RFC 214: Distributed Eventing

Draft

31 Pages

Abstract

10 point Arial Centered.

The OSGi specifications have described how to distribute OSGi services across VM boundaries now for a number of years. However many OSGi services are synchronous in their nature. Many of today's business applications require asynchronous distributed communication mechanisms. While the OSGi Event Admin specification describes an asynchronous eventing model inside the Java VM this does not address event distribution to other Vms. In addition, while the OSGi Asynchronous Services specification defines mechanisms for asynchronously invoking services, it does not address some concerns specific to eventing. This RFP aims to address the issue of Distributed Events in an OSGi context.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

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 1.

```
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	Aug 27 2014	Initial version of the document ahead of the Madrid F2F. Copied from the RFP. Tim Ward (Paremus)
0.1	Sep 10 2015	Add a proposal for asynchronous event streams and an updated Event Admin
0.2	Nov 12 2015	Separate the Push Stream into RFC 216

1 Introduction

This RFC began as an RFP nearly two years ago, in an effort to provide a better asynchronous messaging and eventing solution between OSGi framework. The RFP experienced some delays because parts of the problem space related to other OSGi RFCs. The primary blocks were the lack of an “updatatable” remote service, and the lack of native support for asynchronous primitives. The Enterprise R6 release will include both RSA 1.1, the Async Service, and OSGi Promises, meaning that further progress is now possible for Distributed Eventing.

The RFC aims to overcome some of the limitations of the existing EventAdmin, particularly when applied to remote systems, and to make use of the advanced features now available within the framework.

2 Application Domain

Distributed systems may be built using a number of different *interaction patterns.* Despite vocal proponents for each approach - it is increasingly clear that no one architectural solution is optimal in every context. Rather there is a continuous spectrum of interaction behaviors. If at all possible – these should ideally be supported in a consistent / coherent manner.

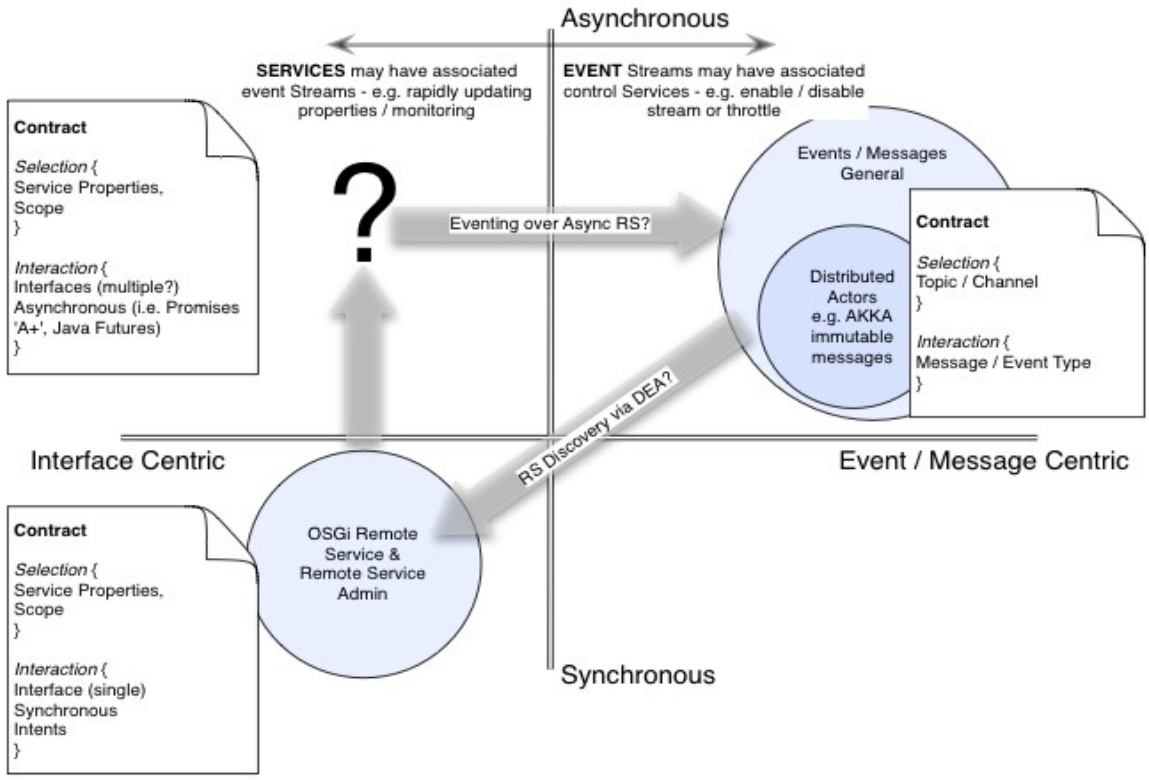

Figure 1: Types of distributed interaction

Synchronous RPC Services: The OSGi Alliance addressed the requirement for Remote Services via the *remote service* and *remote service admin specifications:* these specifications for synchronous RPC Services. In a dynamic environment (1..n) Services may be dynamically discovered, a sub-set of which (1..m where m<n) may be selected for use based on advertised Service properties. Service properties might be *Immutable* e.g. *Version, Location* or *Ownership* information – or *Mutable*: *reliability metrics*, *rating* or *cost metrics* – which may changing frequently in time.

It should be noted that:

- The RSA architecture is modular – allowing different Data Distribution and Discovery Providers to be used. This approach is extremely flexible. Some RSA implementations may choose to use a distributed P2P event substrate to provide Service discovery while other implementations use some form of static look-up service. Which ever is used - a coherent OSGi architecture is maintained. The use of an distributed Event Substrate for Service Discovery is one example of how RS/RSA and Distributed Eventing might interact.

- The current RSA specification does not address Service property updates: properties may change – and one does not necessarily wish to remove and re-add a Service because of this change. In more extreme cases, for volatile Service properties, one may wish to monitor these. Here a reference to the appropriate event streams might be advertised as Service properties. This scenario highlights a second potential relationship between RS/RSA and Distributed Eventing. (Note that it is planned to update the Remote Service Admin specification for Enterprise R6 to support Service property updates, see RFC 203.)

Note the suggested RS/RSA enhancements are out of scope of this RFP – they are mentioned to illustrate the potential relationships between RS/RSA and a Distributed Eventing specification.

Asynchronous Services: Synchronous Services will block until a response is received from the local or remote service. However, on occasions it is preferable for the client to return from the call immediately and continue – some form of notification received at a future point in time – i.e. a 'promise' or a 'future'.

While a number of remote service data distribution providers (RSA) can – in principle support - asynchronous operation, there are currently no OSGi specifications to address support for asynchronous Services (local or remote). Such a specification is desirable as asynchronous Services are increasing popular in Cloud and Distributed environments – and increasingly the JavaScript development community (e.g. node.js). Work in the planned JavaScript RFC will look at implementations of asynchronous Service Registries.

As indicated in Figure 1 – in static environments - Asynchronous Services might be used as a mechanism to implement distributed events. This makes less sense in dynamic environments as some form of discovery mechanism is required – which is usually event based for scaling. So Distributed Eventing would more likely underpin RS/RSA.

This is an important area that requires OSGi Alliance specification work, work that clear relates to both Distributed Eventing & RS / RSA, but Asynchronous Services are out of scope of this current RFP; they are the topic of RFP 132.

Distributed Events / Messages:

Asynchronous Message / Event based approaches are increasingly the underpinnings of large scale distributed environments including 'Cloud'. In these distributed systems *Publishers* endpoints are decoupled from *Subscribers* endpoints; association is achieved via the *Topic* the *Publishers* and *Subscribers* choose to subscribe to a named Topic - and /or – a specific Message type.

Implementations vary considerably – and range from 'classic' enterprise message solutions - (e.g. JMS/AMQP) with centralized message brokers – to peer-to-peer de-centralized P2P solutions – e.g. 0MQ and the Object Management Group's (OMG) Data Distribution Service -
see .http://www.omg.org/technology/documents/dds_spec_catalog.html

In principle asynchronous message based system provide the potential for greater scalability. However one cannot naively claim that asynchronous messaging will always scale more effectively than synchronous services: the performance characteristics are implementation dependent. An asynchronous messaging Service implemented via a central broker may introduce significant latency, throughput bottlenecks and points of architectural fragility. Whereas a dynamic Services approach with effective Service endpoint load balancing capabilities – would avoid these issues. However, correctly implemented P2P asynchronous message based systems will out perform both - with lower latency, higher throughput and increased resilience.

Due to the increased level of end-point de-coupling and potentially the use of parallel asynchronous pipelines, interaction contracts within message / event based systems are more challenging. Unlike a Service centric approach - failure of Subscribers is not obvious to Publishers (or visa-versa).

Dependent upon the *Capabilities* of the Distributed Eventing provider – events may / may not be durable – and in-order delivery of message / events may / may not be possible.

- Broker based messaging solutions (i.e. JMS Brokers) typically rely on ACID transactions between publishers and the message broker, then the message broker and subscribers. Such solutions are typically used for chaining interactions between coarse grained services running on large compute servers – i.e. Mainframes / large Unix Systems in traditional Enterprise environments. However centralized brokers / ACID transactions represent bottlenecks and points of fragility: failing to efficiently scale in large distributed highly parallel asynchronous environments.

