Asynchronous Communication in Distributed Environments!

OSGi Messaging Specification

Mark Hoffmann

Data In Motion Consulting GmbH

About Data In Motion

- Founded in 2010
- Located in Jena/Thuringia - Germany
- Consulting, Training, Independen RnD
  - Distributed environments
  - OSGi
  - Model-Driven Software Development
- Wide Range of Industries: Medical, Insurance, Transportation, Traffic, Public Sector

Content

- Introduction
- Publishing Messages
- Subscribing Messages
- MessageContext
- Reply-To Behavior
- Acknowledgment
- Features

Introduction

- Asynchronous messaging important in IoT area and distributed environments
- Wide variety of messaging protocols (MQTT, XMPP, Kafka, AMQP)
- Different implementation for same protocol
- Asynchronous event handling with reactive streams

History of Messaging in OSGi

- Event Admin Specification - In-VM eventing
- Distributed Eventing - RFC-214
- MQTT Adapter - RFC-229
- Reactive: Promises and PushStreams
- Talk at ECE 2017 about Messaging and PushStreams
- Custom Messaging implementation Gecko-Messaging
- OSGi Messaging - RFC 246

What is it about

- Easy-to-use API for messaging in OSGi
- Integration of 3rd-party messaging solutions into OSGi
- API for a common set of messaging use-cases
- Easy to use via Declarative Services
- Extensibility for vendor specific customizations
- Capabilities for mandatory and optional functionality
- Introspection - Message Runtime Service

Publishing “Hello World”


```java
/**
 * Smack XMPP
 */
XMPPTCPConnectionConfiguration.Builder configBuilder = XMPPTCPConnectionConfiguration.builder();
configBuilder.setUsernameAndPassword("username", "password");
configBuilder.setResource("SomeResource");
configBuilder.setXmppDomain("jabber.org");

AbstractXMPPConnection xmppConnection = new XMPPTCPConnection(configBuilder.build());
xmppConnection.connect();
xmppConnection.login();

ChatManager chatManager = ChatManager.getInstanceFor(xmppConnection);
EntityBareJid meAtYou = JidCreate.entityBareFrom("me@you.org");
Chat chat = chatManager.chatWith(meAtYou);
chat.send("Hello, world!");
// do something
xmppConnection.disconnect();
```

```java
/**
 * Eclipse Paho MQTT
 */
MqttAsyncClient mqttClient = new MqttAsyncClient("mqtt://iot.eclipse.org", "myclient");
MqttConnectOptions mqttOptions = new MqttConnectOptions();
mqttOptions.setAutomaticReconnect(true);
mqttOptions.setUserName("username");
mqttOptions.setPassword("password".toCharArray());
mqttClient.connect(mqttOptions);
mqttClient.publish("/demo", new MqttMessage("Hello, world!".getBytes()));
mqttClient.close();
```

```java
/**
 * AMQP / RabbitMQ
 */
ConnectionFactory factory = new ConnectionFactory();
factory.setUsername("username");
factory.setPassword("password");
factory.setVirtualHost("myHost");
factory.setHost("rabbitmqhost");
Connection amqpConnection = factory.newConnection();
Channel channel = amqpConnection.createChannel();
channel.basicPublish("myExchange", "myRoutingKey", null, "Hello, world!".getBytes());
channel.close();
amqpConnection.close();
```

Hmmm ....

Publisher Services

DATA IN MOTION

- Get Services using OSGi DS
- **`osgi.messaging.protocol`** target filter

```
@Reference(target="(osgi.messaging.protocol=mqtt)")
private MessagePublisher mqttPublisher;
@Reference(target="(osgi.messaging.protocol=amqp)")
private MessagePublisher amqpPublisher;
@Reference(target="(osgi.messaging.protocol=xmpp)")
private MessagePublisher xmppPublisher;
@Reference
MessageContextBuilderFactory builderFactory;
```

Publish “Hello World”


```
Message mqttMessage = builderFactory.createBuilder("(osgi.messaging.protocol=mqtt)")
        .channel("/demo")
        .content(ByteBuffer.wrap("Hello, world!".getBytes()))
        .buildMessage();
mqttPublisher.publish(mqttMessage);
```

```
Message amqpMessage = builderFactory.createBuilder("(osgi.messaging.protocol=amqp)")
        .channel("myExchange", "myRoutingKey")
        .content(ByteBuffer.wrap("Hello, world!".getBytes()))
        .buildMessage();
amqpPublisher.publish(amqpMessage);
```

```
Message xmppMessage = builderFactory.createBuilder("(osgi.messaging.protocol=xmpp)")
        .channel("me@you.org")
        .content(ByteBuffer.wrap("Hello, world!".getBytes()))
        .buildMessage();
xmppPublisher.publish(xmppMessage);
```

