RFC 48 - State Management

Confidential, Draft

8 Pages

Abstract

RFC 48 provides a comprehensive model of how to obtain values from items like sensors or fixed values and to set values to actuators. It addresses the issues raised in supporting a vehicle that can have hundreds of sensors and actuators and static information. This RFC proposes a mechanism for using Wire Admin to connect to a provider with multiple values

0 Document Information

0.1 Table of Contents

0.2 Status

This document specifies RFC 48 for the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

0.3 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.
```

0.4 Revision History

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

Revision	Date	Comments
Initial	APR 05 2002	Peter Kriens, Discussion document for Virginia
Rewrite	AUG 12 2002	Peter Kriens, after many discussions with the VEG

All Page Within This Box

Leuven comments	AUG 26 2002	Peter Kriens
Rewrite	SEP 02 2002	Rick DeNatale, after conference call on 28 AUG
Update	SEP 03 2002	Addressed some comments from Sofia Doncheva on the mail reflector today.
Update	SEP 04 2002	Updated after conference call with cpeg/veg
Update	SEP 06 2002	Updated after comments from Peter on the mail reflector and conference call

1 Introduction

This RFC addresses problems where a large number of items are present that can provide and/or accept state information. In this RFC, these items are called *status items*. Status items represent a sensor/actuator, for example, a fuel sensor in a car can provide a quantity of fuel left. Other information items can be a Vehicle Identification Number (VIN) or the number of seats. Information items can be dynamic, i.e. the value can change over time. For example, the speed of the car will change continuously.

Actual environments can have hundreds, sometimes thousands, of these information items. This RFC defines a number of basic requirements for handling these information items in an OSGi environment.

This RFC was mainly driven by the Vehicle Expert Group but was deemed more general than vehicles only.

2 Motivation and Rationale

This RFC is at the core of the OSGi applications: connecting to the real world. It is therefore clear that such a service is necessary.

The service must fulfill the following requirements:

All Page Within This Box

1. It must be able to handle a large number of information items. This means it should easily handle 1000 information items and it should be workable up to 10.000.

2. Information items can be a combination of readable, writable, and mandatory. Note that some actuators are actual write only, no existing value can be queried from for example a mirror tilt.

3. Changes to the state of information items must be delivered to interested parties.

4. The rate of update must be controllable.

5. The proposal should include a general mechanism for indicating the location of a particular information item. In practice, the naming of such items is complex and particular to the situation. Details of the name space for addressing items is left to the provider.

6. Many information items will be rarely used. It should be possible to optimize applications that they minimize resource consumption based on this assumption.

7. The service should leverage available services in the OSGi environment

8. It should be easy for the programmer to use

9. It should provide secure access to information items.

3 Technical Discussion

State Management is proposed as an application of the Wire Admin specification. It uses Wire Admin to connect Producers to Consumers and vice versa. However, State Management introduces the concept of a Composite Target. Such a target is a Consumer or Producer service that does not represent a single value, but can represent a (potentially very) large set of information items. For example a Composite Target can abstract all or a subset of the sensors/actuators connected to a single bus. The Composite Target concept is thus an optimization that was introduced because a single producer/consumer per information entity turned out to be a performance problem (after prototyping).

3.1 Exchange of Information

The main problem addressed by this RFC is how to represent and identity values passed over a wire, when multiple types can be passed. The proposed solution is to define an interface which describes an “envelope” for passing a value with enough additional information to provide for the identification of individual status items, and for security policies based on the notion of a security scope.

The Envelope interface defines a type to be used for passing and identifying values between composite targets. An Envelope carries three components, the actual value of the status item, an identification of which value is actually represented, and a security scope.

All Page Within This Box

3.1.1 Values carried in Envelopes

The actual type of object used to represent a status value is up to the design of the Producer. In many cases existing Java or OSGi objects are natural representations of status values conveyed in Envelopes, for example:

Switches (such as are found in MOST or CEBus) are naturally represented by Booleans

Textual information is naturally represented by java Strings.

Dimensioned Numbers (as found in MOST or CEBus) are naturally represented by OSGi Measurements

3.1.2 Status item identification

At this time, it is not felt that a generally agreeable model exists for how to map the varieties of status item identification. The RFC therefore simply provides for an identification object to be attached to a Envelope. The actual mapping of this object to a status namespace is left up to the designer of a particular compound producer. Although in many cases the identification might be a simple string, we have left the identification as an Object to allow for cases when another approach such as a property list might be required.

3.1.3 Scope

In order to provide a mechanism, which allows implementing security policies in complex targets, Envelopes carry a security scope. This is discussed below in the section on changes to the WireAdmin specification.

3.1.4 Envelope and BasicEnvelope

The RFC proposes that Envelope be an Interface to allow flexibility in implementation. It also proposes a class called BasicEnvelope, which implements the EnvelopeInterface in a straightforward way.

3.2 Scope

Composite Targets could be implemented without a change to Wire Admin because that specification is largely agnostic of the data types that are interchanged between consumers and producers. Producers and consumers communicate with a mutually understood class (a flavor). However, the requirement to support fine grained access control makes it necessary to introduce some changes in the Wire Admin specifications.