- Increasingly highly distributed / parallelized environments typical of 'Cloud' are using P2P messaging solutions with compensational / recovery / eventual consistency / based approaches to recovery. In such environments the components with a distributed system need to be idempotent as messages / events / may be re-injected in response to some form of timeout or failure notifications. In such environments aggregation points are still required to coordination at input (fan-out) and output (fan-in) boundaries of the parallel flows.

From a developer perspective '*Actors*' are an increasingly popular asynchronous programming style. While popularised by the the Scala / Akka community – Java Actor frameworks also exist – i.e. the kilim actor framework (http://www.malhar.net/sriram/kilim/) and the NetFlix RXJava project https://github.com/Netflix/RxJava/wiki. In these environments local asynchronous events (locally using a message / mailbox pattern) may be distributed between remote 'Actors' via a plug-able messaging layer; e.g. for Akka 0MQ or via Camel / Camel plug-ins. An OSGi Distributed Eventing specification would provide a natural remoting substrate for '*Actor*' / OSGi based applications.

2.1 Point-to-point/Queue semantics with current Event Admin Service

Some projects use the OSGi Service Registry Hooks to add point-to-point and/or queue messaging semantics to existing Event Admin Service implementations. This approach is working well for these projects and does not actually require a change to the Event Admin Service specification as it uses the hooks to only show the listeners that should receive the message to the Event Admin Service. While not distributed across remote frameworks such a design could also be relevant in a distributed context.

2.2 Existing approaches to distribute the Event Admin Service

A number of projects have successfully implemented a distribution-enabled Event Admin Service employing the existing OSGi API of the Event Admin Service to send events to remote clients. A master thesis was also written on the topic in 2009 by Marc Schaaf [4].

While this approach is very useful in certain situations, it has limitations which make the current Event Admin Service not generally applicable as a service for distributing events.

2.3 Terminology + Abbreviations

Event: a notification that a circumstance or situation has occurred or an incident happened. Events are represented as data that can be stored and forwarded using any mechanism and/or technology and often include information about the time of occurrence, what caused the event and what entity created the event.

Message: a piece of data conveyed via an asynchronous remote communication mechanism such as a message queue product or other middleware. A message can contain an event, but can also have other information or instructions as its payload.

Common definitions for messaging systems include:

Queue: A messaging channel which delivers each message only to one receiver even if multiple receivers are registered, the message will only be delivered to one of them. If a receiver disconnects than all following messages are distributed between the remaining recipients. (It should be configurable that if no recipient is registered when a message is about to be delivered if the message is kept until a receiver is registered or if the message will be lost)

Topic: A publish and subscribe channel that delivers a message to all currently subscribed receivers. Therefore one message is broadcasted to multiple recipients. If no subscription of a receiver is available when a message is about to be delivered, the message is discarded. If a messaging client is disconnected for a period of time, it will miss all messages transferred during this period.

3 Problem Description

The OSGi Alliance has an elegant approach to services & remote-services model via which local services, perhaps expressed by DS & Blueprint, may be simply made visible to clients in remote OSGi frameworks.

However, unlike Remote Services, the OSGi Alliance has no coherent approach to the support of distributed messaging / events. Given the increased mindshare in development communities – this driven by Cloud Computing and use of Actor type patterns, this is an important omission and is the focus of this RFP.

This is doubly surprising given that a number of existing OSGi specifications would benefit from such specifications:

- RSA might leverage a Distributed Eventing implementations for Service Discovery Events – i.e. to Announce/Publish local Service endpoints and Subscribe/Discover Remote Services endpoint events.
- A version of local Event Admin might leverage Distributed Eventing to distribute Events to remote frameworks.
- ConfigAdmin – local ConfigAdmin services might be update by remote Configuration Events.

3.1 Hybrid Interactions & Contracts

As previously suggested synchronous Services and asynchronous events represent two ends of a continuous spectrum of interaction behaviors.

Examples include:

- Service announcements and discovery (i.e. Service Events) are already used by RSA – though the implementation is viewed as RSA specific and effectively isolated from the rest of the OSGi framework – unless one treats an entity as a Service. (e.g. example in Cloud EcoSystems – RFC-183).

- We may wish to associate an event channel with an advertised Service – for communicating rapidly changing properties.
- Alternatively we may wish to have a synchronous Service interaction with users of a Topic – perhaps to change the rate/throttle message flow or change recovery behavior.

Defining a generic approach to Contracts or Service Level Agreements is still mostly an area of research – especially for non-functional properties (NFP) – (see SLAng, CQML etc).

However the start of a coherent strategy for OSGi is suggested by P Kriens & BJ Hargrave who have argued that – from a Service perspective – interactions should be expressed in-terms of *contracts*; each participant having its own role with respect to the interaction contract: i.e.

- what the participant is expected to provide
- what the participant can expect from the other participants.

P. Kriens has said: "A contract is just the agreed set of interactions between modules. With Service based interactions one tends to think in terms of interfaces... e.g. the CreditCheck service provides method calculateRating(). This is a simplistic contract between one provider (i.e. the credit rating provider) and the consumers; and it doesn't appear to support asynchronous interaction because invocation always originates from a consumer.

Instead consider a contract based on a group of interfaces; this is somewhat more powerful as each participant can provide some interfaces and consume other interfaces. For example a stock exchange: the exchange itself provides the OrderEntry, and other participants provide ExecutionListeners or MarketDataProviders. Hence, the 'contract' is not with a single Java interface but with a coherent collection of interfaces: in other words a package.

The 'contract' concept - expressed as a data transfer object (DTO) - may span JVM boundaries and provide a consistent approach for; synchronous (simple interface – DTO defines rich set of service properties), asynchronous (DTO defines multiple interfaces) and event based (DTO defines event / message format) based interactions."

While not defining a 'Contract' – DTO's provide generic foundations upon which 'Contract' descriptions may be created.

Distributed Eventing should naturally comply with this philosophy, as for Distributed Eventing the interaction '*contract'* is simply a combination of:

- The structure of the message / event
- The Topic
- The SLA provided by the Distributed Eventing implementation

