Messaging

Draft

14 Pages

Text in Red is here to help you. Delete it when you have followed the instructions.
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.

Abstract

Asynchronous communication is an important factor in today's business applications. Especially in the IoT domain but also for distributed infrastructures the communication over publish/subscribe protocols are common mechanisms. Whereas the existing OSGi Event Admin specification already describes an asynchronous event model within an OSGi framework, this RFP addresses the interaction of an OSGi environment with third-party communication protocols using a common interface.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

0.2 Trademarks

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in Fehler: Verweis nicht gefunden.

`Source code is shown in this typeface.`

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	26 Nov 2019	Mark Hoffmann, initial content copied from RFP
1.0	*January 2020*	*Mark Hoffmann, some initial API*

1 Introduction

In the past there have already been some efforts to bring asynchronous messaging into the OSGi framework. There was the Distributed Eventing RFC-214 and the MQTT Adapter RFC-229. In addition to that there are further available specification like OSGi RSA, Promises and PushStreams, that focus on remote events, asynchronous processing and reactive event handling. Promises and PushStreams are optimal partners to deal with the asynchronous programming model, that comes with the messaging pattern.

Because of the growing popularity of the IoT domain, it is important to enable OSGi to be connected with the common services of the IoT world. This RFP is meant to provide an easy to use solution in OSGi, to connect to and work with messaging systems. It is not meant to provide access to the full feature set and service guarantees of Enterprise class messaging solution like MQSeries or TIBCO EMS or massively scalable solutions like Kafka. Service guarantees and configuration details will be designed as configuration hints. The implementation will be optional and depend on the binding.

Protocols like AMQP, the Kafka Protocol or JMS are heavily used in back-end infrastructures. This RFP tries to address the use-case for connecting to those protocols with a subset of their functionality. For a seamless use of OSGi as well in IoT infrastructures and cloud-infrastructures, it is important to provide an easy to use and also seamless integration of different communication protocols in OSGi.

With the Event Admin specification, there is already an ease to use approach, for in-framework events. Distributed events often needs additional configuration parameters like quality of service, time-to-live or event strategies that needs to be configured at connection time or set at message publication time. This RFP is seen for standalone use but also as an extension to the Event Admin to provide the possibility for a Remote Event Admin.

2 Application Domain

Messaging is a pattern to reliably transport messages over an inherent unreliable network from a produce to one or more receivers. The process is to move messages from one system to another or many. The messaging system uses channels to do that. Because it is never clear, if the network or the receiver system is available, it is the task of the messaging system to handle that.

There are also different concepts in moving messages:

1. Send and Forget – Guarantees successful send
2. Send and Forward – Guarantees eventual successful receive

The following communication patterns exist:

1. Point-to-point
2. Publish-Subscribe
3. Guaranteed delivery
4. Temporary/transient channel
5. Dead-letter
6. Monitoring, error and administration channels

The use cases above cover Reply-To style communication and handling of common error conditions.

Another important fact is the structure of the messages. They usually consist of a header and body. The body contains the payload that is an array of bytes. The header provides additional properties for the message. There are some common properties that are used for handling Reply-To, sequencing/ordering of messages, time-synchronization, filtering and others. This is important because messaging decouples communication and has different demands regarding assumptions that are made to the process compared to a local call. Thus there is are additional semantics for e.g. time-outs, retry-counts, address-spaces.

Messaging in general induces an asynchronous programming model. Therefore Promises and PushStreams are already existing specifications that are an optimal solution for data-handling as well as scaling of actors over more than one thread. These specifications allowing flexible message transformation into internal data formats. Further Promises provide the possibility to realize patterns like message construction or aggregation. [1]

2.1 Terminology + Abbreviations

2.1.1 Message

A message is a data structure that holds the payload and additional meta-information about the content.

2.1.2 Channel

Channels are named resources to connect programs and interchange the messages. .

2.1.3 Sender / Publisher

A sender writes a message into a channel.

2.1.4 Receiver / Subscriber

One or more receivers read messages from a channel.

2.1.5 Reply-To Messaging

Sending a request and receiving a response are two separate atomic operations. Thus waiting for a response is not a blocking operation in the underlying implementation., A special message information, the correlation identifier, is used to assign a request to a response. Sometimes the reply-to address can be generated from the messaging system and is also submitted as property with the request message.