Normally, the security of Wire Admin is based on the fact that only a trusted application could make wires between producers and consumers. Neither producers nor consumers could get, or see, the other party. The Wire formed a barrier between the communicating parties. Applying this concept to Composite Targets implies that a target has/permits access to all its contained information items. This security was considered too broad and it was voiced that a more fine-grained security was needed. This made it necessary to introduce the concept of a *scope* into Wire Admin. This needed to be placed in the Wire Admin service because that is where the security issues are handled.

Consumers and Producers may set a list (String[]) of scope names as a registration property (WIREADMIN_PRODUCER_SCOPE and WIREADMIN_CONSUMER_SCOPE). The Wire object must check of these scope names with the permissions. If there exists a WirePermission object that implies a scope name, then that name is permitted.

If a Producer or Consumer does not set it's corresponding scope registration property, Wire Admin will treat it as if it had specified WIREADMIN_[CONSUMER|PRODUCER]_SCOPE = "*". In the case of the consumer, this is interpreted to mean that a consumer will accept any scopes produced by the Producer. In the case of the

producer this means that the entire list of scopes requested by the consumer will be associated with the wire. Note however that this does not mean that they will actually be produced by the producerthe producer will actually produce them.

The scope of a Wire is defined as the intersection of four sets:

1) The set of requested scope names in the consumer's WIREADMIN_CONSUMER_SCOPE registration property.

2) The set of published scope names in the producer's WIREADMIN_PRODUCER_SCOPE registration property

3) The set of scopes (c) in the first list for which the consumer has WirePermission[CONSUME, c]

4) The set of scopes (p) in the second list for which the producer has WirePermission[PRODUCE, p]

For example, the Producer registers {"DoorLock", "DoorOpen", "Manufacturer"}. It has WirePermission[PRODUCE ,'*'] (all). The Consumer service registers { "DoorLock", "DoorOpen", "IgnitionLock" } and has WirePermission[CONSUME,DoorLock] and WirePermission[CONSUME,DoorOpen]. The resulting scope is then: { "DoorLock", "DoorOpen" }. This list can be obtained from the Wire object with the getScope() method. The hasScope(String) method can be used to find out if a name is in scope.

Note that the definition of the wire scope is declarative. An implementation of WireAdmin may use any ordering of set operations which meets the definition that it deems to be most appropriate.

3.2.1 Scopes and Permission Checking

This RFC extends the definition of Wire object to require it to check the scope for wires carrying Envelopes. This means that consumers will never receive Envelopes with scopes for which they do not have the appropriate permissions, and Envelopes with scopes for which a producer does not have the proper permission will be silently discarded by the wire.

If a producer passes a Envelope to a wire via update() which doesn't include the scope because the Consumer didn't specify it or because the producer doesn't have permission to produce it, the WireObject will ignore it and not pass it to the consumer. This also implies that the wire will remove elements whose scope is not in the wire scope from the array returned by a complex producer's polled() method (see 3.4 Use of Poll on page 7)

If the scope of an Envelope is not set (default is null), the Wire object must do no permission checking. This implies that a producer that supports restrictive access must verify that the scope is set before it hands out object or deny access.

3.2.2 Scope Syntax and Semantics

This RFC does not define the meaning of the scope. The Wire Admin service must not assume any meaning to the scope name. The meaning is bilaterally defined by the Consumer and Producer service. The scope is a mechanism provided by the Wire Admin service that provides a tool to consumer and producers to implement fine-grained access control. The only reserved scope name is "*" which is used to represent a universal set.

A scope name must be usable with the matching rules as defined in BasicPermission for names. A scope name should follow the rules for fully qualified Java class names. For example, most.doorlock would be a correct scope name while most/doorlock would not.

Scope names used in the properties WIREADMIN_CONSUMER_SCOPE, and WIREADMIN_PRODUCER_SCOPE must be either fully qualified, or be the universal wild card "*". Partially

All Page Within This Box

qualified scope names such as “x.*” may only be used for configuring the permissions granted to the consumer and producer using the Permission Admin service.

3.3 Wire Admin Filtering

The Wire Admin specification supports the filtering of updates depending on update frequency, but also based on value or value change. Filtering values within Envelopes on value changes is complicated because it would require the wire to keep a history of each individual status item.

Filtering of composite values is therefore not supported.

3.4 Use of Poll

A Composite producer can be polled for its value. However, a Composite producer does not represent a single value, but can instead represent many different information items. This is not a problem for update because the producer can call update several times when multiple information items have changed. It must therefore be defined what a type is returned when poll(ed) is called.

The polled() method on a complex producer should return an array of Envelopes of all of the status items “owned” by the producer. The wire will filter this array removing any Envelopes which are not in the wire’s scope and return this to the consumer as the result of the poll() method.

4 Summary of Proposed Changes

4.1 org.osgi.service.wireadmin

- WirePermission – A permission class that can scope name. Standard name comparisons with wildcards must be supported.
- Wire – Introduce a hasScope and getScope method. Require checking of scopes when passing Envelopes
- Constants – Introduce WIREADMIN_CONSUMER_SCOPE, WIREADMIN_PRODUCER_SCOPE, and WIREADMIN_SCOPE_ALL configuration properties.

All Page Within This Box

5 Security Considerations

The security of state management consists of two levels. First, Consumer and Producer services need to have ServicePermission[REGISTER] and very few applications should have ServicePermission[GET]. This means that only trusted applications can connect a Producer service to a Consumer service.

The second level is the scope. If a Producer or Consumer service is Composite, and has sensitive information it should check the scope. The scope must be verified by the Wire using the WirePermission of the appropriate object.

6 Document Support

6.1 References

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

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	Finnasandsvägen 22
Voice	+46 300 39800
e-mail	Peter.Kriens@aQute.se
Name	Rick DeNatale
Company	IBM/OTI
Address	601-109 Hutton Street, Raleigh, NC 27609, USA
Voice	+1 919 821-3220x12

All Page Within This Box

e-mail	Rick_DeNatale@oti.com

6.3 Acronyms and Abbreviations

6.4 End of Document