta, en este caso definir REST APIs

GET `/api/v1.0/schemas/types/:type.json`

Recibe el tipo como parámetro en la URL. Busca el esquema correspondiente al tipo en `schema_store`.

Si encuentra el esquema, usa `schema_builder` para construir una respuesta y la devuelve con código 200.

Si no la encuentra, devuelve 404.

POST `/api/v1.0/messages.json`

Recibe tipo y mensaje. Si el tipo es conocido, usa `publisher` para publicar el mensaje, devolviendo 201 con respuesta JSON vacío.

Si no es un tipo conocido, devuelve 404.

GET `/api/v1.0/messages/types.json`

No recibe parámetros. Devuelve un listado de tipos disponibles a partir de mensajes recibidos usando `data_mapper`. Devuelve 200.

GET `/api/v1.0/messages/topics.json`

No recibe parámetros. Devuelve un listado de *topics* disponibles a partir de mensajes recibidos usando `data_mapper`. Devuelve 200.

GET `/api/v1.0/messages/clients.json`

No recibe parámetros. Devuelve un listado de identificadores de cliente disponibles a partir de mensajes recibidos usando `data_mapper`. Devuelve 200.

GET `/api/v1.0/messages/types/:type.json`

Recibe el tipo como parámetro en la URL y los parámetros de paginación (página, elementos por página y offset). Luego, filtra los mensajes existentes en la base de datos en función del tipo, los pagina y los devuelve en el formato esperado por Dynatable. Devuelve 200 y un cuerpo con `records`, `queryRecords` y `totalRecords`.

GET `/api/v1.0/messages/topics/:topic.json`

Recibe el *topic* como parámetro en la URL y los parámetros de paginación (página, elementos por página y offset). Luego, filtra los mensajes existentes en la base de datos en función del *topic*, los pagina y los devuelve en el formato esperado por Dynatable. Devuelve 200 y un cuerpo con `records`, `queryRecords` y `totalRecords`.

GET `/api/v1.0/messages/clients/:client.json`

Recibe el cliente como parámetro en la URL y los parámetros de paginación (página, elementos por página y offset). Luego, filtra los mensajes existentes en la base de datos en función del cliente, los pagina y los devuelve en el formato esperado por Dynatable. Devuelve 200 y un cuerpo con `records`, `queryRecords` y `totalRecords`.

Dependencias

- `Grape`
- `SchemaStore`
- `SchemaBuilder`
- `DataMapper`
- `MqttPublisher`
- `JSON`

Detalles de interés

Los *endpoints* que listan y filtran entradas de la base de datos son idénticos para los tres tipos de filtros. Por lo tanto, están definidos usando metaprogramación. Esto permite cambiar el comportamiento de los tres filtros con un único cambio sin obligar a cambiar tres partes distintas del código para mantener consistencia.

3.2.10.2 ExceptionHandler

Responsabilidad

Es un Middleware de Rack, se encarga de evitar mostrar *stacktraces* al usuario, devolviendo un error 500 genérico en su lugar.

Atributos

- `app`: Aplicación de Rack a la que encapsula.

Métodos

Públicos

- `initialize(app)`: Instancia la clase y almacena la aplicación de Rack a la que encapsula.
- `internal_error_message`: Devuelve el mensaje genérico que mostrar cuando ha habido una excepción no controlada.
- `call(env)`: Llama a la aplicación `app` y rescata los `StandardErrorEs la clase raíz de las excepciones que deben ser controladas en Ruby.` no controlados.

Dependencias

- `Rack::Response`

3.2.10.3 HealthCheckFactory

Responsabilidad

Se encarga de construir la aplicación de Rack que mostrará el estado de los servicios de los que se depende.

Métodos

Públicos

- `self.build(app_tree)`: Crea una nueva instancia de `MiniCheck::RackApp` con dos pruebas, una para comprobar el estado de Redis y otra para MongoDB. Devuelve la instancia.

Dependencias

- `MiniCheck`

3.2.10.4 HttpServerFactory

Responsabilidad

Se encarga de construir la aplicación que se ejecutará al iniciar el servidor web.

Métodos

Públicos

- `self.build(app_tree)`: Crea una nueva instancia de `Rack::Builder` con el controlador de excepciones, la aplicación de *healthchecks*, *logger*, páginas estáticas y la API. Devuelve el `Rack::Builder`.

Dependencias

- `MiniCheck`

3.2.10.5 RackApps

Responsabilidad

Contenedor para las aplicaciones de Rack.

Atributos

- `app_tree`: Instancia del controlador raíz que se le pasa al inicializar.

Métodos

Públicos

- `initialize(app_tree)`: Instancia el contenedor y almacena el contenedor raíz en una variable de instancia.
- `server`: Almacena una instancia de `Rack::Builder` que se crea al llamar a `HttpServerFactory.build(app_tree)`. Es la aplicación que se ejecuta al arrancar el servidor.
- `health_check`: Usa `HealthCheckFactory.build(app_tree)` para instanciar un `MiniCheck::RackApp`.

- `api`: Configura la clase `Api` inyectando sus dependencias. Devuelve la clase.
- `logger`: Devuelve `Rack::CommonLogger`.
- `exception_handler`: Devuelve `ExceptionHandler`.

Dependencias

- `Rack::CommonLogger`
- `HttpServerFactory`
- `HealthCheckFactory`
- `Api`
- `ExceptionHandler`
- `AppTree`

3.2.11 Agentes MQTT

3.2.11.1 MqttAgents

Responsabilidad

Contenedor para los agentes MQTT.

Atributos

- `app_tree`: Instancia del controlador raíz que se le pasa al inicializar.

