RFC 97 - Generic Event Mechanism

Final

37 Pages

Abstract

This RFC describes a generic publish-subscribe event model for MEG.

0 Document Information

0.1 Table of Contents

All Page Within This Box

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 RFC 2119 [1].

`Source code is shown in this typeface.`

Design notes are 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	August 31, 2004	Kevin Riff
2nd Draft	October 12, 2004	Kevin Riff Incorporated comments received via e-mail and from the CPEG meeting in Amsterdam, September 2004 • Accepted most comments and revisions • Did NOT accept request to reject requirement **Error! Reference source not found.** pending discussion at the next MEG face-to-face meeting.

All Page Within This Box

Revision	Date	Comments
3rd Draft	November 17, 2004	Kevin Riff Incorporated comments received at MEG meeting in Nice, October 2004 • Removed requirement **Error! Reference source not found.** • Added a method for case-sensitive matching in the Filter interface • Clarified exactly what guarantees are made in regards to the order of event delivery
4th Draft	January 19, 2005	Kevin Riff, Espial, kriff@espial.com Incorporated comments received at CPEG meeting in Boca Raton, FL, December 2004 • Removed requirements that are not met by this design • Removed sections for MEG-related events. These should be moved to one or more MEG RFCs. • Renamed API classes to EventAdmin, Event and EventHandler • Added mappings for all of the remaining events from the R3 spec and RFC 103 • Changed the separator character for topics to slash. • Added caution re mutable objects in the event's properties • Added note about handlers not taking too long • Updated JavaDoc APIs
Final Review Draft	February 11, 2005	Kevin Riff, Espial, kriff@espial.com Incorporated comments received at CPEG meeting in Newark, CA, January 2005 • Small clarification to rules for asynchronous delivery. • Tracked down and fixed uses of the old "event channel" terminology • Updated JavaDocs
Final	27 May 2005	No changes BJ Hargrave, hargrave@us.ibm.com

All Page Within This Box

1 Introduction

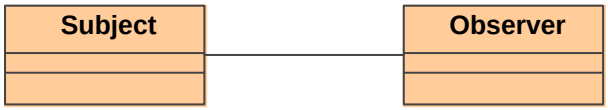
This RFC describes a generic event mechanism for both CPEG and MEG. Most of this material was originally included in the MEG Application Model RFC [7] but at the Burlington meeting [8] it was decided to create a separate RFC for the event mechanism.

The requirements for this RFC come from two sources. The MEG's requirements are contained in RFP 52 [4] while requirements from CPEG's perspective are outlined in RFP 59 [5].

2 Application Domain

2.1 The Observer Pattern

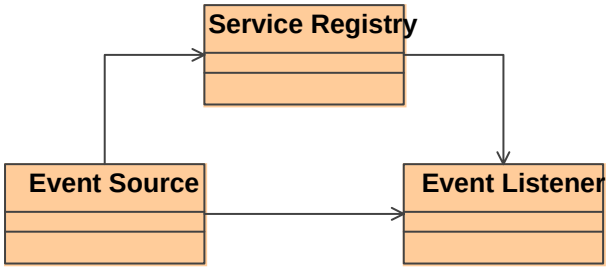
The Observer pattern defines a one-to-many dependency between a *subject* and the *observers* that must be notified when the subject's state changes. Each observer registers itself with the subject. When the subject's state changes, it notifies each registered observer. This couples the subject and the observer because each must know about the other.



In OSGi, either the subject or the observer may be in a bundle that could be removed at any time. Since the framework is unaware of the dependency it cannot clean it up automatically. Therefore both the subject and the observer must monitor the other and take appropriate action if the other is removed from the system. In practice, getting this right is tricky.

2.2 The Whiteboard Pattern

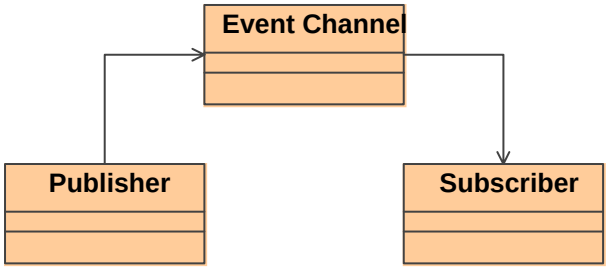
The Whiteboard pattern [3] makes use of the OSGi service registry to manage the dependencies between the event source and its listeners. Each event listener registers itself as a service in the OSGi service registry. The event source queries the registry to obtain the list of listeners, then dispatches the event directly to each one.



In this model, the event source and event listener are less coupled than before because the event listener has no direct knowledge of the event source. However, the event source must still monitor the service registry and take action when a listener is either registered or unregistered. OSGi provides the `ServiceTracker` utility class to handle the necessary details.

2.3 The Publish-Subscribe Pattern

The Publish-Subscribe pattern decouples event sources from event listeners by interposing an *event channel* between them. The event source posts events to the channel, which identifies which listeners need to be notified and then handles the notification process.



In this model, the event source and event listener are completely decoupled because neither has direct knowledge of the other. The complicated logic of monitoring changes in the event sources and event listeners is completely contained within the event channel. This is highly advantageous in an OSGi environment because it simplifies the process of both sending and receiving events.

There are two main flavors of publish-subscribe mechanisms: subject-based and content-based. In subject-based systems, subscribers register to receive events that were sent to a particular topic (or set of topics). Content-based systems examine the entire contents of the event to determine which subscribers should receive it. They are more flexible than subject-based systems but the rules for matching events to subscribers are more complicated so some thought needs to be given to the efficiency of the system.

3 Problem Description

The following is copied from RFP 59:

3.1 Complexity

Recently the Core Platform Expert Group (CPEG) has begun to receive feedback that the OSGi programming model is sometimes too complex. The dynamic nature of services within the OSGi environment requires that programmers abandon certain basic assumptions that are implicit in other kinds of programming. For example, they cannot assume that a service will remain valid for as long as they hold a reference to it. Applications-level programmers would like to be insulated from these complications as much as possible.

Nearly all the bundles in an OSGi framework must deal with events, either as an event source or an event listener. The dynamic nature of the OSGi environment introduces several problems because both event sources and event listeners can "go away" at any time. The whiteboard pattern [3] was created to address these problems but more work is needed to simplify implementations.

Something is required that will simplify the process of event delivery for programmers of both event sources and event listeners.

3.2 Asynchronous Event Delivery

Asynchronous event delivery creates additional complications because event sources and event listeners may disappear from the framework between the time that the event is triggered and delivery is actually attempted. The whiteboard pattern does not address these issues.

Also, asynchronous event delivery implies the use of one or more event threads. Bundles that need to asynchronously deliver events generally provide their own event thread(s). In a worst-case scenario, every bundle that is installed in the framework may need to create its own. This can lead to a lot of overhead to manage all of them.

A general event dispatch mechanism offers a chance to consolidate all of those threads into just a single one (or perhaps a small number of threads), saving a fair amount of resources. This is an important consideration for the sorts of embedded platforms that OSGi targets.