Data transfer object specification (RFC 185) provides a natural natural representation for the structural payload of a distributed message/event. Meanwhile R5 *Capabilities* provide the natural mechanism via which a user may selected the appropriate Distributed Eventing implementation with respect to required SLA: (in-order delivery, durable or transient etc).

3.2 Issues with current Event Admin

It should be noted that the following issues exist with the current Event Admin specification. There is no concept of 'contract' and the messages are untyped, so each participant has to continually work out what kind of message it has received, validate it, handle errors and missing info, work out what it should send in response.

- Current Event Admin only specifies how to send and receive events

- What to do after receiving an Event is unspecified...
- Current Event Admin events are maps, where the values can be anything - Java's instanceof operator to find out the type. Does this / should this / be modernized to be DTO centric?

This is fine if we don't want to go to the trouble of defining a contract for a particular interaction, but the risk is that modules become *more* tightly coupled because of hidden assumptions about the form of events they exchange. Also Event Admin is missing features such as the ability to send a point-to-point reply to a specific message, perhaps to a specific endpoint or subset of endpoints (perhaps via correlation IDs).

For these reasons it may not be possible to repurpose the existing Event Admin since it is already designed for a certain set of local use-cases, and there may be backwards compatibility concerns. Hence a completely new distributed eventing design may be required that might optionally replace or complement the local Event Admin service.

3.3 Event Pipelines

The ReactiveX effort provides an API for event stream processing, where “Observers” have events pushed to them, and may publish the event on, or publish another related event as a result. In general this programming model leads to the creation of event “pipelines” through which events flow and are transformed.

This model is effectively a “push based” version of the Java 8 Streams API, which provides a functional pipleline for operating on Collections. The Streams API is inherently “pull based” as it relies on iterators/spliterators to “pull” the next entry from the stream. This is the primary difference between synchronous and asynchronous models. In an asynchronous world entries are pushed into the pipeline.

The other key difference between a pull-based and push based architecture is that pull-based models inherently throttle themselves. A slow part of the pipeline consumes the thread of execution, preventing more events from being created and overloading the system. In a push-based model the non-blocking nature forces “extra” events to be queued. Fast producers can easily overwhelm slow consumers. To combat this asynchronous systems introduce “back-pressure”. Back-pressure is used to indicate that an event source should slow down its event production to avoid overwhelming the consumer.

3.3.1 Buffering and Circuit breakers

An important part of stream processing is the use of buffering. Importantly, buffers provide an opportunity for thread switching in the asynchronous pipeline. This allows event producing threads to be returned to the event source without forcing them to execute the entire pipeline.

Buffers also provide an opportunity to create “circuit breakers”. Event storms occur when a large number of events occur in a short time, and can overwhelm the system. Buffering policies can move the system into a “blocking” state, or can simply disconnect the listener by “breaking” the pipeline. This is known as circuit breaker behaviour.

4 Requirements

DE010 – The solution MUST allow the sending of asynchronous messages to remote recipients.

DE012 – The solution MUST support a one-to-many, pub-sub/topic messaging semantic.

DE015 – The solution MUST support a one-to-one, queue messaging semantic.

DE020 – The solution MUST be independent of messaging technology used. This may be message broker based, peer-to-peer using a centralized approach or otherwise.

DE030 – The solution MUST allow implementations to advertise their supported Qualities of Service.

DE040 – The solution MUST provide a mechanism to select an Event Service provider based on its provided QoS.

DE042 – The solution SHOULD define a list of well-known QoS. Implementations MUST NOT be required to support all of these well known QoS.

DE045 – An implementation MUST be allowed to provide additional proprietary Qualities of Service.

DE047 – The solution MUST enable the message sender to specify the actual QoS used for sending a certain message.

DE048 – The solution MUST provide a facility for failure detection and/or reporting in cases where the requested Quality of Service cannot be satisfied.

DE050 – Events / Messages MUST be language agnostic – enabling a remote non-Java party to participate; e.g. C/C++ OSGi based agents.

DE055 – The solution MAY define a standard message encoding, for example using XML, JSON and/or other technology if appropriate.

DE060 – The solution MUST provide the means for point-to-point based communications for example to allow replies to specific messages – an event targeted to a specific node.

DE080 – The solution MUST provide the means to obtain information on the sender of an event e.g. bundleID, Framework UUID, SubSystem name. This information MAY be incomplete if the message didn't originate in an OSGi framework.

DE085 – The solution SHOULD provide the means to discover available Topics and Queues..

DE087 – The solution MUST allow certain Topics and Queues to not be advertised in the discovery mechanism.

DE088 – The solution SHOULD allow certain messages to be hidden from potential receivers.

DE090 – The solution SHOULD NOT prevent an implementation from providing a basic distribution solution for the existing Event Admin. While this will not provide all features of a Distributed Eventing solution, it is shown to be useful in certain contexts.

5 Technical Solution

A key part of the event processing model is defining how the events are processed. As OSGi R7 is moving to Java 8 the API design should enable idiomatic use of lambda expressions and functional programming techniques.

5.1 Using DistributedEventAdmin to Send DistributableEvents

The Distributed Event Admin service provides a simple way to broadcast events complete with information about the sender.

5.1.1 Simple Broadcast

An event source can send events programatically using the `DistributedEventAdmin` service's `publishEvent()` methods. These specify the topic to which the event should be published, and the data associated with the event. Distrbutable Events may be forwarded to a topic using the publish command.

A topic is a string consisting of one or more alphanumeric character sequences separated by '/' characters. Specifically:

topic ::= alphanum+ ('/' alphanum+)*

Simple broadcast events contribute to an infinite event stream that may be published by multiple event sources. As the stream is infinite no close or error events will be sent to or received from the stream. If a client attempts to send a close or error message then an IllegalArgumentException must be thrown by the `DistributedEventAdmin` service.

5.1.2 Broadcast Sessions

Event sources may also send events using an `EventPublisherSession` created by the `DistributedEventAdmin` service. Session-based publication of events is more powerful than simple broadcast for two reasons:

- Publishers may send end of stream or error messages to session-aware consumers, indicating that the stream has terminated. Non session-aware consumers will see the data events from the publisher as contributions to their infinite stream of events, but will not see close or error events.
- Publishers may register an `AsyncEventConsumer` as a message handler for reply messages. The replyTo field of the DistributableEvent will be automatically populated so that event recipients may send replies as appropriate