Métodos

Públicos

- `initialize(app_tree)`: Instancia el contenedor y almacena el contenedor raíz en una variable de instancia.
- `subscriber`: Almacena una instancia de `MqttSubscriber` a la que se le inyectan las dependencias apropiadas.
- `processor`: Almacena una instancia de `MqttMessageProcessor` a la que se le inyectan las dependencias apropiadas.
- `publisher`: Almacena una instancia de `MqttPublisher` a la que se le inyectan las dependencias apropiadas.
- `publisher_client`: Almacena una instancia de `MQTT::Client` a la que se configura correctamente y se conecta al *broker*.
- `subscriber_client_thread`: Almacena una instancia de `MqttSubscriberThread` a la que se le pasa un bloque con el código que se ejecuta en el suscriptor.

Dependencias

- `MqttSubscriber`
- `MqttMessageProcessor`
- `MqttPublisher`
- `MQTT::Client`

- `MqttSubscriberThread`
- `JSON::ParserError`
- `AppTree`

3.2.11.2 MqttMessageProcessor

Responsabilidad

Recibe un *topic* y un mensaje recibido con ese *topic* y guarda el mensaje en la base de datos y los pares clave-valor necesarios en Redis.

Módulos

- `Initializable`

Atributos

- `topic_store`: Instancia de `TopicStore` que se inyectará desde el contenedor.
- `schema_store`: Instancia de `SchemaStore` que se inyectará desde el contenedor.
- `data_mapper`: Instancia de `DataMapper` que se inyectará desde el contenedor.

Métodos

Públicos

- `process(topic, message)`: Método que analiza el mensaje, extrae tipo e identificador de cliente y guarda la información necesaria en base de datos y almacén clave-valor.

Privados

- `parse(message)`: Usa la librería JSON para deserializar el mensaje.
- `save_to_db(args)`: Llama a `data_mapper` para crear una nueva entrada en la base de datos.
- `build_schema(type, payload)`: Usa `JSON::SchemaGenerator` para generar el esquema JSON del mensaje.

Dependencias

- `JSON`
- `JSON::SchemaGenerator`
- `SchemaStore`
- `TopicStore`
- `DataMapper`

3.2.11.3 MqttPublisher

Responsabilidad

Publica un mensaje en un *topic*.

Módulos

- `Initializable`

Atributos

- `topic_store`: Instancia de `TopicStore` que se inyectará desde el contenedor.
- `client`: Instancia de `Mqtt::Client` que se inyectará desde el contenedor.

Métodos

Públicos

- `publish(args={})`: Método que construye el mensaje y lo publica. El mensaje tiene identificador de cliente RestClient.

Dependencias

- `JSON`
- `Mqtt::Client`
- `TopicStore`

3.2.11.4 MqttSubscriber

Responsabilidad

Ejecuta el hilo del subscriptor.

Módulos

- `Initializable`

Atributos

- `client_thread`: Instancia de `MqttSubscriberThread` que se inyectará desde el contenedor.

Métodos

Públicos

- `start!`: Ejecuta `run` sobre `client_thread`.

Dependencias

- `MqttSubscriberThread`

3.2.11.5 MqttSubscriberThread

Responsabilidad

Crea el hilo del subscriptor.

Atributos

- `thread`: Instancia de `Thread` que almacenará el hilo en ejecución.

Métodos

Públicos

- `initialize`: Crea un hilo que ejecuta el bloque que se le pasa como argumento de forma que si ocurre cualquier excepción, lo vuelve a intentar ejecutar 100 milisegundos después.

- `log_exception(e)`: Imprime la traza de la excepción que ocurra dentro del hilo.
- `run`: Ejecuta `run` sobre el hilo.

3.2.12 Persistencia

3.2.12.1 DataMapper

Responsabilidad

Se encarga de facilitar el acceso a la base de datos.

Módulos

- `Initializable`

Atributos

- `model`: Modelo que implementa `Mongoid::Document`.

Métodos

Públicos

- `create(args={}`: Instancia un documento con los argumentos apropiados y lo guarda.
- `all`: Delega sobre `model`.
- `delete_all`: Delega sobre `model`.
- `find_by_type`: Filtra los documentos por tipo.
- `find_by_topic`: Filtra los documentos por *topic*.
- `find_by_client`: Filtra los documentos por identificador de cliente.
- `list_types`: Lista los distintos tipos presentes en la base de datos.
- `list_topics`: Lista los distintos *topics* presentes en la base de datos.
- `list_clients`: Lista los distintos identificadores de cliente presentes en la base de datos.

Dependencias

- `Message`

Detalles de interés

Parte de los métodos están definidos usando metaprogramación, ya que el comportamiento es idéntico, cambiando el campo por el que se filtra o busca.

3.2.12.2 Message

Responsabilidad

Modelo de la base de datos para los mensajes.

Módulos

- `Mongoid::Document`

Campos

- `type`: De tipo `String`, almacena el tipo del mensaje.
- `topic`: De tipo `String`, almacena el *topic* del mensaje.
- `time`: De tipo `Time`, por defecto almacena el instante en que se recibe el mensaje.
- `client`: De tipo `String`, almacena el identificador de cliente del mensaje.
- `message`: De tipo `Hash`, almacena el contenido del mensaje.

Índices

- `type`
- `topic`
- `time`
- `client`

Métodos

Públicos

- `as_json`: Devuelve un *hash* con las claves `time`, `type`, `topic`, `client` y `message`. El tiempo está representado en formato `%Y-%m-%d %H:%M:%S` y el mensaje serializado en JSON.

Dependencias

- `Mongoid::Document`
- `JSON`

Detalles de interés

Representa entradas de la base de datos y Mongoid provee el DSL necesario para definir la estructura de dichas entradas. El valor por defecto de `time` es una lambda que llama a `Time.now`[15].

3.2.12.3 SchemaStore

Responsabilidad

Se encarga de tomar, listar o escribir esquemas en Redis.

Módulos

- `Initializable`

Atributos

- `redis`: Instancia de Redis.