3.3 Security

The process of delivering events necessarily involves passing messages (and data) between different bundles. Listeners are generally invoked within the context of the event source, and are limited by its (lack of) privileges. If a listener must perform some sensitive operation, the R3 specification [9] recommends using the `doPrivileged` method to ensure that the listener executes with its full privileges. Knowing when and how to use `doPrivileged` is the mark of an experienced Java programmer and cannot be considered "simple". This is an example of how the Java 2 permission model can be overly complex.

4 Requirements

4.1 CPEG Requirements

The following requirements are from RFP 59 [5]:

4.1.1 General Requirements

[1]. MUST simplify the process of programming an event source.

[2]. SHOULD simplify the process of programming an event listener.

[3]. MUST automatically cleanup any dependencies between event sources and event listeners when the relevant bundles are stopped.

[4]. Removed.

[5]. Removed

[6]. SHOULD support MEG's "generic event model".

4.1.2 Event Delivery Requirements

[7]. MUST support synchronous event delivery.

[8]. MUST support asynchronous event delivery.

[9]. SHOULD limit the number of threads used for asynchronous event delivery. Ideally there should be only one event thread.

[10]. The number of event threads MAY be configurable.

[11]. A snapshot of the current list of event listeners MUST be collected when an asynchronous event delivery is triggered. Only event listeners on this list will be notified of the event.

[12]. MUST only notify event listeners belonging to bundles that are active when notification occurs.

[13]. Event listeners MUST only receive notifications for the event types for which it is registered. In this case "event type" MAY mean a class of event types. For example, an event listener may register to receive all `BundleEvents` but not `BundleEvent.STARTED` events specifically.

*This RFC *does* allow listeners to register for just `BundleEvent.STARTED` events so this requirement is exceeded.*

[14]. Removed

[15]. If event A is fired before event B, then all event listeners MUST return from processing event A before any event listener receives event B.

This requirement is met if both A and B are asynchronous events. But if either one is synchronous then we cannot guarantee any particular ordering. It could even happen that A will be delivered to some listeners, then B will be triggered and delivered, then the rest of A's listeners are notified.

4.1.3 Reliability Requirements

[16]. The event dispatcher MUST catch any exception thrown by an event listener, deal with it, and then continue dispatching events to the remaining event listeners.

[17]. The event dispatcher SHOULD log any exceptions that are thrown by event listeners.

[18]. The event dispatcher SHOULD NOT hold any Java monitors when it dispatches events to event listeners.

[19]. The event dispatcher MAY detect event listeners that do not return in a timely manner (e.g. have deadlocked or entered an endless loop), and automatically recover. Such recovery SHOULD include continuing delivery of the event to the remaining event listeners on another event thread.

4.1.4 Security Requirements

[20]. Removed

[21]. SHOULD require a secure permission to register an event listener

[22]. SHOULD require a secure permission to get an event listener

[23]. SHOULD only deliver events to event listener which have the necessary permissions

4.1.5 Non-functional Requirements

[24]. SHOULD use minimal resources

4.2 MEG Requirements

The following requirements are copied from RFP 52 [4]:

The section numbers have been adjusted to match those in the RFP to avoid confusion.

4.2.1 Notification infrastructure

UEs running within Mobile devices are typically designed to react to a lot of asynchronous events happening within the device as the device interacts with the external world. Some examples of such asynchronous events are – Radio Network Signal Strength change, Incoming SMS/MMS message, Incoming Call, Insertion/Removal of an accessory. Although not all UEs are expected to use these events, a common infrastructure for such an event notification infrastructure is necessary for the UEs to interoperate. Here are some of the characteristics of the event infrastructure.

4.2.1.1 Event classification by type

When a generic MEG framework event is generated, its type has to be conveyed to the consumer for high-level processing and dispatching. This type can be just a string, allowing the event consumer to determine whether the event was a timer going off due to a pre-scheduled activity, an external message arriving, or an accessory attached to the device.

All Page Within This Box

4.2.1.2 Wildcards in event types

In some cases similar actions (e.g launch of the same UE) have to occur when the event type corresponds to a certain pattern. One example is use of the Device Management Tree URI of the node as part of the event type reflecting changes made to the sub-tree. In this case using wildcard at the end of the event type string would enable uniform processing of all changes to a certain sub-tree.

In this RFC, the node would not be part of the "event type". Rather, the node is part of the content of the event. However, listeners can still specify which sub-tree(s) they are interested in using a content-based filter.

4.2.1.3 Event data passing

While the event type reflects at the high level the nature of the event, its details are very specific to the event source: for the timer it would be a timestamp, for a message the details would contain its body and headers, for a DMT change event the exact URI of the affected node(s) as well as type of change etc.

These details have to be passed to the processing application in an agreed and generic way, allowing the detail processing to occur

4.2.1.4 Processing events between Java and native applications

Some of the events meaningful for the Java-based MEG framework originate in the native code, and vice versa. An example of the former is likely to be a physical event, such as SIM card insert or attachment of an accessory. The latter may happen if a Java-based DM facility sends notification of a change in the browser configuration, and a native browser application has to adjust its state accordingly.

If I've understood the way that Motorola and Nokia want to use this feature, the native processes that receive and process these events probably won't be aware of Java in any way. Therefore, only the library that bridges the divide between Java and the native apps should rely on JNI.

4.2.1.5 Multiple event processing

The event processing mechanism has to have the ability to associate the same code with multiple events. For example, both SIM card insert and SIM card data change event have to result in invocation of the SIM-to-device synchronization code. Another example is an application reacting to changes in several sub-trees of the DMT

4.2.2 Specific notifications

4.2.2.1 Standard OSGi framework events

OSGi framework already has number of its own events defined, such as ServiceEvent, BundleEvent and FrameworkEvent. For uniformity of processing, these have to be mapped into generic event types, generated by listeners built into the OSGi framework. Such an approach is useful if the same code has to process multiple events, some of which are original OSGi framework events

4.2.2.2 MEG UEs lifecycle events

Removed

4.2.2.3 Logging event

Logging events are useful for allowing external alerts to be generated when certain error messages are logged.