Sessions are associated with a single topic that is defined when the session is created. When a session is closed a close message is implicitly sent to any remaining active consumers.

5.1.3 Private topics

The `DistributedEventAdmin` service may also be used to create private topics. A private topic has an automatically generated name, and can be identified because unlike a normal topic name it starts with a '.' character.

Once a private topic has been created it can be used like a normal topic to send events. Event receivers must be notified of the topic name using some mechanism in order to listen to it. In this way an event stream can be delivered to a particular listener or set of listeners without the events being generally available.

5.1.4 Delivery Notifications

Sometimes it is important for the sender of an event to know when the event has been received by all consumers. In this case the `publishEventWithNotification()` methods return a promise which will resolve when all of the consumers have been notified. In general these methods should be avoided, as they add a significant performance penalty to the event distribution system.

5.2 Using the Event Admin whiteboard

The EventAdmin service supports two whiteboards, one for consuming events, and the other for publishing events.

5.2.1 Simple Event Consumers

To consume events from `DistributedEventAdmin` a consumer must register a `PushEventConsumer` or `DistributableEventConsumer` service with the following properties:

- event.topics : A String+ property indicating the topic(s) to which this AsyncEventConsumer is listening
- event.type : A String property indicating the name of the data type that this consumer expects to receive

If the consumer whiteboard service advertises the `DistributableEventConsumer` service interface then it will be delivered the entire DistributableEvent. If the consumer only registers the PushEventConsumer interface then the `DistributableEvent` will be unwrapped, and only the payload forwarded to the consumer.

Event consumers are permitted to be lazy. The `DistributedEventAdmin` service will only get the service object when the first event is to be delivered to it.

If the event consumer returns a negative back-pressure on delivery of an event then the consumer object is discarded by the `DistributedEventAdmin` service and will not be called again with subsequent events.

If the event consumer returns positive back-pressure then the `DistributedEventAdmin` must delay delivery of the next event until at least that many milliseconds have passed, queueing the events if necessary. If the event queue becomes too large then the `DistributedEventAdmin` service may disconnect the consumer by sending an error message containing an `IllegalStateException` to it and discarding the service. The service object must then not be called for subsequent events.

5.2.2 Wildcard topics

A wildcard topic name is one which ends in one or two '*' characters.

If a wildcard topic name ends with a single * then it matches any topic with the same name up to the * character and no '/' characters after that. If a wildcard topic ends with two * characters then it matches all topics with the same name up to the first * character, regardless of any subsequent / characters.

A wildcard may not be used to match a private topic

Examples:

Wildcard topic	Topic	Matches

foo/bar*	foo/bar	Yes
	foo/bark	Yes
	foo/baa	No
	foo/bar/foobar	No
foo/bar**	foo/bar	Yes
	foo/bark	Yes
	foo/baa	No
	foo/bar/foobar	Yes

5.2.3 Session-aware Event Consumers

An event consumer may be session-aware. A session-aware event consumer is one that opts in to receiving lifecycle events (close and error) in addition to data events, and will respond appropriately to those events.

The service property that controls the session awareness of the consumer is “event.lifecycle”. Allowable values are “UNIFIED” and “SESSION”. The default value of the event.lifecycle property depends upon the scope of the service. Prototype scope services have a default lifecycle of “SESSION”, bundle and singleton scope services have a default scope of “UNIFIED”.

If a consumer's lifecycle is “UNIFIED” then only one instance of the consumer service is obtained by the `DistributedEventAdmin` service. No session lifecycle events are sent to the service, and all data events from all sessions are sent to the single consumer instance.

If a consumer's lifecycle is “SESSION” then the `DistributedEventAdmin` service should attempt to lazily obtain an instance of the consumer service per sending session, including one for the “default” session, as data events arrive. If the consumer service is not prototype scope then the existing service object must be reused. Data events from a session must be delivered to the instance associated with that session. If a lifecycle event arrives for the service object then it must be delivered to the relevant service object. If the service object has no further active sessions ongoing then it must be released by the `DistributedEventAdmin` service.

5.2.4 Whiteboard Event Sources

Events may be published to `DistributedEventAdmin` by whiteboard services as well as by making direct calls on the DistributedEventAdmin service. To register as a whiteboard service the event source must publish an `AsyncEventSource` service with the “event.topic” property. The value of this property is the topic to which events will be published.

Whiteboard event sources will only be obtained and opened when at least one consumer for the topic is available. When a consumer is available the `DistributedEventAdmin` service must start a new publishing session for the `AsyncEventSource` and publish any emitted events using this session.

5.3 Distributed Events with EventAdmin

DistributedEventAdmin will function normally in a single framework, publishing typed events to local consumers, however it is also able to distribute events to other DistributedEventAdmin instances on remote machines. The remote node discovery mechanism is unspecified, as is the transport used to distribute the events. What is required is that the DistributedEventAdmin implementation is able to support the serialization of Serializable Java types. Implementations may also support additional serialization formats such as JSON or Google Protocol Buffers. Once serialised the data may be published using OSGi Remote Services, JMS, MQTT, or some other transport.

5.3.1 Qualities of Service

When used for distributing event the eventing infrastructure is necessarily less reliable than an in VM solution, however there are trade-offs that can be made to improve reliability at the expense of performance and/or latency. DistributedEventAdmin implementations must advertise the the reliability and other qualities of service that they support.

QoS values are as follows:

- TRANSIENT – Events will not be stored persistently, and may be lost in the event of a system failure
- PERSISTENT – Events will be stored persistently so that redelivery can be attempted after recovery from failure
- AT_MOST_ONCE – Events will be delivered at most once to the consumer, but may never arrive
- AT_LEAST_ONCE – Events will be delivered to the consumer, but multiple copies may be delivered
- EXACTLY_ONCE – Events will be delivered to the consumer exactly once
- IN_ORDER – Events will be delivered to the consumers in the same order in which they were published. Note that ordering only applies relative to events published by a single thread. If multiple threads are publishing then their events may be interleaved.
- OUT_OF_ORDER – Events may be delivered out of order as a result of network failures or internal optimisations.

The supported QoS policies must be advertised by the DistributedEventAdmin service using the “event.delivery.qos” property.

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

7.1 The Distributed Eventing API

OSGi Javadoc

11/12/15 12:21 PM

Package Summary		Page
org.osgi.service.distributedeventing		19

Package org.osgi.service.distributedeventing