3 Problem Description

The OSGi Alliance already has a successful specification for messaging within an OSGi framework. The EventAdmin specification is well defined and widely used. The same is for the RSA specification that provides a good ground for synchronous calls. Also asynchronous remote services are supported in the RSA.

In the domains of IoT there are standardized protocols to connect remote devices and submit data over a broker based messaging system from remote clients. But also in cloud-based infrastructures, messaging systems are often used for de-coupling of services or functions.

Today, to interact with such systems the implementer has to deal with messaging protocol specifics and operational conditions, that are not covered, by existing specifications. With OSGi Promises and the PushStream specification there are already major parts available to deal with an asynchronous programming model. This is a requirement when using messaging.

The missing piece is a standardized way to send and receive data that supports the messaging patterns. Consuming and producing data using common protocols like AMQP, MQTT or JMS using OSGi services, would integrate an OSGi application into more systems.

Also other specifications could benefit from this RFP. It should be possible to layer RSA remote calls over messaging. It should also be possible to provide a remote Event Admin service.

3.1 Intents

Messaging systems vary widely in their capabilities and are configurable with regard to guarantees of delivery. We do not want to expose this complexity the user of this solution. The RSA specification uses intent for that purpose.

4 Requirements

4.1.1 General

- MSG010 – The solution MUST be technology, vendor and messaging protocol independent. MSG030 – The solution MUST be configurable (address-space, timeouts, quality of service guarantees)
- MSG050 – The solution MUST announce their capabilities/intents to service consumers
- MSG060 – The solution MUST provide information about registered channels, client connection states, if available
- MSG070 – The solution MUST support the asynchronous programming model
- MSG080 – The solution MUST support a client API
- MSG090 – The solution MUST respect requested intents
- MSG095 – The solution MUST announce its supported intents
- MSG100 – The solution MUST fail when encountering unknown or unsupported intents.

4.1.2 Channels

- MSG100 – It MUST be possible to asynchronously send messages to a channel.
- MSG120 – The solution MUST support systems that support point-to-point channel type
- MSG130 – The solution MUST support systems that support the publish-subscribe channel type
- MSG140 – The solution MUST support quality of service
- MSG150 – The solution MUST support send-and-forget and send-and-forward semantics

- MSG160 – The solution SHOULD support Reply-To calls, if possible. For that the solution MUST act as caller (publish and subscribe) as well as Reply-To receiver (subscribe on publish)

- MSG170 – The solution SHOULD support filter semantics like exchange / routing-key and wildcards for channels

- MSG180 – The solution MAY support a do-autocreate as well as do-not-autocreate

Messages

- MSG200 – Messages bodies MUST support sending of byte-data

- MSG205 – The implementation MAY place limits on the size of the messages that can be send. The existence of a message size limitation for an implementation SHOULD be signaled.

- MSG210 – It SHOULD be possible to support additional message properties like sequencing and correlation. The implementation SHOULD provide access to properties when available.

- MSG220 – The solution MAY define a content encoding

- MSG230 – The solution MAY support message time-to-live information

- MSG240 – The solution MAY support manual acknowledge/reject support for messages

- MSG250 – The solution MAY have a journalling support

- *MSG260 – It MUST be possible to identify the channel the message was received on*

5 Technical Solution

The concept of the messaging specification relies on a message object and a corresponding message context. A message is bound to a message context. This pattern is similar to the EventAdmin specification, where you get the topic and properties from the Event, you can get the context from a message. The difference is that the MessageContext contains more information and typed information, than just a topic and a properties map.

Service interfaces for subscribing on channels and publishing of messages are separated into own interfaces. That makes it possible just to implement either subscription or publishing. Where the implementations of the service interfaces can be specific to a certain protocol, the message context implementation can be as well protocol specific.

To get runtime information about a certain connection of an implementation, a MessageRuntimeService exists that exposes connection information.

Package: org.osgi.service.messaging

- Adding some sample code

- describing context attributes more in detail Mime-type for content-type , ...