4.2.2.4 SIM card events

Removed

4.2.2.5 SIM card data change events

Removed

4.2.2.6 Configuration data change events

Applications affected by configuration data change have to be able to receive notification of such an event. Configuration data may be application-specific (browser setup, email parameters and such) or system-wide (such as GUI settings)

4.2.2.7 Telephony/Call Handling events

Removed

4.2.2.8 Timer/scheduling events

Removed

4.2.2.9 Accessory attached/detached events

Removed

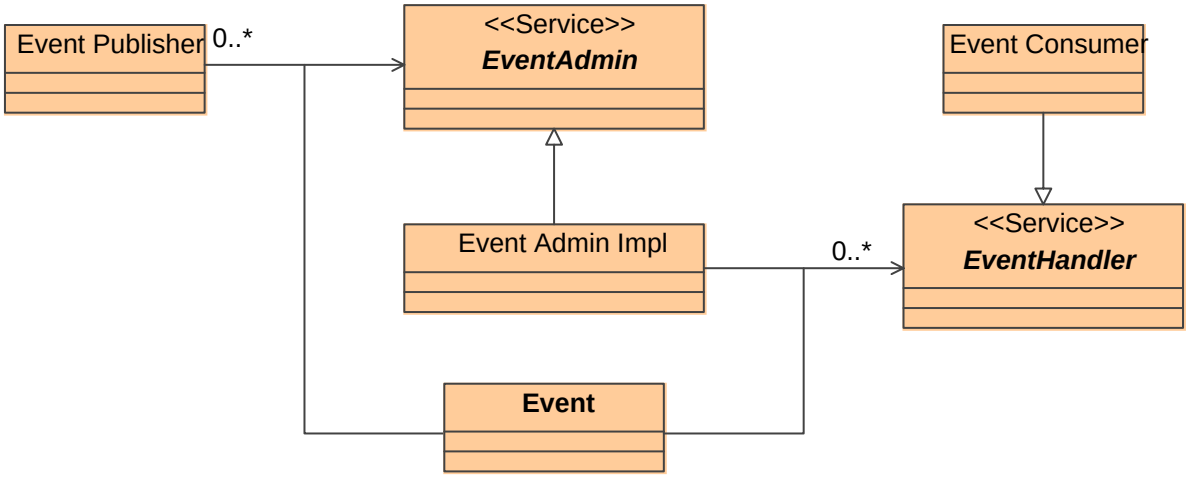
4.2.2.10 Resources low events

Removed

4.2.2.11 Messaging events

Removed

5 Technical Solution



Note: *In the diagram above, the entities in **bold** will be defined by this RFC. The entities in plain type-face are important conceptually but they are not specified by this RFC.*

5.1 Entities

5.1.1 Event

Events have the following attributes:

- A *topic* that describes what happened
- Zero or more *properties* that contain additional information about the event, represented as key-value pairs.

NOTE: The previous RFC included a "priority" and a "handling mode", however these may not actually be necessary. There are some fringe cases to support the idea but the feeling within MEG is that they may not be valid. For now, priority and handling mode are not requirements.

One of the major issues with the previous RFC, IMHO, is that the "event type" was being overloaded to do too much. It embodied not only what happened, but also who it happened to and sometimes other data as well. This made it very difficult to reason about the event type.

*This RFC replaces "event type" with a "topic" that describes what happened and nothing else. Additional data is represented as properties of the event. Event listeners may register a filter that operates on *any* of the event's properties (see section 5.1.3). This arrangement should be both easier to understand and more flexible.*

5.1.1.1 Topics

The topic of an event describes what happened. It should be fairly granular in order to give listeners the opportunity to register for just the events they are interested in. The topic should not include any other information, such as the source of the event or the data associated with the event.

The topic is intended to serve as a first-level filter for determining which listeners should receive the event. It's hoped that by using a simple, hierarchical namespace, event admin implementations should be able to use some sort of optimized data structure for holding the listeners that allows for very efficient dispatch operations.

Topics are arranged in a hierarchical namespace. Each level is defined by a token and levels are separated by slashes. More precisely, the topic MUST conform to the following grammar:

```
topic := token ( "/" token )*
```

Topics are case sensitive. As a convention, topics should follow the reverse domain name scheme used by Java packages.

This RFC will use the convention "fully/qualified/package/ClassName/ACTION". If necessary a psuedo-class-name will be used, even though the class does not actually exist.

5.1.1.2 Properties

Additional information about the event is provided as properties. The property name is a case-sensitive string and the value is an object.

A new case-sensitive matching method will be added to the `Filter` class so that the filtering logic does not need to be duplicated. See section 5.7.1.

The topic of the event is available as a property with the key "topic". This allows filters to include the topic as a condition if necessary.

The previous RFC included a payload which was described simply as a Java object. However, this does not allow listeners to register a filter based on the values contained within the payload. Also, it's not clear how an arbitrary Java object can be passed to a native process.

Although any Java object MAY be used as a property value, only strings and the eight primitive types (plus their wrappers) SHOULD be used. Other types cannot be passed to native processes, which is a strong requirement for MEG. See section 5.5 for details.

Note: The objects used as values of the event's properties should be immutable. If they are not immutable, then any handler that receives the event could change the value. Any handlers that received the event subsequently would see the altered value and not the value as it was when the event was sent.

5.1.2 Event Handler

Event handlers MUST be registered as services with the OSGi framework under the object class "org.osgi.service.event.EventHandler".

Event handlers SHOULD be registered with a property named "topic". The value being an array of strings that describe which topics the listener is interested in. An asterisk may be used as a trailing wildcard. Handlers which do not have a value for "topic" are treated as though they had the value {"*"}.

All Page Within This Box

More precisely, the value of each entry in the "topic" property must conform to the following grammar:

```
topic-filter := "*" | topic [ "/*" ]
```

Event handlers MAY be registered with a property named "filter". The value of this property MUST be a string containing an LDAP-style filter specification. Any of the event's properties may be used in the filter expression.

Each event handler is notified for any event which belongs to the topic(s) the handler has expressed an interest in. If the handler has defined a "filter" property then the properties of the event must also match the filter.

5.1.3 Event Admin Service

The Event Admin service must be registered as a service with the object class "org.osgi.service.event.EventAdmin".

The Event Admin service is a singleton. It should always be present and there should never be more than one instance. If there is ever more than one instance registered, then clients SHOULD use the highest ranking instance, or in the event of a tie, the first one to be registered.

What should clients do if the event admin is not available? Should they queue their own events and wait for the service to become available again? Should they simply discard the events? I think this can safely be left as an implementation detail.

The Event Admin service is responsible for tracking the registered listeners, handling event notifications and providing a thread for asynchronous event delivery.

5.1.4 Event Source

To fire an event, the event source must retrieve the Event Admin service from the OSGI service registry. Then it creates the event object and calls one of the event admin's methods to fire the event either synchronously or asynchronously.

5.2 Event Delivery

5.2.1 Synchronous Event Delivery

Synchronous event delivery is initiated by the `sendEvent` method. When this method is invoked, the Event Admin service determines which listeners should be notified of the event and then notifies each one in turn. The listeners may be notified in the caller's thread or in an event-delivery thread, depending on the implementation. In either case, all notifications will be completely handled before the `sendEvent` method returns to the caller.

Clients of this service will need to be coded defensively and assume that synchronous event notifications will be handled in a separate thread. That means they MUST NOT be holding any locks when they invoke the `sendEvent` method. Otherwise they increase the likelihood of deadlocks because Java monitors will NOT be reentrant. Of course this is good practice even when the event is dispatched in the same thread.

5.2.2 Asynchronous Event Delivery

Asynchronous event delivery is initiated by the `postEvent` method. When this method is invoked, the Event Admin service MUST determine which listeners are interested in the event. By collecting this list of listeners now, the event admin ensures that only listeners which were registered at the time the event was triggered will receive the event notification.