Métodos

Públicos

- `list`: Llama a `hkeys` sobre Redis con la clave del *hash*.
- `get(name)`: Llama a `hget` sobre Redis con la clave del *hash* y el tipo.
- `set(name, value)`: Llama a `hset` sobre Redis con la clave del *hash*, el tipo y el esquema si no existe esquema para ese tipo.

Dependencias

- `Redis`

[15] Una función lambda es una forma de llamar a las funciones anónimas, es una forma de almacenar una función como un objeto que se puede pasar como argumento entre otros usos. En este caso, la ventaja sobre no usar una lambda es que se evalúa cada vez que se usa la clase, no solamente en tiempo de cargado, lo que devolvería siempre el mismo valor.

Detalles de interés

La abstracción elegida para este caso es un *hash* en Redis, ya que los tipos y esquemas forman una asociación uno a uno, siendo la clave el tipo y el valor el esquema. Redis provee una interfaz cómoda para interactuar con este tipo de relaciones[19].

3.2.12.4 TopicStore

Responsabilidad

Se encarga de listar o agregar *topics* en los que se envían mensajes de cierto tipo.

Módulos

- `Initializable`

Atributos

- `redis`: Instancia de Redis.

Métodos

Públicos

- `list`: Llama a `smembers` sobre Redis con la clave del *set*.
- `add`: Llama a `sadd` sobre Redis con la clave del *set* y el *topic*.

Privados

- `get_key(type)`: Devuelve una clave de la forma *topics_for_* seguido por el tipo, que será clave del *set* para ese tipo.

Dependencias

- `Redis`

Detalles de interés

La abstracción elegida para este caso es un *set* en Redis, ya que tienen la característica clave de no aceptar miembros repetidos, por lo que el procesador no tiene que preocuparse por si el *topic* ha sido visto anteriormente[19].

3.2.12.5 Stores

Responsabilidad

Contenedor para almacenamiento.

Atributos

- `app_tree`: Instancia del controlador raíz que se le pasa al inicializar.

Métodos

Públicos

- `initialize(app_tree)`: Instancia el contenedor y almacena el contenedor raíz en una variable de instancia.
- `redis`: Almacena una única instancia de `Redis` configurada con los parámetros adecuados.
- `message_mapper`: Se asegura de que la base de datos está configurada correctamente y luego almacena una instancia de `DataMapper` con el modelo `Message`.
- `schema_store`: Almacena una instancia de `SchemaStore` a la que se le inyecta la instancia de Redis.
- `topic_store`: Almacena una instancia de `TopicStore` a la que se le inyecta la instancia de Redis.

Dependencias

- `Redis`
- `DataMapper`
- `Message`
- `SchemaStore`

- `TopicStore`
- `AppTree`

3.2.13 *Tests*

Para probar el código de la aplicación Ruby se han creado varias baterías de pruebas usando RSpec[30].

Cada fichero crea un bloque `describe` en el que puede haber a su vez más bloques `describe` o `context` que se exponen los diferentes *tests* para una clase y cada *test* se encuentra dentro de un bloque *it*.

3.2.13.1 Api

Pruebas de la clase `Api`, prueban diferentes *endpoints*.

- **GET** `/api/v1.0/schemas.json`

  Pruebas realizadas

  RSpec Test 1 (Devuelve 200) *Comprueba que el endpoint devuelve el código de estado correcto.*
  RSpec Test 2 (Lista los esquemas) *Comprueba que* `schema_store` *recibe* `list`*.*

- **GET** `/api/v1.0/schemas/types/:type.json`

  Pruebas realizadas

  RSpec Test 3 (Devuelve 200 si el tipo existe) *Comprueba que el endpoint devuelve el código de estado correcto.*
  RSpec Test 4 (Devuelve el esquema) *Comprueba que se devuelve el esquema correcto retornado por* `schema_store`*.*
  RSpec Test 5 (Devuelve 404 si el tipo no existe) *Comprueba que el endpoint devuelve el código de estado correcto.*

- **POST** `/api/v1.0/messages.json`

  Pruebas realizadas

  RSpec Test 6 (Devuelve 201 si el tipo existe) *Comprueba que el endpoint devuelve el código de estado correcto.*
  RSpec Test 7 (Publica un mensaje) *Comprueba que se llama a* `publisher` *con los parámetros adecuados.*

RSpec Test 8 (Devuelve 404 si el tipo no existe) *Comprueba que el endpoint devuelve el código de estado correcto.*

- **GET** `/api/v1.0/messages/types.json`

Pruebas realizadas

RSpec Test 9 (Devuelve 200) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 10 (Lista los tipos) *Comprueba que `data_mapper` recibe `list_types`.*

- **GET** `/api/v1.0/messages/types/:type.json`

Pruebas realizadas

RSpec Test 11 (Devuelve 200 cuando no hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 12 (No devuelve mensajes) *Comprueba que la respuesta es la esperada en el estado vacío.*
RSpec Test 13 (Devuelve 200 cuando hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 14 (Devuelve los mensajes correctos filtrados por tipo y paginados) *Comprueba que la respuesta es la esperada cuando se devuelven entradas paginadas.*

- **GET** `/api/v1.0/messages/topics.json`

Pruebas realizadas

RSpec Test 15 (Devuelve 200) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 16 (Lista los *topics*) *Comprueba que `data_mapper` recibe `list_topics`.*

- **GET** `/api/v1.0/messages/topics/:topic.json`

Pruebas realizadas

RSpec Test 17 (Devuelve 200 cuando no hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 18 (No devuelve mensajes) *Comprueba que la respuesta es la esperada en el estado vacío.*
RSpec Test 19 (Devuelve 200 cuando hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*

RSpec Test 20 (Devuelve los mensajes correctos filtrados por *topic* y paginados) *Comprueba que la respuesta es la esperada cuando se devuelven entradas paginadas.*

- **GET** `/api/v1.0/messages/clients.json`

Pruebas realizadas

RSpec Test 21 (Devuelve 200) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 22 (Lista los clientes) *Comprueba que `data_mapper` recibe `list_clients`.*

- **GET** `/api/v1.0/messages/clients/:client.json`

Pruebas realizadas

RSpec Test 23 (Devuelve 200 cuando no hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 24 (No devuelve mensajes) *Comprueba que la respuesta es la esperada en el estado vacío.*
RSpec Test 25 (Devuelve 200 cuando hay mensajes) *Comprueba que el endpoint devuelve el código de estado correcto.*
RSpec Test 26 (Devuelve los mensajes correctos filtrados por cliente y paginados) *Comprueba que la respuesta es la esperada cuando se devuelven entradas paginadas.*

3.2.13.2 App

Pruebas de la clase `App`.

