RFP-192-Messaging

Final

10 Pages

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 Table of Contents

0.2 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 6.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

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

Revision	Date	Comments
Initial	*28.12.2018*	*Initial - Mark Hoffmann*
0.1	*31.01.2019*	*David Bosschaert – minor editorial changes and some comments*
0.2	*13.02.2019*	*Mark Hoffmann – changed wording for RPC and added some proposed use-cases*
0.3	*19.02.2019*	*Mark Hoffmann – comments from the F2F Berlin discussion*
0.4	*13.03.2019*	*Michael Roeschter – reviewed, edited for clarity with regard to design goals. The F2F in Berlin clarified the design goal to provide a simple interface to messaging systems, but not providing access to the full feature set of Enterprise class messaging solution (like MQ Series or TIBCO EMS) or massively scalable solutions (like Kafka).*
0.5	*21.05.2019*	*Mark Hoffmann – review and fixed some typos*
0.6	*02.06.2019*	*Mark Hoffmann – Accepted changes from F2F discussion, rewrote Use-case section*

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 Use Cases

4.1 Message sending

A publisher wants to be able to send single messages as well as provide messages through a PushStream. It must be possible to provide additional message properties for either case.

4.2 Receiving messages

As a recipient I want to be able to poll the next message in a channel. I want to do that in a blocking or non-blocking way.

It is also expected to support filter criteria for receiving messages, if supported by the underlying messaging system. So it could be possible to obtain a message with a specific identifier or correlation-ID.

~~The receiver must be able to poll for the next message in a channel. Both a blocking and non-blocking APIs must be available. Optionally, polling for a message may support filter criteria if supported by the underlying messaging system. E.g. a filter may used to obtain a message with a specific identifier or correlations ID.~~

If the messaging system supports journalling (like Kafka) then it should be possible to start consuming from any point in a retained history. Received messages should provide information about their position (offset) in the

journal. Alternatively to providing a absolute position to start from the system should also allow to start from earliest or latest.

4.3 Reply-To messaging

I want to use efficient Reply-to communication, as long as the implementation supports it. In that case I don't want to take care about setting or/and accessing Reply-To headers, correlation-ID or generating temporary channels. This should be handled by the implementation.

~~The implementation must allow access to mechanisms used for efficient Reply-to communication. E.g. setting/accessing Reply-To headers fields, setting/accessing correlation IDs, creating temporary channels.~~

The implementation should respect the use cases to be a caller or a call receiver.

~~The receiver/replier implementation must provide a simple means to reply-to a message and implement sensible default behavior. Such as, replying to the "Reply-to" channel when available, copying the correlation ID when available.~~

To provide synchronous and asynchonous behavior Promises should be used.

~~The sender/requester implementation must provide a simple synchronous means of performing an Reply-To using sensible default behavior. Such as, temporary channels, reply-to fields, correlations IDs.~~

4.4 Guarantee of delivery

If an implementation supports guarantees of delivery, it should be possible to take profit of this feature. So e.g. many implementation support a auto-acknowledgment for the send-and-forget principle. Otherwise it maybe necessary to acknowledge or reject messages explicitly in certain situations.

There are also other possible implementation specific aspects like: Storage guarantees for messages, journalling, persistence, exactly once delivery, transactions, out of sequence acknowledgment, at most once delivery, duplicate delivery. It is expected that various implementations handle guarantee of delivery in a different way.

~~To simplify the usage of the messaging service only basic guarantees of delivery should be exposed in the API. Advanced guarantees of delivery MAY be provided in as an intent and MAY be implementation specific.~~

~~The following aspects are implementation specific: Storage guarantees for messages, journalling, persistence, exactly once delivery, transactions, out of sequence acknowledgment, at most once delivery, duplicate delivery.~~

~~The sender has only "send" mode. The delivery status of the messages is after the "send" method returns is intent and implementation specific.~~

~~The two receiver modes:~~

1. ~~"Auto acknowledge": The implementation attempts to acknowledge the messages as soon as possible. The details are intent and implementation specific. The message MAY be acknowledge when it enters the receiver buffer. The message MUST be acknowledged before is delivered to the receiving code.~~

~~Explicit acknowledge. The implementation attempts to defer the acknowledgment of the messages till an explicit signal is given. The details are intent and implementation specific. The message MUST NOT be acknowledged before it enters the receiving code. The message MUST be acknowledged after a callback from Pushstream delivery returns. The message MUST be acknowledged the receiving code signals acknowledgment.~~

4.5 Event Admin

It should be possible to connect an OSGi Event Admin with a channel, to send or receive Event Admin Events. For that a topic-to-channel mapping could be defined

4.6 Intents

Because there are different features in different messaging system implementations that are either supported or not, these features should be signaled like intents in the RSA specification.

So it should be possible, to filter implementation against these intents. Implementation specific behavior, like the before mentioned “guarantee of delivery” can also signaled.

~~The solution will allow to signal intents, to the following situations:~~

1. ~~Connection time: Defining service defaults and connection parameters~~
2. ~~Connecting/Creating a channel~~
3. ~~Sending a messages~~

~~Binding a Pushstream to a service/channel~~

~~Well known intents will be defined as constants in a reserved namespace. The solution will allow the passing of arbitrary intents.~~

1. ~~The implementation MAY respect intents.~~
2. ~~The implementation MUST ignore intents it cannot service silently.~~

~~If the service consumer wants a guarantee of capability, it is advised to query the capabilities of the messaging services implementation and react accordingly.~~

5 Requirements

5.1.1 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
- MSG100 – The solution MUST fail when encountering unknown or unsupported intents.

5.1.2 Channels

- MSG100 – It MUST be possible to send asynchronous 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 MUST 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

6 Document Support

6.1 References

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

6.2 Author's Address

Name	Mark Hoffmann
Company	Data In Motion Consulting GmbH
Address	Kahlaische Str. 4, 07743 Jena, Germany
Voice	+49 175 7012201
e-mail	m.hoffmann@data-in-motion.biz

6.3 End of Document