It's important to ensure that events are delivered in a well-defined order. Otherwise some funky things could happen. For example, if a thread posts events A and B then the listeners should not receive them in the order B, A. Note however that if A and B are posted by different threads at about the same time then we cannot make any guarantees about the order of delivery.

At this point, control returns to the caller.

At the same time or at some later point, the event admin will begin delivering the event on a different thread.

The Event Admin service MAY use more than one thread to deliver events. If it does then it must guarantee that each handler receives the events in the same order as they were posted. This ensures that handlers see events in a sensible order. For example, it would be an error to see a “destroyed” event before the corresponding “created” event. The number of event delivery threads MAY be configurable via some implementation-defined mechanism.

Before notifying each listener, the event delivery thread MUST ensure that the listener is still registered in the service registry. If it has been unregistered then the listener MUST NOT be notified of the event.

5.2.3 Order of Event Delivery

Asynchronous events are delivered in the order in which they arrive in the event queue. Thus if two events are posted by the same thread then they will be delivered in the same order (though other events may come between them). However, if two or more events are posted by different threads then the order in which they arrive in the queue (and therefore the order in which they are delivered) will depend very much on subtle timing issues. The event delivery system cannot make any guarantees in this case.

Synchronous events are delivered as soon as they are sent. If two events are sent by the same thread, one after the other, then they are guaranteed to be processed serially and in the same order. However, if two events are sent by different threads then no guarantees can be made. The events MAY be processed in parallel or serially, depending on whether or not the Event Admin service dispatches synchronous events in the caller's thread or in a separate thread.

Note that if the actions of a listener trigger a synchronous event, then the delivery of the first event will be paused and delivery of the second event will begin. Once delivery of the second event has completed, delivery of the first event will resume. Thus some listeners may observe the second event before they observe the first one.

5.3 Reliability

5.3.1 Dealing with Exceptions

If a listener throws an exception during a callback, it MUST be caught by the Event Admin service and handled in some implementation specific way. If a log service is available the exception SHOULD be logged. Once the exception has been caught and dealt with, event delivery MUST continue with the next listener to be notified, if any.

5.3.2 Dealing with Stalled Listeners

Event handlers SHOULD NOT spend too long in the `handleEvent` method. Doing so will prevent other handlers in the system from being notified. If a handler needs to do something that may take a while, it SHOULD do it in a different thread.

An event admin implementation MAY attempt to detect stalled or deadlocked handlers and deal with them appropriately. Exactly how it deals with this situation is left as an implementation detail. One allowed

implementation is to mark the current event delivery thread as invalid and spawn a new event delivery thread. Event delivery SHOULD resume with the next handler to be notified.

I'm more than a little concerned about the implementation of this feature. It seems the only way to shutdown a misbehaving thread is to invoke `Thread.stop()` on it. But that triggers an asynchronous exception which can cause synchronization monitors to be released prematurely and objects to be left in inconsistent internal states. There seems to be no way to deal with these asynchronous exceptions that is 100% safe. So it seems to me like you're trading one kind of instability for new, exciting and mysterious instabilities further down the line. For that reason I'd like to drop this feature, but I know that some people in CPEG & MEG are very keen on it.

In fact it's worse than that. The `Thread.stop()` method is not available in the CDC/Foundation profile so there is absolutely no way to shutdown a thread that is stalled or deadlocked. This means the only recourse is to start a new event delivery thread and hope that the stalled listener will eventually wake up so that the thread may be shutdown in the normal way. In the meantime, the previous thread still exists, is still consuming memory for its stack and may still be holding Java synchronization locks, greatly increasing the chances that another deadlock will occur.

Implementations MAY choose to blacklist any handlers that they determine are misbehaving. Blacklisted handlers MUST NOT be notified of any events. If a handler is blacklisted, the event admin SHOULD log a message that explains the reason for it.

I think the ability to blacklist handlers is an important feature for secure implementations. Otherwise, a malicious handler could force the event admin to keep spawning new threads until the system ran out of resources. Exactly how the implementation determines that the handler is misbehaving is left as an implementation detail.

5.4 Security

5.4.1 TopicPermission

The `TopicPermission` class allows fine-grained control over which bundles may post events to a given topic and which bundles may receive those events.

The name parameter for the permission is the topic name. `TopicPermission` uses a wildcard matching algorithm similar to BasicPermission, except that slashes are used as separators instead of periods. For example, a name of "a/b/*" implies "a/b/c" but not "x/y/z" or "a/b".

There are two available actions: "publish" and "subscribe". These control a bundle's ability to either publish or receive events, respectively. Neither one implies the other, however an action string of "publish,subscribe" implies both "publish" and "subscribe", as you would expect.

5.4.2 Required Permissions

Bundles that need to register an event listener MUST be granted (ServicePermission "org.osgi.service.event.EventHandler" "register") so that it may register the listener as a service. In addition, listeners require (TopicPermission "<*topic*>" "subscribe") for each topic they want to be notified about.

Bundles that need to publish an event MUST be granted (ServicePermission "org.osgi.service.event.EventAdmin" "get") so that they may retrieve the event admin service and use it. In addition, event sources require (TopicPermission "<*topic*>" "publish") for each topic they want to send events to.

Bundles that need to iterate the listeners registered with the system MUST be granted (ServicePermission "org.osgi.service.event.EventHandler" "get") to retrieve the event listeners from the service registry.

Only the bundle that contains the event admin implementation SHOULD be granted (ServicePermission "org.osgi.service.event.EventAdmin" "register") to register the event admin service.

5.4.3 Security Context During Event Callbacks

During an event notification, the event admin's protection domain will be on the stack above the handler's protection domain. In the case of a synchronous event, the event publisher's protection domain may also be on the stack. Therefore, if a handler needs to perform a secure operation using its own privileges, it MUST invoke the `doPrivileged` method to isolate its security context from that of its caller.

The event delivery mechanism MUST NOT wrap event notifications in a `doPrivileged` call. See section 7.3 for details.

5.5 Interoperability with Native Applications

Implementations of the Event Admin service MAY support passing events to, and/or receiving events from native applications.

*I don't want to require that *all* implementations support native interoperability since that may not be required in some cases.*

If the implementation supports native interoperability, it MUST be able to pass the topic of the event and its properties to/from native code. Implementations MUST be able to support property values of the following types:

- Strings, including full Unicode support
- The eight wrapper types for Java's primitive types (Integer, Long, Byte, Short, Float, Double, Boolean, Character)
- Single-dimension arrays of the above types (including String)
- Single-dimension arrays of Java's eight primitive types (int, long, byte, short, float, double, boolean, char)

Implementations MAY support additional types. Property values of an unsupported type MUST be silently discarded.