Subscribe

DATA IN MOTION

```
@Reference(target="(osgi.messaging.protocol=mqtt)")
private MessageSubscription mqttSubscription;
@Reference
MessageContextBuilderFactory builderFactory;

public void subscribeMessage() {

    MessageContext mqttContext = builderFactory.createBuilder("(osgi.messaging.protocol=mqtt)")
            .channel("/demo")
            .buildContext();
    PushStream<Message> messageStream = mqttSubscription.subscribe(mqttContext);

    messageStream.forEach((message)->{
        String content = new String(message.payload().array());
        String channel = message.getContext().getChannel();
        System.out.println("Received message over: " + channel + ", with content: " + content);
    });

}
```

Messages / MessageContext

- Message
  - ByteBuffer Content
  - MessageContext for Meta-Information
- Message Context
  - Channel Information
  - Content-Type / Encoding
  - Correlation-ID
  - Extension-Map
- Message Context Builder Factory

Reply-To-Behavior

- Optional in Messaging Specification
- RPC style behavior - Send request, receive an answer
- Not all protocols or implementations support that
- Availability is announced using features and capabilities
- Reply-To Publisher
- Reply-To Whiteboard provided from implementation
- User registers ReplyToSubscriptionHandler that bind to a whiteboard

Reply-To-Publishing


```
@Reference(target="(osgi.messaging.protocol=mqtt)")
private ReplyToPublisher mqttPublisher;
@Reference
MessageContextBuilderFactory builderFactory;

public void publishReplyToMessage() {

    Message request = builderFactory
            .createBuilder("(osgi.messaging.protocol=mqtt)")
            .channel("/demo")
            .correlationId("test123")
            .replyTo("demo_response")
            .content(ByteBuffer.wrap("Hello Word!".getBytes()))
            .buildMessage();

    Promise<Message> response = mqttPublisher.publishAndReply(request);

}
```

Reply-To-Subscription

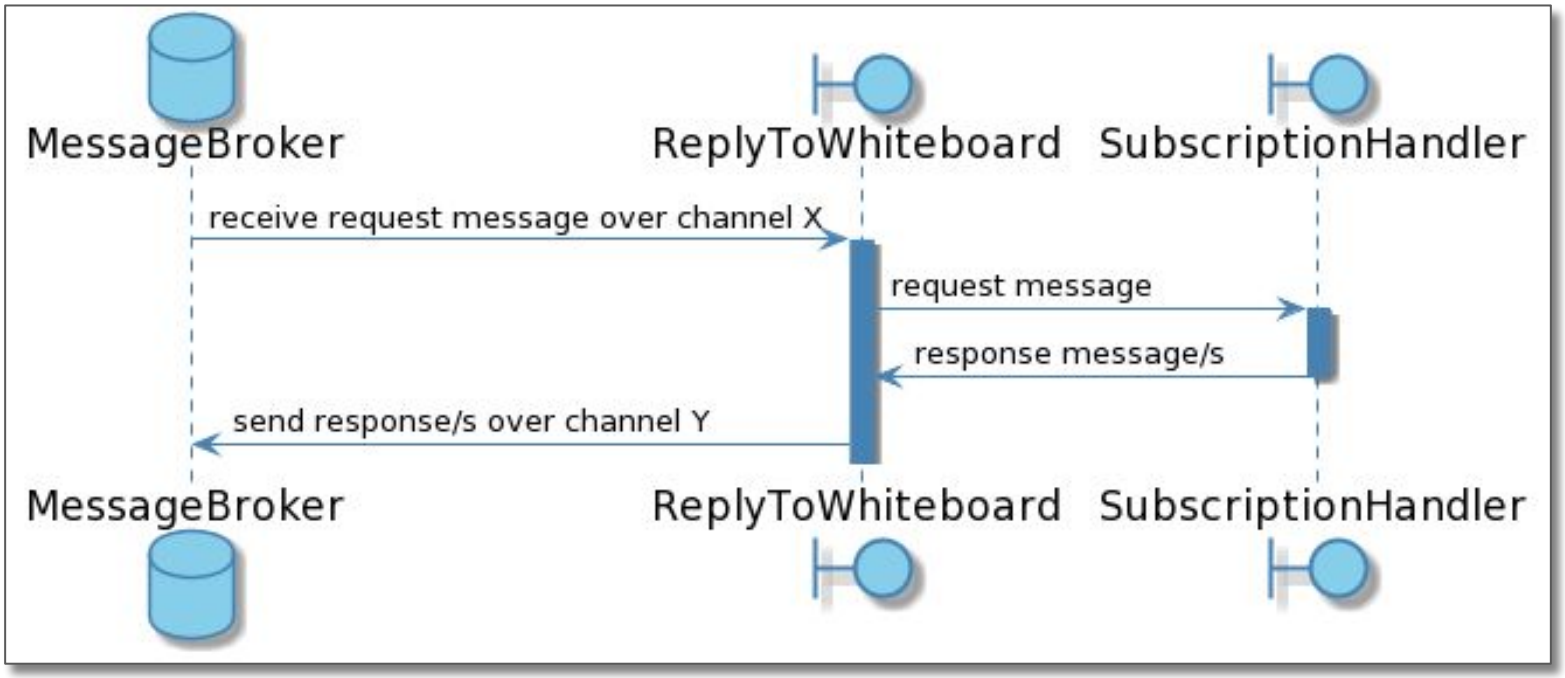
- ReplyToSubscriptionHandler
- ReplyToSingleSubscriptionHandler
- ReplyToManySubscriptionHandler

```
@Component(property = {
        "osgi.messaging.replyToSubscription.target=(replyWhiteboard=foo)",
        "osgi.messaging.replyToSubscription.channel=foo-request-topic"
})
public class FooBarSubscriptionHandler implements ReplyToSingleSubscriptionHandler {

    @Override
    public Message handleResponse(Message requestMessage, MessageContextBuilder responseBuilder) {
        String requestContent = new String(requestMessage.payload().array());
        requestContent += "Response";
        return responseBuilder.content(ByteBuffer.wrap(requestContent.getBytes())).buildMessage();
    }

}
```