Pruebas realizadas

RSpec Test 27 (Ejecuta Puma) *Configurado con el puerto correcto.*
RSpec Test 28 (Llama `start!` sobre el subscriptor)

3.2.13.3 Cli

Pruebas de la clase `Cli`.

Pruebas realizadas

RSpec Test 29 (Ejecuta los tests) *Ejecuta la suite de RSpec al llamar a `cli test`.*
RSpec Test 30 (Llama `start!` sobre la aplicación) *Ejecuta `start!` al llamar a `cli start`.*

RSpec Test 31 (Abre una consola REPL) *Abre una consola REPL usando `pry` al llamar a `cli console`.*

3.2.13.4 Config

Pruebas de la clase `Config`.

Pruebas realizadas

RSpec Test 32 (Inicializa `app_tree`)
RSpec Test 33 (Encapsula la configuración)
RSpec Test 34 (Encapsula la configuración del servidor web)
RSpec Test 35 (Encapsula la configuración de Redis)
RSpec Test 36 (Encapsula la configuración de la base de datos)
RSpec Test 37 (Encapsula la configuración del cliente MQTT)
RSpec Test 38 (Carga la configuración de la base de datos)
RSpec Test 39 (Crea los índices de la base de datos)

3.2.13.5 DataMapper

Pruebas de la clase `DataMapper`. Se crearán *mocks* para el modelo y para algunos de los parámetros.

Pruebas realizadas

RSpec Test 40 (Inicializa `model`)
RSpec Test 41 (Permite persistir una entrada en la base de datos)
RSpec Test 42 (Devuelve la entrada al persistirla)
RSpec Test 43 (Delega `all` al modelo)
RSpec Test 44 (Delega `delete_all` al modelo)
RSpec Test 45 (Permite filtrar por tipo)
RSpec Test 46 (Permite filtrar por *topic*)
RSpec Test 47 (Permite filtrar por cliente)
RSpec Test 48 (Permite listar tipos)
RSpec Test 49 (Permite listar *topics*)
RSpec Test 50 (Permite listar clientes)

3.2.13.6 HttpServerFactory

Pruebas de la aplicación de Rack. En particular, de la aplicación de *health checks*.

Se falsifican las llamadas a MongoDB y Redis para estas pruebas.

Pruebas realizadas

RSpec Test 51 (Devuelve 200 si el estado es correcto)
RSpec Test 52 (Devuelve 500 si el estado es incorrecto)

3.2.13.7 MqttPublisher

Pruebas del publicador Mqtt. Se crean *mocks* para `topic_store` y `client`, y también para el mensaje.

Pruebas realizadas

RSpec Test 53 (Publica un mensaje usando el cliente) *Espera que el cliente reciba `publish` con el topic y el mensaje adecuado*

3.2.13.8 MqttSubscriber

Pruebas del subscriptor Mqtt. Se crea un `mock` del hilo.

Pruebas realizadas

RSpec Test 54 (Permite arrancar el hilo al llamar a `start!`)

3.2.13.9 SchemaBuilder

Pruebas de la clase `SchemaBuilder`. Se crea *mock* de `topic_store`.

Pruebas realizadas

RSpec Test 55 (Inicializa `topic_store`)
RSpec Test 56 (Construye el esquema esperado) *Construye el esquema completo dados el tipo y el esquema de mensaje.*

3.2.13.10 SchemaStore

Pruebas de la clase `SchemaStore`. Se crea *mock* de `redis`.

Pruebas realizadas

RSpec Test 57 (Inicializa `redis`)
RSpec Test 58 (Se llama a `hkeys` sobre el *hash* de Redis al listar)
RSpec Test 59 (Se llama a `hget` sobre el *hash* de Redis al tomar un esquema)
RSpec Test 60 (Se llama a `hset` si no hay esquema)
RSpec Test 61 (No se llama a `hset` si hay esquema)

3.2.13.11 TopicStore

Pruebas de la clase `TopicStore`. Se crea *mock* de `redis`.

Pruebas realizadas

RSpec Test 62 (Inicializa `redis`)
RSpec Test 63 (Se llama a `smembers` sobre el *set* de Redis al listar)
RSpec Test 64 (Se llama a `sadd` sobre el *set* de Redis al añadir)

3.2.14 Gemas

A continuación se describirá brevemente qué uso se ha hecho de cada gema[16] utilizada y de qué servicios dependen (si dependen de alguno).

Las dependencias se resuelven usando Bundler[29].