```
@org.osgi.annotation.versioning.Version(value="1.0.0")
```

Interface Summary		Page
DistributableEventConsumer	An event source.	*22*
DistributedEventAdmin		*24*
EventPublisherSession		*27*

Class Summary		Page
DistributableEvent	This class encapsulates event data for publishing via EventAdmin in an `org.osgi.util.pushstream.PushEvent`	*20*

Enum Summary		Page
DistributableEventConsumer.ConsumerLifecycle		*23*

Class DistributableEvent

org.osgi.service.distributedeventing

```
java.lang.Object
  └ org.osgi.service.distributedeventing.DistributableEvent
```

All Implemented Interfaces:
Serializable

```
final public class DistributableEvent
extends Object
implements Serializable
```

This class encapsulates event data for publishing via EventAdmin in an `org.osgi.util.pushstream.PushEvent`

Constructor Summary	Page
DistributableEvent(String topic, long originatingBundle, UUID originatingFramework, String replyTo, String correlationId, T data)	20

Method Summary		Page
String	getCorrelationId() The correlation id for this message	21
T	getData() The raw event data	21
long	getOriginatingBundle() The id of the bundle which sent this event	21
UUID	getOriginatingFramework() The framework from which this event originated	21
String	getReplyTo() The replyTo location for this event	21
String	getTopic() The topic for this Event	20

Constructor Detail

DistributableEvent

