RFC 7 - Configuration Management

Members Only, **RC 4**
cpeg-rfc_7_cm-1_00K

41 Pages

Abstract

Configuration Management (CM) comprises all the activities needed to configure bundles when they are deployed. This document proposes a comprehensive API for this purpose. The API is defined, explained and related to other approaches.

1 Contents

2 Document Information

2.1 Status

This document proposes a comprehensive strategy for OSGi configuration management. Its current status is the release candidate for voting by the Common Platform Expert Group. Distribution is OSGi members limited.

2.2 Acknowledgement

This document was reviewed, discussed and contributed by Jan Luehe (SUN), Maria Ivanova (ProSyst), Petri Niska (Nokia) and K. Hellden (Gatespace).

2.3 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 [9].

`Source code is shown in this typeface.`

2.4 Revision History

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

Revision	Date	Comments
First draft.	26 October 2000	First draft. Peter Kriens
B draft	28 November 2000	Minor changes, added dynamic data typing and CMPlugin
C draft	5 December 2000	Minor changes and added a chapter for management issues, P. Kriens
D draft	13 December 2000	Major rewrite to clarify, removed Selection object
E draft	10 January 2001	Editorial changes P. Kriens, mainly from David Bowen's comments. Added Javadoc. Added a provisionary chapter about meta typing
F draft	24 January 2001	Added David as author, defined schemes for PIDs and changed rankning name
G draft	29 January 2001	Review by CM subgroup
H draft	20 February 2001	Adding more comments from subgroup
J draft	2 march 2001	Comments resolutions from CPEG meeting Dallas

K draft	12 March 2001	Renamed interfaces: ManagedServiceConfiguration -> Configuration, CM -> ConfigurationAdmin, CMPlugin -> ConfigurationPlugin. Renamed methods: updateManagedService->updated, updateManagedServiceFactory -> updated, deleteManagedService -> deleted. Added one-arg version of createFactoryConfiguration. Changed prohibition on plugins with service.cmRanking <0 or >1000 modifying properties to be a recommendation only. Removed requirement that plugins implement MetaTypeProvider if they want to change properties. Made all methods on Configuration throw an IllegalStateException if called after the configuration has been deleted. Extensive editing to make the specification more clear.
L draft	12 April 2001	Comments San Francisco: cm.target, no management information

3 Introduction

Configuration Management (CM) is probably one of the most important aspects of deployment. CM resides between the bundle developers and the end users. The CM adapts the flexibility that a software developer designs into his bundle down to concrete choices. For example, a software developer can write its software so it can use any TCP/ IP port for a specific service, however, the CM must decide which port will actually be used on a specific gateway.