El fichero `Gemfile` describe dichas dependencias:

Listado 3.18 Contenido de `Gemfile`.

```
source 'https://rubygems.org'

gem 'grape', :git => 'https://github.com/intridea/grape.git', require: false
gem 'thor', require: false
gem 'puma', require: false
gem 'mini_check', git: 'https://github.com/workshare/mini-check.git', require: false
gem 'activesupport', require: false
gem 'rake', require: false
gem 'mqtt', require: false
gem 'json-schema-generator', require: false
gem 'json-schema', require: false
gem 'redis', require: false
gem 'mongoid', require: false
gem 'hashie', require: false


group :development, :test do
  gem 'pry', require: false
end

group :test do
  gem 'rspec'
  gem 'rack-test', require: false
  gem 'guard', require: false
  gem 'guard-rspec', require: false
end
```

3.2.14.1 Grape

Existen una gran cantidad de *frameworks* web para Ruby. Entre las más usadas están Rails, Sinatra, Lotus y Grape. Con diferencia, la más conocida es Rails, sin embargo es muy pesada y esconde gran parte de su complejidad. Esto permite que provea muchísimas funcionalidades útiles, pero

[16] A las librerías en Ruby se las suele llamar *gems*, gemas.

hace al desarrollador depender demasiado del *framework*, hasta el punto de condicionar el diseño de la aplicación para adaptarlo a unos raíles bien definidos.

Sinatra es un *micro-framework* web de propósito general. Lotus está centrado en ser modular y fácilmente testeable. Pero para el propósito de esta aplicación (crear APIs) se ha escogido Grape, ya que es idóneo para el diseño de APIs de forma sencilla.

De Grape se usa la clase `Grape::API`.

3.2.14.2 Thor

Se usa la clase `Thor` para definir una interfaz de línea de comandos que permite ejecutar los *tests*, arrancar la aplicación o una consola REPL.

3.2.14.3 Puma

Se usa la clase `Puma::CLI` para arrancar el servidor web con la aplicación de Rack correspondiente en el puerto adecuado.

3.2.14.4 MiniCheck

Se usa la clase `MiniCheck::RackApp` para definir una aplicación de Rack que permite probar que los servicios de los que depende la aplicación funcionan correctamente.

3.2.14.5 ActiveSupport

Es una colección de utilidades y extensiones a clases del sistema normalmente utilizada en Rails. Se usa la parte de inflexión, que provee métodos para convertir de CamelCase a underscore_case y viceversa, y nos permite cargar las clases dinámicamente.

3.2.14.6 Rake

Permite definir tareas que se pueden ejecutar llamando a `rake tarea`.

3.2.14.7 Mqtt

Provee la clase `Mqtt::Client` que permite publicar mensajes y suscribirse a diferentes *topics*. Provee los métodos `connect`, `publish`, `subscribe` y `get` entre otros.

3.2.14.8 JsonSchemaGenerator

Provee la clase `JSON::SchemaGenerator`, que permite generar esquemas JSON a partir de los mensajes recibidos.

3.2.14.9 Redis

Librería de Redis para Ruby. Se usa la clase Redis para ejecutar comandos contra Redis.

Obviamente, su funcionamiento depende de la conexión a una instancia de Redis.

3.2.14.10 Mongoid

Adaptador de MongoDB para Ruby. Permite hacer consultas a Mongo y obtener objetos que representan a cada una de las entradas del documento.

Depende del funcionamiento de MongoDB.

3.2.14.11 Hashie

Se usa la clase `Hashie::Mash` para encapsular la clase `Hash`. Permite extraer los valores de los elementos llamando a métodos en lugar de usando la notación habitual (`hash[:campo]`).

3.2.14.12 Pry

Pry es una *shell* alternativa a IRB para Ruby. Permite ejecutar código de forma cómoda, lo que ayuda a depurar el código.

3.2.14.13 RSpec

RSpec y MiniTest son las dos librerías predominantes para pruebas en Ruby. RSpec está más activo en los últimos años.

Provee un DSL que permite definir grupos de ejemplos y diferentes tests, usando los métodos `describe` e `it`.

3.2.14.14 Rack/Test

Provee métodos para probar aplicaciones de Rack en RSpec usando por ejemplo `get path` en un bloque `it`.

3.2.14.15 Guard

Es una utilidad que detecta cambios en el sistema de ficheros y lanza una tarea cuando dichos cambios ocurren. Se configura una tarea para lanzar los *tests* cuando se produce un cambio en el código o en los mismos *tests*. Esto hace cómodo programar escribiendo *tests* (especialmente TDD[17]).

[17] Test-Driven Development

3.2.15 *Front-end*

El código de *front-end* en Javascript es muy breve. Está compuesto de únicamente dos ficheros (más librerías externas, hojas de estilos y HTML).

Se usa JQuery para las peticiones a la API. En concreto las funciones `getJSON` y `ajax`[32].

Se usa JSON Editor para generar formularios en base a esquemas JSON generados a partir de mensajes recibidos[34].

Por último, se usa Dynatable para interactuar con la API y crear tablas con los mensajes recopilados, filtrados de diferentes maneras[33].

3.2.15.1 Página de publicación

En esta página se carga el fichero `kabuki.js`.

Figura 3.11 Selector de la página de publicación.

Este fichero llama a `/api/v1.0/schemas.json` para obtener un listado de todos los esquemas disponibles (ver figura 3.11) y reconstruye el editor cuando se selecciona uno u otro, llamando a `/api/v1.0/schemas/types/:type.json` para obtener el esquema a utilizar.