5.6 Specific Events

5.6.1 Conventions for All Events

Some listeners will be more interested in the contents of an event rather than what actually happened. For example, a listener might want to be notified whenever an exception occurs anywhere in the system. Both `FrameworkEvents` and `LogEntry` events may contain an exception that would be of interest to our hypothetical listener. If both `FrameworkEvents` and `LogEntrys` use the same property names then the listener can access the exception in exactly the same way. If some future event type follows the same conventions then our listener can receive and process the new event type even though it had no knowledge of it when it was compiled. This suggests the need for some kind of convention regarding which properties will be available, their names and types.

The following properties are suggested as conventions. When new event types are defined they SHOULD use these names with the corresponding types and values where appropriate. These values should be set only if they are non-null.

Name	Type	Notes
bundle	Bundle	A bundle object

All Page Within This Box

bundle.id	Long	A bundle's ID
bundle.symbolicName	String	A bundle's symbolic name
event	Object	The actual event object. Used when rebroadcasting an event that was sent via some other event mechanism.
exception	Throwable	An exception or error
exception.class	String	Must be equal to exception.getClass().getName()
exception.message	String	Must be equal to exception.getMessage()
message	String	A human-readable message
service.objectClass	String[]	A service's objectClass
service	ServiceReference	A service
service.id	Long	A service's ID
service.pid	String	A service's persistent identity
timestamp	Long	The time when the event occurred, as reported by `System.currentTimeMillis()`

5.6.2 Framework Events

In order to present a consistent view of all the events occurring in the system, the existing framework-level events are mapped to MEG's publish-subscribe model (see requirement 4.2.2.1). This allows event subscribers to treat framework events exactly the same as other events. They may not even be aware that these events are being rebroadcast from outside the publish-subscribe event mechanism. However, bundles are still free to register listeners with the framework in the usual way if they so desire.

The topic of an OSGi event is constructed by taking the fully qualified name of the event class, substituting a slash for every period, and appending a slash followed by the name of the constant that defines the event type. For example, the topic of a BundleEvent.STARTED event would be "org/osgi/framework/BundleEvent/STARTED". If the type code for the event is unknown then the event is ignored.

This definition is intended to make it possible to update the framework specification without requiring an immediate change to the MEG specifications. If a new event type is created in a future release of the core framework, then implementers may update the implementation of the Event Admin service without having to wait for the updated MEG specification.

The properties associated with the event depends on its class as outlined below.

5.6.2.1 FrameworkEvent

`FrameworkEvents` are delivered asynchronously.

Property	**Type**	**Notes**
bundle.id	Long	The source bundle's id.
bundle.symbolicName	String	The source bundle's symbolic name. Only set if not null.

All Page Within This Box

bundle	Bundle	The source bundle.
exception.class	String	The fully-qualified class name of the attached exception. Only set if `getThrowable` returns a non-null value.
exception.message	String	The message of the attached exception. Only set if `getThrowable` returns a non-null value and the exception message is not null.
exception	Throwable	The exception returned by `getThrowable`. Only set if it is not null.
event	FrameworkEvent	The original event object.

5.6.2.2 BundleEvent

`BundleEvents` are delivered asynchronously. If listeners require synchronous delivery then they should register a `SynchronousBundleListener` with the framework.

Property	Type	Notes
bundle.id	Long	The source bundle's id.
bundle.symbolicName	String	The source bundle's symbolic name. Only set if not null.
bundle	Bundle	The source bundle.
event	BundleEvent	The original event object.

5.6.2.3 ServiceEvent

`ServiceEvents` are delivered synchronously.

Property	Type	Notes
service	ServiceReference	The result of the getServiceReference method
service.id	Long	The service's ID.
service.pid	String	The service's persistent identity. Only set if not null.
service.objectClass	String[]	The service's object class.
event	ServiceEvent	The original event object.

5.6.3 Log Events

Log events are delivered asynchronously.

The topic of all log events is "org/osgi/service/log/LogEntry".

The properties of a log event are:

Property	Type	Notes
bundle.id	Long	The source bundle's id.

bundle.symbolicName	String	The source bundle's symbolic name. Only set if not null.
bundle	Bundle	The source bundle.
exception.class	String	The fully-qualified class name of the attached exception. Only set if `getException` returns a non-null value.
exception.message	String	The message of the attached exception. Only set if `getException` returns a non-null value and the exception message is not null.
exception	Throwable	The exception returned by `getException`. Only set if it is not null.
log.level	Integer	The log level.
message	String	The log message.
service	ServiceReference	The result of the getServiceReference method (if not null)
service.id	Long	The service's ID. Only set if `getServiceReference` is not null.
service.pid	String	The service's persistent identity. Only set if `getServiceReference` is not null and its service.pid property is not null.
service.objectClass	String[]	The service's object class. Only set if `getServiceReference` is not null.
timestamp	Long	The log entry's timestamp.
log.entry	LogEntry	The `LogEntry` object.

5.6.4 Configuration Change Events

Note: This info is based on RFC 103 [11]Configuration events are delivered asynchronously.

The topic of a configuration event is one of the following:

Topic	**Description**
org/osgi/service/cm/ConfigurationEvent/CM_UPDATED	A Configuration has been updated
org/osgi/service/cm/ConfigurationEvent/CM_DELETED	A Configuration has been deleted

The properties of a configuration event are:

Name	**Type**	**Description**
cm.factoryPid	String	The factory PID of the associated configuration, if the target is a ManagedServiceFactory. Otherwise not set.
cm.pid	String	The PID of the associated configuration.
service	ServiceReference	The service reference of the ConfigurationAdmin service
service.id	Long	The ConfigurationAdmin service's ID.

All Page Within This Box

service.objectClass	String[]	The ConfigurationAdmin service's object class (which must include "org.osgi.service.cm.ConfigurationAdmin")
service.pid	String	The ConfigurationAdmin service's persistent identity

5.6.5 User Admin Events

User admin events are delivered asynchronously.

The topic of a user admin event is one of the following:

Topic	Description
org/osgi/service/useradmin/UserAdminEvent/ROLE_CREATED	A new Role object was created
org/osgi/service/useradmin/UserAdminEvent/ROLE_CHANGED	A Role object was modified
org/osgi/service/useradmin/UserAdminEvent/ROLE_REMOVED	A Role object has been removed

All user admin events have the following properties (These values are set only if they are not null):

Name	Type	Description
event	UserAdminEvent	The event that was broadcast by the UserAdmin service
role	Role	The Role object that was created, modified or removed
role.name	String	The name of the role
role.type	Integer	One of ROLE, USER or GROUP
service	ServiceReference	The service reference of the UserAdmin service
service.id	Long	The UserAdmin service's ID.
service.objectClass	String[]	The UserAdmin service's object class (which must include "org.osgi.service.useradmin.UserAdmin")
service.pid	String	The UserAdmin service's persistent identity

5.6.6 Wire Admin Events

Wire admin events are sent asynchronously.