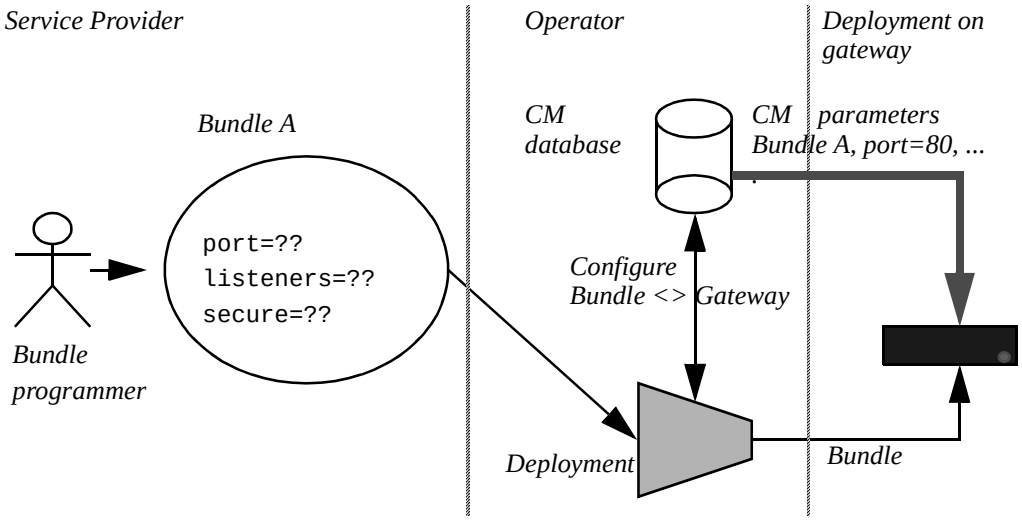

Fig. 1 Overview of CM principles

This document is about the API needed to perform the process represented by the thick arrow in Overview of CM principles. An API that will allow a bundle programmer to use parameters as specified by the operator and/or user of the residential gateway.

3.1 Requirements on CM

1 A CM API will have two clients: The bundle programmer receiving configuration parameters and the management bundle that will require a mechanism to set the configuration parameters. As always the API must not prescribe a certain implementation, neither for the management system, nor for the bundle.

2 Trade-offs in simplicity should be made at the expense of the CM implementation bundle and in favor of the bundle. The rationale being that there will be many normal bundles and few CM bundles. Also, CM bundles will probably by their nature be written with more care than normal bundles.

3 The API must support both bundles that have their own user interface to change their configurations and bundles that rely on a central management facility to provide the parameters.

4 The management bundle must be able to deduce the names and the types of the needed configuration parameters.

5 The API should not limit the bundle programmers to a single set of parameters. Many bundles consist of services that can be instantiated many times. For example, a bundle implementing a service listening to an ICQ server would probably require several instances for different members of the family. Each of these listeners will require a separate set of parameters related to the family member. This pattern is so common, that it must be supported by the API. This to prevent each bundle from having to translate a single configuration parameter set to a number of instances.

6 The API must be deployed on a wide range of gateways. This means that the API should not assume file storage on the gateway. The choice to use file storage should be left to the implementation of the management bundle.

7 The API should fully allow for remotely managed gateways and not assume that configuration is stored locally in any way. Nor should it assume it is always done remotely, both implementation approaches should be viable.

8 The gateway is a dynamic environment that must run 24/7/365. Configuration management updates must happen dynamically and should not require reboots or restarts of bundles. Changes in the configuration should immediately be reflected.

9 Will not require more than the minimal profile as requested by RFP 0006 Request For Proposal for Execution Profile.

10 The API should not assume “always-on” connectivity so that the API is also applicable for mobile applications in cars, phones or boats.

11 The CM will play a crucial role in the OSGi framework implementations. Its importance is probably only surpassed by the framework itself. This position makes it necessary for bundles to be able to extend the functionality of any CM in a defined way. The CM proposal must allow for other bundles to influence the properties that are being set on a service. This influence should at least encompass: initiating an update, removing certain properties, adding properties and modifying the value of properties, potentially based on existing property or service values.

4 Motivation and rationale

A core consideration in the proposal is that it would be false to assume a single set of configuration parameters for a bundle would be sufficient. Bundles often contain multiple distinct functions, each requiring its own configuration parameters.

There are actually two distinct cases. The first case is when some bundle can automatically detect or knows that a service is needed. The second case, which is very common, is when a bundle can provide a function but does not know how many instances of this function are needed a priori.

A few use cases in the following sections.

4.1 Automatic Detection

When a device in the external world needs to be represented in the OSGi framework it needs to be detected in some way. These are some use cases with automatic detection of devices or services. The principal aspect of this is that the CM cannot control the number of instances, this is controlled by the outside world. However, when a device is detected it needs configuration information to play a useful role in the framework.

4.1.1 Networks with detection

A power line network automatically detects devices on its network. Once it detects for example a switch, it will register a Switch service in the OSGi service registry. It is highly likely that this Switch needs configuration information specific for that instance. For example which lamps should be turned on for that switch, or what timer should be started, the zone in which it resides, etc. It should be obvious that one bundle could potentially have hundreds of these switches and lamps, and each needs its own configuration set.