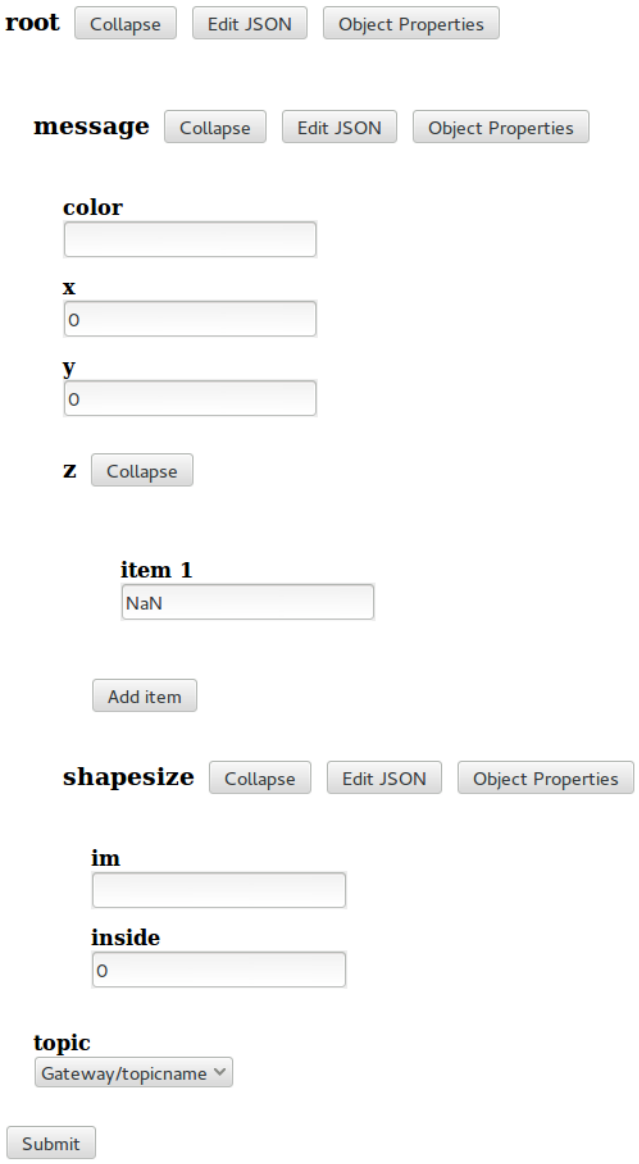

Figura 3.12 Editor de la página de publicación.

Luego, permite utilizar un botón para lanzar un mensaje con los valores introducidos en los campos del formulario, utilizando un POST a `/api/v1.0/messages.json` (ver figura 3.12).

3.2.15.2 Páginas de listado

En estas páginas se carga el fichero `kabuki_lists.json`.

Este fichero utiliza la URL de la ventana actual para saber a qué *endpoint* de la API llamar (filtro por tipo, cliente o *topic*). Las vistas están representadas en las figuras 3.13, 3.14 y 3.15 respectivamente. Muestra un listado de posibles valores del filtro obtenidos de la API, de los que se puede seleccionar uno.

Kabuki REST Client

something

Show: 10

Time	Type	Topic	Client	Message
2015-04-25 12:03:37	something	topic	cid	{"mensaje":1.01}
2015-04-25 12:03:47	something	topic	cid	{"mensaje":1.01}
2015-04-25 12:03:48	something	topic	cid	{"mensaje":1.01}
2015-04-25 12:03:49	something	topic	cid	{"mensaje":1.01}
2015-04-25 12:03:50	something	topic	cid	{"mensaje":1.01}
2015-04-25 12:06:14	something	topic	RestClient	{"mensaje":0}
2015-04-25 12:06:17	something	topic	RestClient	{"mensaje":12312312}

Pages: Previous 1 Next

Showing 7 of 7 records

Figura 3.13 Página de listado por tipo.

Kabuki REST Client

RestClient

Show: 10

Time	Type	Topic	Client	Message
2015-04-25 12:06:14	something	topic	RestClient	{"mensaje":0}
2015-04-25 12:06:17	something	topic	RestClient	{"mensaje":12312312}

Pages: Previous 1 Next

Showing 2 of 2 records (filtered from 7 total records)

Figura 3.14 Página de listado por cliente.

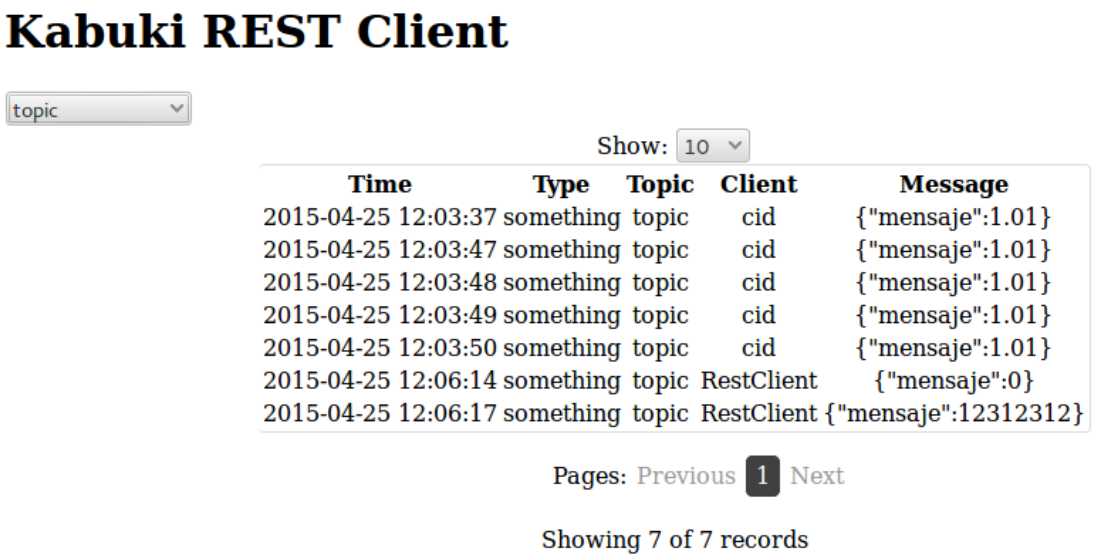

Figura 3.15 Página de listado por *topic*.

Hecho eso, es capaz de paginar las entradas que cumplen el filtro y mostrarlas en forma de tabla, navegando por las páginas y cambiando el número de entradas a mostrar. Esto es muy útil a la hora de depurar el código de la aplicación, ya que inspecciona un registro persistente de los mensajes que viajan por el sistema.

4 Ampliaciones

Este proyecto sienta las bases para una comunicación básica entre DDS y MQTT, pero hay multitud de aspectos abiertos a ampliación.