Reply-To-Whiteboard



Acknowledgement

- Optional in the specification
- Announced via capabilities and features
- Support for acknowledge and rejection
- Lambda and Service based filtering, acknowledge handling
- Direct Acknowledging before message enters the stream
- Programmatic Acknowledging within a PushStream

Direct Acknowledgement


```
@Component(property = "foo=bar")
public class FooAckHandler implements Consumer<Message> {

    @Override
    public void accept(Message m) {
        AcknowledgeMessageContext ctx = (AcknowledgeMessageContext)m.getContext();
        AcknowledgeHandler handler = ctx.getAcknowledgeHandler();
        if (isGood(m)) {
            handler.acknowledge();
        } else {
            handler.reject();
        }
    }

    private boolean isGood(Message m) {
        return true;
    }

}
```

```
AcknowledgeMessageContextBuilder ackBuilder = (AcknowledgeMessageContextBuilder)
        builderFactory.createBuilder("(osgi.messaging.feature=acknowledge)");

MessageContext context = ackBuilder
        .handleAcknowledge("(foo=bar)")
        .postAcknowledge(m->{
            AcknowledgeMessageContext ctx = (AcknowledgeMessageContext)m.getContext();
            System.out.println("Acknowledge state is: " + ctx.getAcknowledgeState());
        })
        .messageContextBuilder()
        .channel("/demo")
        .buildContext();

PushStream<Message> messageStream = mqttSubscription.subscribe(context);
```

Programmatic Acknowledgement


```
MessageContext context = builderFactory
        .createBuilder("(osgi.messaging.feature=acknowledge)")
        .channel("/demo")
        .buildContext();

PushStream<Message> messageStream = mqttSubscription.subscribe(context);
messageStream.forEach(m->{
    AcknowledgeMessageContext ctx = (AcknowledgeMessageContext)m.getContext();
    AcknowledgeHandler handler = ctx.getAcknowledgeHandler();
    if (good) {
        handler.acknowledge();
    } else {
        handler.reject();
    }
});
```

Features

- Functional Features announce functionality
- Extension Features can also be used for configuration
- Set via extension map in context builder
- Examples:
  - auto-acknowledge
  - quality of service
  - last will message

Features Example


```
@Reference(target="(osgi.messaging.protocol=mqtt)")
private MessagePublisher mqttPublisher;
@Reference
MessageContextBuilderFactory builderFactory;

public void publishMessage() {

    Message lastWill = builderFactory.createBuilder("(osgi.messaging.protocol=mqtt)")
            .channel("last-will-channel")
            .content(ByteBuffer.wrap("EXIT".getBytes()))
            .extensionEntry(Features.EXTENSION_QOS, "2")
            .extensionEntry(Features.EXTENSION_LAST_WILL, Boolean.TRUE)
            .buildMessage();
    mqttPublisher.publish(lastWill);

}
```

What happens next?

- RFC 246 is finished
- Specification writing already in progress
- Reference implementation
- MQTT as protocol for the RI
- Compliance Tests
- Availability in next OSGi Enterprise release

Thanks for listening!

Resources:

Compendium:	https://docs.osgi.org/specification/osgi.cmpn/7.0.0/introduction.html
Github:	https://github.com/osgi/design
OSGi:	https://osgi.org
Twitter:	@motion_data
Web:	https://www.datainmotion.de