4.1.2 JINI, UPnP

Services discovered on the external networks can be represented in the OSGi registry. For example, a printer could be detected via UPnP. Once in the registry, these services usually require local configuration information. E.g. a printer service needs to be configured for its local role and maybe its location. This information needs to be available in the OSGi registry whenever that specific printer service is registered in the registry. This means that the CM must remember that configuration information and must have a way to connect CM information to this service.

4.2 Singletons

When an object can only be instantiated once, it is a singleton. A singleton only requires a single set of properties for configuration. However, it should be noted that one bundle could implement several different singleton *types.* This implies that the bundle would require multiple configuration parameter sets. Each of these functions will require its own properties.

4.2.1 Watchdog

A watchdog function would have to watch the registry for the status and presence of services in the registry. There is no need for multiple instances of a watchdog, so the watchdog only needs a single set of properties that contain the polling time and list of devices to watch.

4.3 Instantiating multiple services

In other cases, the number of instances is under control of the CM; this may be the owner or alternatively the operator. In this case a bundle has no a-priori knowledge of how many instances are needed and needs to be informed about this count. In this model, the management system can create and delete objects in the OSGi environment where the bundles can deliver the classes.

4.3.1 Mail fetcher

Assume a bundle implements a mail fetcher which shows the number of mails that a user has and shows this on a display. This function is probably required for each member of the household. This function could be viewed as a Class that needs to be instantiated for each family member. Each instance will require different parameters (password, host, protocol, user id etc.).

4.3.2 Conversion service

Assume a bundle has the code to implement a conversion service that receives a temperature and depending on some settings can turn on and off a switch. This service would need to be instantiated many times depending on the configuration, and each instance would require its own parameters for upper value, lower value and the identification for the switch.

4.3.3 Serial port definitions

Serial ports can often not be used by Device Access because they cannot detect the device type that is connected, nor is it always possible to detect the serial ports beforehand. Each serial port would require a description of the device that is connected. The bundle managing the serial ports would need to instantiate a number of serial ports under CM control with the appropriate device_id properties so it can participate in Device Access.

4.3.4 X10 like networks

The X10 home automation standard does not have a standard for detecting which devices are present and what their type is. A bundle managing an X10 network needs to know a priori what devices it can expect on its local network. This requires parameters like address, type and connectivity information on a per device basis in the configuration.

5 Technical Discussion

5.1 Overview

This proposal is based around the concept of a ConfigurationAdmin service that manages the configuration of an OSGi box. This service maintains a database of configuration objects, either local or remotely. A configuration object contains a set of properties to configure an aspect of the environment.

A bundle can retrieve these configuration objects by registering a service with a persistent identity (PID) property. During registration, the ConfigurationAdmin will detect these services and hand over the configuration information via a call-back. When later that configuration information is modified, the modified properties are handed over to this service again with the same call-back.

This proposal defines two types of service that can receive configuration information:

A *ManagedService* will receive a single set of properties when it is registered, or when a configuration object is created for it in the ConfigurationAdmin, whichever happens later. It is also informed when its configuration object is changed in the ConfigurationAdmin.

A *ManagedServiceFactory* will receive from 0 to n configuration objects when it registers, depending on the current configuration. The factory is informed of configuration changes and also of the creation and deletion of its configuration objects in the ConfigurationAdmin.

The configuration object is defined in a *Configuration* interface. It is primarily a set of properties that can be updated by a management system, user interfaces on the box, or other applications. Changes in configurations are first made persistent and then immediately passed to the service via a call-back.

The ConfigurationAdmin allows third party applications to participate in the configuration process: bundles that register a *ConfigurationPlugin* interface can process the properties before they reach the target service.