The topic of a wire admin event is one of the following:

Topic	Description
org/osgi/service/wireadmin/WireAdminEvent/WIRE_CREATED	A new Wire object has been created
org/osgi/service/wireadmin/WireAdminEvent/WIRE_CONNECTED	An existing Wire object has been connected

All Page Within This Box

org/osgi/service/wireadmin/WireAdminEvent/WIRE_UPDATED	An existing Wire object has been updated with new properties
org/osgi/service/wireadmin/WireAdminEvent/WIRE_TRACE	A new value is transferred over the Wire object
org/osgi/service/wireadmin/WireAdminEvent/WIRE_DISCONNECTED	An existing Wire object has become disconnected
org/osgi/service/wireadmin/WireAdminEvent/WIRE_DELETED	An existing Wire has been deleted
org/osgi/service/wireadmin/WireAdminEvent/PRODUCER_EXCEPTION	A Producer service method has thrown an exception
org/osgi/service/wireadmin/WireAdminEvent/CONSUMER_EXCEPTION	A Consumer service method has thrown an exception

The properties of a wire admin event are the following. (Only set if the value is not null)

Name	Type	Description
event	WireAdminEvent	The WireAdminEvent object broadcast by the WireAdmin service
exception	Throwable	The exception returned by `getThrowable` (if not null)
exception.class	String	The fully-qualified class name of the attached exception
exception.message	String	The message of the attached exception
service	ServiceReference	The service reference of the WireAdmin service
service.id	Long	The WireAdmin service's ID.
service.objectClass	String[]	The WireAdmin service's object class (which must include "org.osgi.service.wireadmin.WireAdmin")
service.pid	String	The WireAdmin service's persistent identity
wire	Wire	The Wire object returned by getWire
wire.flavors	String[]	The names of the classes returned by the Wire's getFlavors method
wire.scope	String[]	The scope of the Wire, as returned by its getScope method
wire.connected	Boolean	The result of the Wire's isConnected method
wire.valid	Boolean	The result of the Wire's isValid method

5.6.7 UPnP Events

UPnP events are delivered asynchronously.

All Page Within This Box

The topic of a UPnP event is “org/osgi/service/upnp/UPnPEvent”.

The properties of a UPnP event are the following: (Only set if not null)

Name	Type	Description
upnp.deviceId	String	The ID of the device sending the event
upnp.serviceId	String	The ID of the service sending the events
upnp.events	Dictionary	A Dictionary object containing the new values for the state variables that have changed

5.7 Framework API Changes

The following section covers changes in the framework-level API that are needed to support the event delivery mechanism:

5.7.1 Case-Sensitive Matching in Filters

The event delivery system uses LDAP-style filters just like the framework, but unlike the framework, the names of properties of the event are case-sensitive. Case-sensitive matching is generally more efficient than case-insensitive matching. To avoid having to reimplement the LDAP parsing and filtering logic, the `Filter` interface will be extended with a method to support case-sensitive matching in addition to the case-insensitive matching it already supports. Many framework implementations already include a similar method for internal use.

The following method must be added to the `org.osgi.framework.Filter` interface:

```
public boolean matchCase(java.util.Dictionary dictionary)
```

I'm open to suggestions for a better name.

6 JavaDoc APIs

6.1 Package org.osgi.service.event

The OSGi Event Admin Specification Version 1.0.

See:

Description

Interface Summary	
EventAdmin	The Event Admin service.

All Page Within This Box

EventConstants	Defines standard names for EventHandler properties.
EventHandler	Listener for Events.

Class Summary

Event	An event.
TopicPermission	A bundle's authority to publish or subscribe to event on a topic.

6.2 Package org.osgi.service.event Description

The OSGi Event Admin Specification Version 1.0.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.event; version=1.0
```

6.3 org.osgi.service.event
Class Event

java.lang.Object
  |
  +-**org.osgi.service.event.Event**

public class **Event**
extends java.lang.Object

An event.

Event objects are delivered to EventHandler services which subsrcibe to the topic of the event.

NOTE: Although it is permitted to subclass Event, the operations defined by this class MUST NOT be overridden.

Constructor Summary	
Event(java.lang.String topic, Constructs an event.	java.util.Dictionary properties)

Method Summary	
boolean	**equals**(java.lang.Object object)

All Page Within This Box

	Compares this Event object to another object.
java.lang.Object	**getProperty**(java.lang.String name) Retrieves a property.
java.lang.String[]	**getPropertyNames**() Returns a list of this event's property names.
java.lang.String	**getTopic**() Returns the topic of this event.
int	**hashCode**() Returns a hash code value for the object.
boolean	**matches**(org.osgi.framework.Filter filter) Tests this event's properties against the given filter.
java.lang.String	**toString**() Returns the string representation of this event.

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Constructor Detail

6.3.1 Event

public **Event**(java.lang.String topic,
java.util.Dictionary properties)
Constructs an event.
Parameters:
topic - The topic of the event.
properties - The event's properties (may be null).
Throws:
java.lang.IllegalArgumentException - If topic is not a valid topic name.

Method Detail

6.3.2 getProperty

public final java.lang.Object **getProperty**(java.lang.String name)
Retrieves a property.
Parameters:
name - the name of the property to retrieve
Returns:
The value of the property, or null if not found.

6.3.3 getPropertyNames

public final java.lang.String[] **getPropertyNames**()
Returns a list of this event's property names.
Returns:

All Page Within This Box

A non-empty array with one element per property.

6.3.4 getTopic

public final java.lang.String **getTopic**()

Returns the topic of this event.

Returns:

The topic of this event.

6.3.5 matches

public final boolean **matches**(org.osgi.framework.Filter filter)

Tests this event's properties against the given filter.

Parameters:

filter - The filter to test.

Returns:

true If this event's properties match the filter, false otherwise.

6.3.6 equals

public boolean **equals**(java.lang.Object object)

Compares this Event object to another object.

An event is considered to be **equal to** another event if the topic is equal and the properties are equal.

Overrides:

equals in class java.lang.Object

Parameters:

object - The Event object to be compared.

Returns:

true if object is a Event and is equal to this object; false otherwise.

6.3.7 hashCode

public int **hashCode**()

Returns a hash code value for the object.

Overrides:

hashCode in class java.lang.Object

Returns:

An integer which is a hash code value for this object.

6.3.8 toString

public java.lang.String **toString**()

Returns the string representation of this event.

Overrides:

toString in class java.lang.Object

Returns:

The string representation of this event.

All Page Within This Box

6.4 org.osgi.service.event
Interface EventAdmin

public interface **EventAdmin**

The Event Admin service. Bundles wishing to publish events must obtain the Event Admin service and call one of the event delivery methods.

Method Summary

void	**postEvent**(Event event) Initiate asynchronous delivery of an event.
void	**sendEvent**(Event event) Initiate synchronous delivery of an event.

Method Detail

6.4.1 postEvent

public void **postEvent**(Event event)

Initiate asynchronous delivery of an event. This method returns to the caller before delivery of the event is completed.

Parameters:

event - The event to send to all listeners which subscribe to the topic of the event.

Throws:

java.lang.SecurityException - If the caller does not have TopicPermission[topic,PUBLISH] for the topic specified in the event.

6.4.2 sendEvent

public void **sendEvent**(Event event)

Initiate synchronous delivery of an event. This method does not return to the caller until delivery of the event is completed.

Parameters:

event - The event to send to all listeners which subscribe to the topic of the event.

Throws:

java.lang.SecurityException - If the caller does not have TopicPermission[topic,PUBLISH] for the topic specified in the event.

6.5 org.osgi.service.event
Interface EventConstants

All Page Within This Box

public interface **EventConstants**

Defines standard names for EventHandler properties.

Field Summary

static java.lang.String	**EVENT_FILTER** Service Registration property (named event.filter) specifying a filter to further select Event s of interest to a Event Handler service.
static java.lang.String	**EVENT_TOPIC** Service registration property (named event.topic) specifying the Event topics of interest to a Event Handler service.

Field Detail

6.5.1 EVENT_TOPIC

public static final java.lang.String **EVENT_TOPIC**

Service registration property (named event.topic) specifying the Event topics of interest to a Event Handler service.

Event handlers SHOULD be registered with this property. The value of the property is an array of strings that describe the topics in which the handler is interested. An asterisk ("*") may be used as a trailing wildcard. Event handlers which do not have a value for this propery are treated as though they had specified this property with the value { "*" }. More precisely, the value of each entry in the array must conform to the following grammar:

```
topic-description := "*" | topic ( "/*" )?
topic := token ( "/" token )*
```

See Also:
Event

6.5.2 EVENT_FILTER

public static final java.lang.String **EVENT_FILTER**

Service Registration property (named event.filter) specifying a filter to further select Event s of interest to a Event Handler service.

Event handlers MAY be registered with this property. The value of this property is a string containing an LDAP-style filter specification. Any of the event's properties may be used in the filter expression. Each event handler is notified for any event which belongs to the topics in which the handler has expressed an interest. If the event handler is also registered with this service property, then the properties of the event must also match the filter for the event to be delivered to the event handler.

See Also:
Event, Filter

6.6 org.osgi.service.event
Interface EventHandler

public interface **EventHandler**

Listener for Events.

EventHandler objects are registered with the Framework service registry and are notified with an Event object when an event is broadcast.

EventHandler objects can inspect the received Event object to determine its topic and properties.

EventHandler objects should be registered with a service property EventConstants.EVENT_TOPIC whose value is the list of topics in which the event handler is interesed.

For example:

```
String[] topics = new String[] {"org/osgi/topic", "com/isv/*"};
Hashtable ht = new Hashtable();
ht.put(EVENT_TOPIC, topics);
context.registerService(EventHandler.class.getName(), this, ht);
```

If an EventHandler object is registered without a service property EventConstants.EVENT_TOPIC, then the EventHandler will receive events of all topics.

Security Considerations. Bundles wishing to monitor Event objects will require ServicePermission[EventHandler,REGISTER] to register an EventHandler service. The bundle must also have TopicPermission[topic,SUBSCRIBE] for the topic specified in the event in order to receive the event.

See Also:
Event

Method Summary

void	**handleEvent**(Event event) Called by the EventAdmin service to notify the listener of an event.

Method Detail

6.6.1 handleEvent

public void **handleEvent**(Event event)

Called by the EventAdmin service to notify the listener of an event.

Parameters:
event - The event that occurred.

All Page Within This Box

6.7 org.osgi.service.event
Class TopicPermission

java.lang.Object
 |
 +-java.security.Permission
 |
 +-**org.osgi.service.event.TopicPermission**

All Implemented Interfaces:
java.security.Guard, java.io.Serializable

public final class **TopicPermission**
extends java.security.Permission

A bundle's authority to publish or subscribe to event on a topic.

A topic is a slash-separated string that defines a topic.

For example:

```
org/osgi/service/foo/FooEvent/ACTION
```

TopicPermission has two actions: PUBLISH and SUBSCRIBE.

See Also:
Serialized Form

Field Summary	
static java.lang.String	**PUBLISH** The action string publish.
static java.lang.String	**SUBSCRIBE** The action string subscribe.

Constructor Summary
TopicPermission(java.lang.String name, java.lang.String actions) Defines the authority to publich and/or subscribe to a topic within the EventAdmin service.

Method Summary	
boolean	**equals**(java.lang.Object obj) Determines the equality of two TopicPermission objects.
java.lang.String	**getActions**() Returns the canonical string representation of the TopicPermission actions.
int	**hashCode**()

All Page Within This Box

	Returns the hash code value for this object.
boolean	**implies**(java.security.Permission p) Determines if the specified permission is implied by this object.
java.security.PermissionCollection	**newPermissionCollection**() Returns a new PermissionCollection object suitable for storing TopicPermission objects.

Methods inherited from class java.security.Permission
checkGuard, getName, toString

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

6.7.1 PUBLISH

public static final java.lang.String **PUBLISH**
The action string publish.

6.7.2 SUBSCRIBE

public static final java.lang.String **SUBSCRIBE**
The action string subscribe.

Constructor Detail

6.7.3 TopicPermission

public **TopicPermission**(java.lang.String name,
java.lang.String actions)
Defines the authority to publich and/or subscribe to a topic within the EventAdmin service.

The name is specified as a dot-separated string. Wildcards may be used. For example:

```
org/osgi/service/fooFooEvent/ACTION
com/isv/*
*
```

A bundle that needs to publish events on a topic must have the appropriate TopicPermission for that topic; similarly, a bundle that needs to subscribe to events on a topic must have the appropriate TopicPermssion for that topic.

Parameters:
name - Topic name.

All Page Within This Box

actions - PUBLISH,SUBSCRIBE (canonical order).

Method Detail

6.7.4 implies

public boolean **implies**(java.security.Permission p)

Determines if the specified permission is implied by this object.

This method checks that the topic name of the target is implied by the topic name of this object. The list of TopicPermission actions must either match or allow for the list of the target object to imply the target TopicPermission action.