Algunas de estas ampliaciones son:

- **Análisis de robustez del código:** El código es funcional pero podría presentar problemas de rendimiento, concurrencia, excepciones sobre las que no se ha pensado, etc, ya que no ha sido probado fuera de entornos muy controlados.

- **Pruebas de carga:** Sería conveniente ejecutar pruebas de carga para describir el retraso que la pasarela crea en el sistema, la carga que permite transmitir (mensajes por segundo), etc.

- **Pruebas de interoperabilidad con otras implementaciones:** No se ha probado la interoperabilidad con clientes DDS basados en OpenDDS u OpenSplice.

- **Permitir definir dinámicamente las terminaciones de la pasarela:** Actualmente, la pasarela tiene una terminación en un dominio DDS y otra que se comunica con un *broker* MQTT. Terminaciones definidas dinámicamente en un fichero de configuración permitirían que una pasarela intercomunicase más de un dominio DDS con un *broker* o viceversa. Además, abriría la posibilidad de desarrollar otras terminaciones, por ejemplo AMQP.

- **Interoperabilidad con REST:** Definir APIs que permitan interoperar con DDS y MQTT, exponer esas APIs a través de un servidor web y desarrollar clientes que utilicen dichas APIs para enviar mensajes al sistema y recabar información. Posiblemente usando WebSockets o alguna tecnología similar para la recepción de dichos mensajes.

  Una opción interesante es desarrollar esta aplicación en un lenguaje altamente concurrente, por ejemplo en Elixir.

Elixir es un lenguaje funcional dinámico, diseñado para construir aplicaciones escalables y fáciles de mantener. Funciona sobre la máquina virtual de Erlang, permitiendo hacer uso de la plataforma OTP (*Open Telecom Platform*), que permite crear aplicaciones concurrentes sin tener que diseñar la aplicación con la concurrencia en mente[35]. Al ser un lenguaje relativamente nuevo funcionando sobre una base tan estable como la de Erlang, es un momento interesante para trabajar sobre él en pequeños proyectos como este. Un buen punto de partida sería el *framework* Phoenix[36].

- **Soporte para más formatos de serialización:** Sólo existe soporte para JSON actualmente. El código de ingreso y extracción de datos está acoplado al formato de serialización, sería interesante desacoplarlo todo lo posible previamente a trabajar en esta ampliación.

- **Filtro de *topics* que deben retransmitirse:** En su estado actual la pasarela retransmite todos los *topics*.

- **Permitir crear nuevos *topics* desde clientes MQTT:** Actualmente solamente los participantes DDS pueden crear nuevos *topics* en el sistema.

- **Añadir autenticación.**

- **Automatización de la configuración y despliegue:** Permitir desplegar la aplicación de forma automatizada y cómoda, permitiendo personalizar la configuración previamente.

- **Desarrollo de clientes MQTT:** Podrían crearse en diferentes lenguajes, permitiendo siempre la interoperabilidad con la pasarela. Originalmente, el cliente Kabuki era una utilidad de línea de comandos para enviar mensajes desde el lado MQTT similar a `mosquitto_pub`, pero que comprobaba el esquema de los mensajes contra los anteriormente recibidos. Debería ser trivial alterar esta aplicación para desarrollar un cliente MQTT eliminando la API REST.

5 Conclusiones

Este proyecto provee una base de interoperabilidad entre DDS y MQTT. Sin embargo, como se ha expuesto en el capítulo de ampliaciones, hay muchos frentes sobre los que habría que ampliar con el fin de disponer de una solución más completa.

Al ser sistemas tan dispares, esta pequeña base para la interoperabilidad presenta más complejidad de la esperada, siendo el reto principal el análisis de estructuras arbitrarias complejas de datos que a su vez pueden estar anidadas.

La solución se centra en poder extraer datos de los mensajes y transmitirlos por el otro extremo de la pasarela, sin tener en cuenta calidad de servicio, retrasos, sobrecarga introducida, etc.

Más allá de la solución dada, las aplicaciones han sido desarrolladas de manera que debería facilitar futuros cambios y ampliaciones, empleando diversas técnicas de ingeniería de *software* y programación orientada a objetos.

Durante la realización del proyecto se han empleado muchas tecnologías y herramientas, siempre intentando ir más allá de lo cubierto durante los años de carrera. En una nota subjetiva, esto contribuye al desarrollo del programador, en un panorama que cada vez tiende más al poliglotismo, tanto en lenguajes como en herramientas. Algunas de las ampliaciones ya se proponen acompañadas del uso de otros lenguajes y herramientas.

Por último, para un alumno que quiera introducirse a estas tecnologías, este proyecto es una base sobre la que se podrían construir numerosas ampliaciones. Sin embargo, no es una solución simple y la curva de aprendizaje puede ser empinada al principio, dado el uso de técnicas como la inyección de dependencias, a las que el alumno probablemente no haya estado expuesto aún.

Guía de instalación

Se presupone Linux como sistema operativo, ninguna de las aplicaciones ha sido probada en otros sistemas operativos. Únicamente se ha probado en CentOS, Ubuntu y Arch Linux.

En este apartado se detallan las dependencias de cada uno de los programas y se proporciona una fuente en la que informarse sobre su instalación:

1 Programa principal

- **JDK:** Instalar OpenJDK o alguna alternativa (el programa está probado en OpenJDK 7). Debe estar disponible en los repositorios de prácticamente todas las distribuciones Linux. Instalar vía `aptitude`, `yum`, `pacman`, etc. Esta dependencia permite ejecutar el código Java.

- **Maven:** Mismo caso que con OpenJDK, debe estar disponible en los diversos repositorios. Si se usa algún IDE (como Eclipse) es aconsejable instalar el *plugin* de integración adecuado. Esta dependencia permite instalar las demás dependencias de la aplicación Java.