5.1 Message Object

The message object is a wrapper for the plain content as well as connection information of a message.

```
/**
 * An message object
 */
public interface Message {

	/**
	 * Returns the playload of the message as {@link ByteBuffer}
	 * @return the playload of the message as {@link ByteBuffer}
	 */
	public ByteBuffer payload();

	/**
	 * Returns the message context. This must not be <code>null</code>.
	 * @return the {@link MessageContext} instance of this message
	 */
	public MessageContext getContext();

}
```

5.2 MessageContext

Messaging is not only about just sending data over a distributed network. It may happen, that you need certain connection information when the message has arrived. On the other hand, you just want to send/publish one special message, with additional connection information. The message context object is meant for that.

```
/**
 * Context object that can be used to provide additional properties that
 * can be put to the underlying driver / connection.
 * The context holds meta-information for a message to be send or received
 */
public interface MessageContext {

	ChannelDTO getChannel();

	/**
	 * Returns the content type
	 * @return the content type
	 */
	public String getContentType();

	/**
	 * Returns the content encoding
	 * @return the content encoding
	 */
	public String getContentEncoding();

	/**
```

```
         * Returns the correlation id
         * @return the correlation id
         */
        public String getCorrelationId();

        /**
         * Returns the reply to channel
         * @return the reply to channel
         */
        public ChannelDTO getReplyToChannel();

}
```

5.3 Message Service Runtime

```
/**
 * The MessageServiceRuntime service represents the runtime information of a
 * Message Service implementation.
 * <p>
 * It provides access to DTOs representing the current state of the connection.
 * <p>
 * The MessageServiceRuntime service must be registered with the
 * {@link MessageConstants#MESSAGE_CONNECTION_URI} service
 * property.
 *
 */
@ProviderType
public interface MessageServiceRuntime {

        /**
         * Return the runtime DTO containing the connection state
         *
         * @return the runtime DTO
         */
        RuntimeDTO getRuntimeDTO();

}
```

5.4 Subscription

The proposed interface defines a message subscription. I covers two cases:

1. expecting a stream of data for a channel
2. expecting just one answer, for a reply-to-request

```
public interface MessageSubscription {

        /**
```

```
         * Subscribe the {@link PushStream} to the given topic
         * @param topic the topic string to subscribe to
         * @return a {@link PushStream} instance for the subscription
         */
        public PushStream<Message> subscribe(String topic);

        /**
         * Subscribe the {@link PushStream} to the given topic with a certain quality
of service
         * @param topic the message topic to subscribe
         * @param context the optional properties in the context
         * @return a {@link PushStream} instance for the given topic
         */
        public PushStream<Message> subscribe(MessageContext context);

        /**
         * Subscribe for single answer on a request
         * @param message the request message
         * @return the {@link Promise} that is resolved, when the answer arrived
         */
        public Promise<Message> subscribeToRequest(Message message);

        /**
         * Subscribe for single answer on a request
         * @param message the request message
         * @param context the optional properties in the context
         * @return the {@link Promise} that is resolved, when the answer message has
arrived
         */
        public Promise<Message> subscribeToRequest(Message message, MessageContext
context);

}
```

6 Data Transfer Objects

6.1 RuntimeDTO

```
/**
 * Represents the message runtime DTO
 *
 */
public class RuntimeDTO extends DTO {
```

```
    /**
     * The DTO for the corresponding {@code MessageServiceRuntime}. This value is
     * never {@code null}.
     */
    public ServiceReferenceDTO          serviceDTO;

}
```

6.2 ChannelDTO

The ChannelDTO describes a channel, which can be a topic or queue. It additionally contains the possibility to define extensions, beside the channel name. This can be information like a routing-key.

```
/**
 * A {@link DTO} that defines a channel with the possibility to provide additional
 * channel information like routing keys.
 */
public class ChannelDTO extends DTO {

    /**
     * The name of the channel
     */
    public String name;

    /**
     * A possible extension to a channel like a routing key
     */
    public String extension;

}
```

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Enterprise Integration Pattern: Designing, Building, and Deploying Messaging Solutions. Gregor Hohpe, Bobby Woolf. ISBN 0-133-06510-7.

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Mark Hoffmann
Company	Data In Motion Consulting GmbH
Address	Kahlaische Strasse 4, 07745 Jena
Voice	
e-mail	m.hoffmann@data-in-motion.biz

10.3 Acronyms and Abbreviations

10.4 End of Document