```
x/y/*,"publish" -> x/y/z,"publish" is true
*,"subscribe" -> x/y,"subscribe"   is true
*,"publish" -> x/y,"subscribe"     is false
x/y,"publish" -> x/y/z,"publish"   is false
```

Overrides:
implies in class java.security.Permission
Parameters:
p - The target permission to interrogate.
Returns:
true if the specified TopicPermission action is implied by this object; false otherwise.

6.7.5 getActions

public java.lang.String **getActions**()

Returns the canonical string representation of the TopicPermission actions.

Always returns present TopicPermission actions in the following order: PUBLISH,SUBSCRIBE.

Overrides:
getActions in class java.security.Permission
Returns:
Canonical string representation of the TopicPermission actions.

6.7.6 newPermissionCollection

public java.security.PermissionCollection **newPermissionCollection**()

Returns a new PermissionCollection object suitable for storing TopicPermission objects.
Overrides:
newPermissionCollection in class java.security.Permission
Returns:
A new PermissionCollection object.

6.7.7 equals

public boolean **equals**(java.lang.Object obj)

Determines the equality of two TopicPermission objects. This method checks that specified Topic has the same Topic name and TopicPermission actions as this TopicPermission object.
Overrides:

All Page Within This Box

equals in class java.security.Permission
Parameters:
obj - The object to test for equality with this TopicPermission object.
Returns:
true if obj is a TopicPermission, and has the same Topic name and actions as this TopicPermission object; false otherwise.

6.7.8 hashCode

public int **hashCode**()
Returns the hash code value for this object.
Overrides:
hashCode in class java.security.Permission
Returns:
A hash code value for this object.

7 Considered Alternatives

7.1 Define an "Event Source"

It's not clear what entity should be defined as the source of an event that is published using this mechanism. Is it the bundle that obtained the `EventAdmin` service? Such a scheme would be easy to enforce using `ServiceFactorys` but would limit each bundle to a single event source. The event source could be a service, but not all designs will include a service that could serve as the event source. For example, which service is the source of a "SIM card inserted" event? The event source could be any old object, but even then there might not be an appropriate object available. For example, in OSGi R1, the source of the `FrameworkEvent.STARTED` event was undefined because the concept of a system bundle had not been introduced. It seems it is not always guaranteed that there will be an identifiable source for the event so trying to define one for the general case is a lost cause.

Although the event source is undefined in the general case, there are several cases where the event source *is* well defined. For examples one needs only to look to the framework-level events that are rebroadcast by the `EventAdmin`. In such cases, the event should carry a property (or properties) that identifies the source of the event. Such a property, however, would be specific to each event type.

7.2 Allow Listeners to Throw Exceptions and/or Return Status

Some event mechanisms allow the listeners to return a status flag or some other data to the event source. For example, the AWT `InputEvent` includes a "consumed" flag which listeners may set to indicate that the event source should not continue with default event processing. Other designs allow listeners to raise an exception. For example, `VetoableChangeListeners` may throw a `PropertyVetoException` to indicate that the new

property value is unacceptable. While such designs are interesting, they are considered to be fringe cases. Most event publishers will not need any status information from the listeners and will not want to deal with any exceptions they might throw.

If necessary, the Event class may be subclassed to add a simple facility for returning data to the event source. Such a facility would only be useful with synchronously delivered events. (Otherwise some kind of callback mechanism would be required, which is out of scope for this RFC.) Listeners could then set a variable on the event object itself which the event source can interrogate after event delivery is finished.

Listeners cannot use exceptions to signal the event source. All exceptions thrown by a listener are caught by the event delivery sub-system and treated as errors. However, most exceptions can probably be turned into simple data and handled by subclassing as above.

7.3 Privileged Callbacks

The first draft of this RFC included a requirement for all event notifications to be wrapped in a `doPrivileged` block by the event delivery system. Listeners would always execute with their full privileges which removes the burden to consider security issues from the application programmer. Unfortunately, the event delivery system itself would need to be granted `AllPermission` so that its protection domain does not constrain listeners' privileges.

This feature was dropped after discussions at the MEG face-to-face meeting in Nice [10]. The problem is that it does not completely prevent application programmers from having to deal with security issues. There are other callbacks besides events for which a `doPrivileged` call would be unavoidable. We could attempt to track them down and isolate them with `doPrivileged` calls too. However, this seems to start a slippery slope that ends with all infrastructure bundles (that is, all non-application bundles) needing to have `AllPermission`. This is clearly an undesirable situation because it defeats the point of having fine-grained security and renders much of the MEG policy mechanisms moot.

The conclusion therefore is that `doPrivileged` calls are simply a fact of life in OSGi and cannot be avoided, therefore we should not try to eliminate them.

8 Security Considerations

8.1 Threat Assessment

The following sections examine the sorts of attacks that malicious bundles might try to launch against the system.

8.1.1 Hijacking the Event Delivery Thread

If a listener never returns from the callback then subsequent listeners may never be notified of the event. The thread would stall and become essentially deadlocked. Malicious bundles might use this fact to launch a sort of denial of service attack against the event delivery system. A successful attack could leave the system in an unusable state if vital threads are deadlocked waiting for a listener that will never return.

Implementations MAY attempt to detect this situation and deal with it as described in section 5.3.2, however the best defense is to avoid granting permission to receive events to a bundle you do not trust.

8.1.2 Flooding the Event Queue

If an event source posts too many events in a short amount of time then the event delivery thread may become overloaded. The system could become bogged down doing nothing but delivering events. If the event queue grows faster than the event delivery thread can dispatch the events, it's possible that the system would run out of memory.

To combat this problem, implementations MAY limit the rate at which an event source may post events.

Exactly how throttling is accomplished is left as an implementation detail. One scheme I can think of is to insert a steadily increasing delay using `Thread.sleep()` each time an event is posted. The delay would reset to zero if no events are fired during some timeout period. I'm sure there are other schemes that would work.

As always, the surest defense is to avoid granting malicious bundles permission to access the event admin in the first place.

9 Document Support

9.1 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]. Peter Kriens, BJ Hargrave. Listeners Considered Harmful
http://membercvs.osgi.org/docs/whitepaper/whiteboard.pdf

[4]. RFP 52 – MEG Application Model
http://membercvs.osgi.org/rfps/rfp-0052_MEG_WS_Application_Model.doc

[5]. RFP 59 – Simplifying Event Delivery
http://membercvs.osgi.org/rfps/rfp-0059-Simplifying_Event_Delivery.doc

[6]. RFC 85 – Device Management
http://membercvs.osgi.org/rfcs/rfc0085/

[7]. RFC 91 – MEG Application Model
http://membercvs.osgi.org/rfcs/rfc0091/

[8]. Minutes of the Burlington meeting, August 24-25, 2004
http://membercvs.osgi.org/eg/meg/minutes/2004-08-25-Boston.txt

[9]. The OSGi Service Platform Release 3

http://membercvs.osgi.org/sp-r3/sp-r3/project/sp-r3/released.html

[10]. Minutes of the Nice meeting, October 20-22, 2004
http://membercvs.osgi.org/eg/meg/minutes/2004-10-20-Nice.txt

[11]. RFC 103 – Configuration Listener
http://membercvs.osgi.org/rfcs/rfc0103/

[12]. Minutes of the Boca Raton meeting, December 6, 2004
http://membercvs.osgi.org/eg/cpeg/minutes/2004-12-06_CPEG_BocaRaton.txt

[13]. Minutes of the Newark meeting, January 25, 2005
http://membercvs.osgi.org/eg/cpeg/minutes/2005-01-25_CPEG_Newark.txt

9.2 Author's Address

Name	Kevin Riff
Company	Espial Group, Inc.
Address	Suite 901 200 Elgin St., Ottawa ON, Canada, K2P 1L5
Voice	(613) 230-4770 x1136
e-mail	kriff@espial.com

9.3 Acronyms and Abbreviations

9.4 End of Document

All Page Within This Box