- **Redis:** Suele estar disponible en los repositorios, si no, en *http://redis.io/* está disponible el código fuente para compilarlo. Más información sobre el proceso de compilación está disponible en el fichero README. Esta dependencia instala el almacén clave-valor usado por la aplicación. Esta aplicación no tiene porqué estar ejecutándose en el mismo equipo que la pasarela. Su dirección y puerto son configurables en las propiedades de la aplicación.

- ***Broker* MQTT:** El más comúnmente disponible en gestores de paquetes es `mosquitto`, aunque puede haber distintos paquetes para clientes y *broker*. Si dicho paquete no estuviese disponible, el código fuente está disponible en *http://mosquitto.org/*. Al igual que Redis, su dirección es configurable en las propiedades de la aplicación.

- **RTI DDS:** La guía de instalación está disponible en *http://www.rti.com/downloads/connext-files.html*. Es necesario establecer una serie de variables de entorno para el correcto funcionamiento del archivo de licencia.

2 Aplicación Ruby

Como ya se mencionó antes, ninguna de las aplicaciones se ha probado en sistemas operativos no Linux. Sin embargo, con Ruby son increíblemente frecuentes los problemas en entornos Windows, así que directamente se desaconseja su uso fuera de Linux, Mac OS, OpenBSD u otros similares a Unix.

Además de necesitar la aplicación principal en funcionamiento, la aplicación Ruby necesita de la instalación de las siguientes dependencias:

- **RVM:** Gestor de versiones de Ruby, es la mejor forma de instalar Ruby cómodamente. Toda la información está disponible en *https://rvm.io*.

- **MongoDB:** El servidor suele estar disponible en muchos repositorios como `mongod` o `mongo-server`. Es la base de datos donde se almacenan los datos con los que trabaja esta aplicación.

- **Bundler:** Una vez elegida la version de Ruby usando el comando `rvm use 2.2.0` o la versión en cuestión, se debe instalar `bundler` con el comando `gem install bundler`. Esta dependencia permite instalar todas aquellas necesarias para el funcionamiento de la aplicación Ruby.

Bibliografía

[1] MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. *http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html*

[2] DDS Version 1.2. OMG Standard. *http://www.omg.org/spec/DDS/1.2/PDF/*

[3] Extensible and Dynamic Topic Types for DDS. *http://www.omg.org/spec/DDS-XTypes/1.0/PDF/*

[4] Deploying DDS on a WAN and the GIG. F. Crespo, A. Campos, G. Pardo-Castellote. *http://www.omg.org/news/meetings/GOV-WS/pr/rte-pres/DDS_RoutingService_RTEW09.pdf*

[5] Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] Designing Reusable Classes. Ralph E. Johnson , Brian Foote. Journal of Object Oriented Programming. 1988.

[7] Agile Principles, Patterns, and Practices in C#. Robert C. Martin, Micah Martin. 2006. Prentice Hall.

[8] Inversion of Control. Martin Fowler. 2005. *http://martinfowler.com/bliki/InversionOfControl.html*

[9] DIP in the Wild. Brett L. Schuchert. 2013. *http://martinfowler.com/articles/dipInTheWild.html*

[10] Inversion of Control Containers and the Dependency Injection pattern. Martin Fowler. 2004. *http://martinfowler.com/articles/injection.html*

[11] Java Code to ByteCode. Part one. James D. Bloom. 2013. *http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html*

[12] Google Guice Github Wiki Pages. *https://github.com/google/guice/wiki*

[13] Google Inject TypeLiteral Wiki Page. *http://google.github.io/guice/api-docs/latest/javadoc/index.html?com/google/inject/TypeLiteral.html*

[14] Polyglot Persistence. Martin Fowler. 2011. *http://martinfowler.com/bliki/PolyglotPersistence.html*

[15] Common Object Request Broker Architecture (CORBA) Specification, Version 3.2. OMG Standard. Part 2: CORBA Interoperability. *http://www.omg.org/spec/CORBA/3.2/Interoperability/PDF*

[16] RTI Connext Java API Documentation. *http://community.rti.com/rti-doc/510/ndds/doc/html/api_java/*

[17] RTI DDS Best Practices: Use WaitSets, except when you need extreme performance. *https://community.rti.com/best-practices/use-waitsets-except-when-you-need-extreme-performance*

[18] RTI DDS Examples: DataSpy DDS. Gerardo Pardo-Castellote. *https://community.rti.com/filedepot_download/1655/21*

[19] Redis Documentation. *http://redis.io/documentation*

[20] The MongoDB 3.0 Manual. *https://docs.mongodb.org/manual/*

[21] Jedis Github Wiki Pages. *https://github.com/xetorthio/jedis/wiki*

[22] Gson User Guide. *https://sites.google.com/site/gson/gson-user-guide*

[23] Eclipse MQTT Paho Documentation. *http://www.eclipse.org/paho/files/javadoc/index.html*

[24] Apache Log4j Javadoc Documentation. *https://logging.apache.org/log4j/2.x/javadoc.html*

[25] Apache Maven User Guide *https://maven.apache.org/*

[26] Guava User Guide. *https://code.google.com/p/guava-libraries/wiki/GuavaExplained*

[27] JUnit Github Wiki. *https://github.com/junit-team/junit/wiki*

[28] Mockito Google Code Documentation. *http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html*

[29] Bundler Documentation. *http://bundler.io/*

[30] Relish App: RSpec Documentation. *https://relishapp.com/rspec*

[31] Rack Rubydoc. *http://www.rubydoc.info/github/rack/rack/master/file/SPEC*

[32] jQuery API Documentation. *http://api.jquery.com/*

[33] Dynatable Documentation. *http://www.dynatable.com/*

[34] JSON Editor Github Wiki. *https://github.com/jdorn/json-editor/wiki*

[35] Elixir Language Documentation. *http://elixir-lang.org/docs.html*

[36] Phoenix Framework Documentation. *http://www.phoenixframework.org/*