```
public DistributableEvent(String topic,
                          long originatingBundle,
                          UUID originatingFramework,
                          String replyTo,
                          String correlationId,
                          T data)
```

Method Detail

getTopic

```
public String getTopic()
```

The topic for this Event

---

getOriginatingBundle

```
public long getOriginatingBundle()
```

The id of the bundle which sent this event

---

getOriginatingFramework

```
public UUID getOriginatingFramework()
```

The framework from which this event originated

---

getReplyTo

```
public String getReplyTo()
```

The replyTo location for this event

May be null if the event sender did not register to receive replies

---

getCorrelationId

```
public String getCorrelationId()
```

The correlation id for this message

---

getData

```
public T getData()
```

The raw event data

Interface DistributableEventConsumer

org.osgi.service.distributedeventing

Type Parameters:
T - The payload type
All Superinterfaces:
org.osgi.util.pushstream.PushEventConsumer<DistributableEvent<T>>

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface DistributableEventConsumer
extends org.osgi.util.pushstream.PushEventConsumer<DistributableEvent<T>>
```

An event source. An event source can open a channel between a source and a consumer. Once the channel is opened (even before it returns) the source can send events to the consumer. A source should stop sending and automatically close the channel when sending an event returns a negative value, see AsyncEventConsumer.ABORT. Values that are larger than 0 should be treated as a request to delay the next events with those number of milliseconds. Distribution Providers can recognize this type and allow the establishment of a channel over a network.

Nested Class Summary		Page
static enum	DistributableEventConsumer.ConsumerLifecycle	23

Field Summary		Page
String	CONSUMED_EVENT_TOPICS	22
String	EVENT_TYPE	22
String	LIFECYCLE The lifecycle of the consumer.	22

Fields inherited from interface org.osgi.util.pushstream.PushEventConsumer
ABORT, CONTINUE

Methods inherited from interface org.osgi.util.pushstream.PushEventConsumer
accept

Field Detail

CONSUMED_EVENT_TOPICS

```
public static final String CONSUMED_EVENT_TOPICS = "event.topics"
```

EVENT_TYPE

```
public static final String EVENT_TYPE = "event.type"
```

LIFECYCLE

```
public static final String LIFECYCLE = "consumer.lifecycle"
```

The lifecycle of the consumer. Prototype scope services default to a consumer lifecycle

Enum DistributableEventConsumer.ConsumerLifecycle

org.osgi.service.distributedeventing

```
java.lang.Object
  └ java.lang.Enum<DistributableEventConsumer.ConsumerLifecycle>
      └ org.osgi.service.distributedeventing.DistributableEventConsumer.ConsumerLifecycle
```

All Implemented Interfaces:
Comparable<DistributableEventConsumer.ConsumerLifecycle>, Serializable

Enclosing class:
DistributableEventConsumer

```
public static enum DistributableEventConsumer.ConsumerLifecycle
extends Enum<DistributableEventConsumer.ConsumerLifecycle>
```

Enum Constant Summary	Page
GLOBAL	23
PER_SESSION	23

Method Summary		Page
static DistributableEventConsumer.ConsumerLifecycle	**valueOf**(String name)	23
static DistributableEventConsumer.ConsumerLifecycle[]	**values**()	23

Enum Constant Detail

GLOBAL

```
public static final DistributableEventConsumer.ConsumerLifecycle GLOBAL
```

PER_SESSION

```
public static final DistributableEventConsumer.ConsumerLifecycle PER_SESSION
```

Method Detail

values

```
public static DistributableEventConsumer.ConsumerLifecycle[] values()
```

---

valueOf

```
public static DistributableEventConsumer.ConsumerLifecycle valueOf(String name)
```

Interface DistributedEventAdmin

org.osgi.service.distributedeventing

```
@org.osgi.annotation.versioning.ProviderType
public interface DistributedEventAdmin
```

Method Summary		Page
String	**createPrivateTopic**() A private topic is a topic with an automatically generated name that start with a '.' character.	26
EventPublisherSession<T>	**createSession**(String topic, Class<T> type) Create a session for sending events to a given topic.	25
EventPublisherSession<T>	**createSession**(String topic, Class<T> type, org.osgi.util.pushstream.PushEventConsumer<U> responseHandler, Class<U> responseType) Create a session for sending events to a given topic.	26
org.osgi.util.pushstream.PushStream<T>	**createSubscription**(String s, Class<T> type) Create a subscription to the named topic.	26
void	**publish**(DistributableEvent<?> e) Asynchronously send a single event for a given topic using the "default" session.	25
void	**publishEvent**(String topic, Object payload) Asynchronously send a single event for a given topic using the "default" session.	24
void	**publishEvent**(String topic, String correlationId, Object payload) Asynchronously send a single event for a given topic using the "default" session.	24
org.osgi.util.promise.Promise<Void>	**publishEventWithNotification**(String topic, Object payload) Asynchronously send a single event for a given topic using the "default" session.	25
org.osgi.util.promise.Promise<Void>	**publishEventWithNotification**(String topic, String correlationId, Object payload) Asynchronously send a single event for a given topic using the "default" session.	25
org.osgi.util.promise.Promise<Void>	**publishWithNotification**(DistributableEvent<?> e) * Asynchronously send a single event for a given topic using the "default" session.	25

Method Detail

publishEvent

```
void publishEvent(String topic,
                  Object payload)
```

Asynchronously send a single event for a given topic using the "default" session. The payload will automatically associated with DistributableEvent data.

---

publishEvent

```
void publishEvent(String topic,
                  String correlationId,
                  Object payload)
```

Asynchronously send a single event for a given topic using the "default" session. The payload will automatically associated with DistributableEvent data.

publish

```
void publish(DistributableEvent<?> e)
```

Asynchronously send a single event for a given topic using the "default" session.

publishEventWithNotification

```
org.osgi.util.promise.Promise<Void> publishEventWithNotification(String topic,
                                                                 Object payload)
```

Asynchronously send a single event for a given topic using the "default" session. The payload will automatically associated with DistributableEvent data.

This version of the publish method returns a promise that will resolve when all handlers have been notified of the event. This includes handlers that are connected remotely. If no notification is needed then the publishEvent(String, Object) method will offer superior performance.

publishEventWithNotification

```
org.osgi.util.promise.Promise<Void> publishEventWithNotification(String topic,
                                                                 String correlationId,
                                                                 Object payload)
```

Asynchronously send a single event for a given topic using the "default" session. The payload will automatically associated with DistributableEvent data.

This version of the publish method returns a promise that will resolve when all handlers have been notified of the event. This includes handlers that are connected remotely. If no notification is needed then the publishEvent(String, String, Object) method will offer superior performance.

publishWithNotification

```
org.osgi.util.promise.Promise<Void> publishWithNotification(DistributableEvent<?> e)
```

* Asynchronously send a single event for a given topic using the "default" session.

This version of the publish method returns a promise that will resolve when all handlers have been notified of the event. This includes handlers that are connected remotely. If no notification is needed then the publish(DistributableEvent) method will offer superior performance.

createSession

```
EventPublisherSession<T> createSession(String topic,
                                       Class<T> type)
```

Create a session for sending events to a given topic. The session may be temporarily or permanently closed by calling its lifecycle methods. If the DistributedEventAdmin instance is released, or the DistributedEventAdmin service is unregistered then the Session will automatically close.

createSession

```
EventPublisherSession<T> createSession(String topic,
                                       Class<T> type,
                                       org.osgi.util.pushstream.PushEventConsumer<U> responseH
andler,
                                       Class<U> responseType)
```

Create a session for sending events to a given topic. The session may be temporarily or permanently closed by calling its lifecycle methods. If the `DistributedEventAdmin` instance is released, or the `DistributedEventAdmin` service is unregistered then the Session will automatically close.

This session also registers a handler for "reply" events on an automatically generated "private" topic. This handler can be reached by publishing an event to the `DistributableEvent.getReplyTo()` topic. Normally reply events will also contain a correlationId

---

createSubscription

```
org.osgi.util.pushstream.PushStream<T> createSubscription(String s,
                                                          Class<T> type)
```

Create a subscription to the named topic. Typically used to pre-register a listener for a private topic before the sender begins sending events.

---

createPrivateTopic

```
String createPrivateTopic()
```

A private topic is a topic with an automatically generated name that start with a '.' character. Aside from the first character, the format of a private topic name is unspecified.

A private topic name may be shared remotely and so must not be reused. In particular two calls to `createPrivateTopic()` must not return the same value, even after the VM has been restarted

Interface EventPublisherSession

org.osgi.service.distributedeventing
All Superinterfaces:
AutoCloseable, Closeable, org.osgi.util.pushstream.PushEventSource<DistributableEvent<T>>, org.osgi.util.pushstream.SimplePushEventSource<DistributableEvent<T>>

```
@org.osgi.annotation.versioning.ProviderType
public interface EventPublisherSession
extends org.osgi.util.pushstream.SimplePushEventSource<DistributableEvent<T>>
```

Method Summary		Page
org.osgi.util.promise.Promise<Void>	**closeWithNotification**()	28
org.osgi.util.promise.Promise<Void>	**endOfStreamWithNotification**()	28
org.osgi.util.promise.Promise<Void>	**errorWithNotification**(Exception e)	28
String	**getTopic**()	28
void	**publishEvent**(T t)	27
void	**publishEvent**(T t, String correlationId)	27
org.osgi.util.promise.Promise<Void>	**publishEventWithNotification**(T t)	28
org.osgi.util.promise.Promise<Void>	**publishWithNotification**(DistributableEvent<T> event)	28
org.osgi.util.promise.Promise<Void>	**publishWithNotification**(T t, String correlationId)	28

Methods inherited from interface org.osgi.util.pushstream.SimplePushEventSource
close, endOfStream, error, isConnected, publish

Methods inherited from interface org.osgi.util.pushstream.PushEventSource
open

Method Detail

publishEvent

```
void publishEvent(T t)
```

---

publishEvent

```
void publishEvent(T t,
                  String correlationId)
```

---

publishWithNotification

```
org.osgi.util.promise.Promise<Void> publishWithNotification(DistributableEvent<T> event)
```

---

publishEventWithNotification

```
org.osgi.util.promise.Promise<Void> publishEventWithNotification(T t)
```

---

publishWithNotification

```
org.osgi.util.promise.Promise<Void> publishWithNotification(T t,
                                                            String correlationId)
```

---

closeWithNotification

```
org.osgi.util.promise.Promise<Void> closeWithNotification()
```

---

endOfStreamWithNotification

```
org.osgi.util.promise.Promise<Void> endOfStreamWithNotification()
```

---

errorWithNotification

```
org.osgi.util.promise.Promise<Void> errorWithNotification(Exception e)
```

---

getTopic

```
String getTopic()
```

---

8 Considered Alternatives

8.1 Basic Remoting of EventHandler services

The existing eventing mechanism in OSGi is the Event Admin service. Event Admin is used to deliver events, either synchronously or asynchronously, to listeners registered in the local framework.

The simplest conceivable solution for distributed eventing involves using the existing Event Admin listener interface, and registering it as a Remote Service. This Remote service will then receive any and all events sent via Event Admin.

To support the point-to-point behaviours required a client could target particular service instances based on their service properties. All remote services are registered with information about the framework they originate from, and their service id within that framework. This allows any remote service to be uniquely identified and selected.

Asynchronous invocation of Event Handlers can be achieved by making use of the new OSGi Async Service. This allows Event Handler services to be called asynchronously, either in a fire-and-forget fashion or receiving a notification on completion.

One possible enhancement to this model would be to add support for the async service and/or Promise pattern to the Event Admin service. The client could potentially provide configuration describing whether redelivery should be attempted, or providing actions to run on failure.

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]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. The Power of Events'. D. C. Luckham. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[4]. Extending OSGi by Means of Asynchronous Messaging - Master Thesis, Marc Schaaf, September 2009, University of Applied Sciences and Arts Hannover. http://schaaf.es/docs/master_thesis_marc_schaaf_Extending_OSGi_by_Means_of_Asynchronous_Messaging.pdf

10.2 Author's Address

Name	Tim Ward
Company	Paremus
Address	
Voice	
e-mail	tim.ward@paremus.com

Name	Peter Kriens
Company	aQute
Address	
Voice	
e-mail	peter.kriens@aqute.biz

Name	Richard Nicholson
Company	Paremus Ltd
Address	107-111 Fleet Street London
Voice	
e-mail	richard.nicholson@paremus.com

Name	Marc Schaaf
Company	
Address	
Voice	
e-mail	marc@marc-schaaf.de

Name	David Bosschaert
Company	Adobe
Address	
Voice	
e-mail	bosschae@adobe.com

Name	Carsten Ziegeler
Company	Adobe
Address	
Voice	
e-mail	cziegele@adobe.com

Name	Graham Charters
Company	IBM
Address	
Voice	
e-mail	charters@uk.ibm.com

10.3 Acronyms and Abbreviations

10.4 End of Document