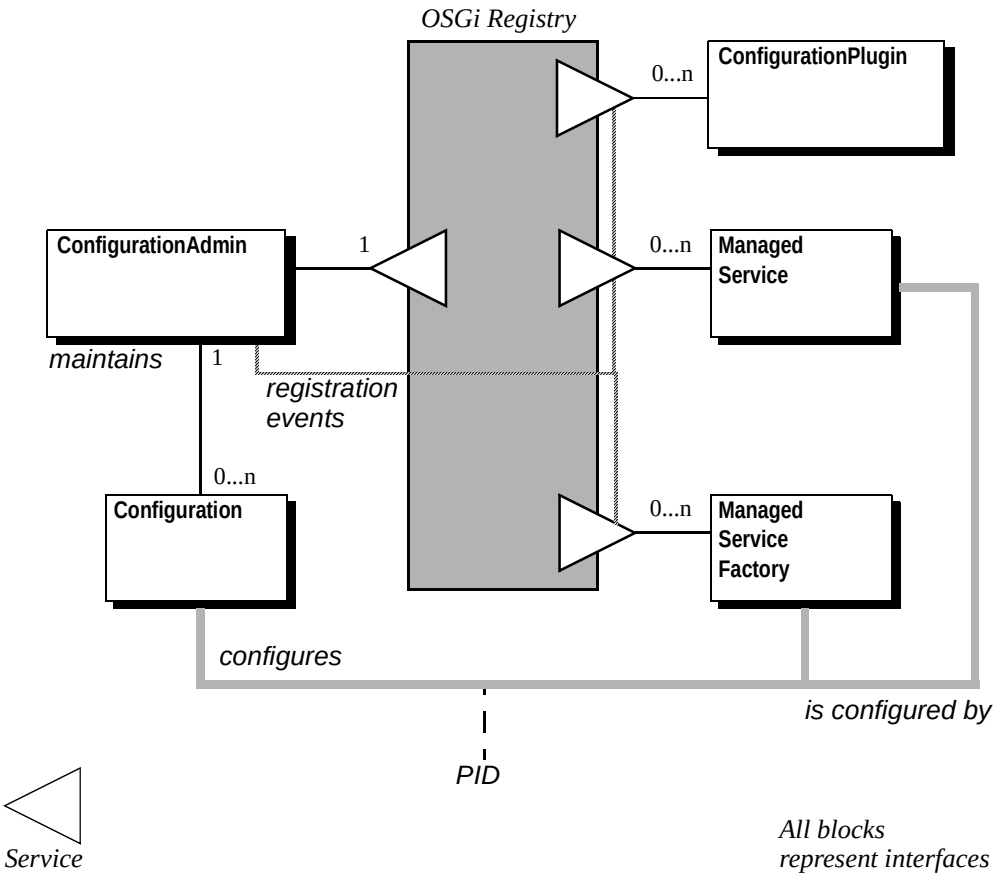

Fig. 2 Overview ConfigurationAdmin interfaces

5.2 The Persistent Identity, PID

A PID is a unique identity for a service that persists over multiple invocations of the framework. When a bundle registers a service with a PID, i.e. a service registration property named `service.pid`, it should always specify

the same PID for that service. If the bundle is stopped and later started, the same PIDs should be used for the services that are registered.

PIDS can be useful for all services, but the ConfigurationAdmin requires their use with *ManagedService* and *ManagedServiceFactory* registrations, because it associates configuration information with PIDs.

A bundle must not register multiple services with the same PID. If this happens anyway, the ConfigurationAdmin must send the appropriate configuration objects to all services registered under that PID from that bundle and should report an error in the log. Duplicate PIDs from other bundles must be ignored and should be reported in the log.

5.2.1 Human readability

PIDs are not intended to be used by humans but sometimes the user is confronted with one. For example when the user has installed an alarm system it will be necessary to identify the different components to some configuration program. To improve this process it is advised to follow the schemes for PIDa as defined in this document and also add a “description” property which describes the object in more detail.

Any globally unique string can be used as a PID. However, the following sections define schemes for common cases. These schemes are not mandatory, but bundles are urged to use these schemes to increase consistency.

5.2.2 Local Bundle PIDs

As a convention, names starting with the bundle id and a dot are reserved for a bundle. E.g. the PID “ 16.345” belongs to bundle 16.

5.2.3 Software PIDs

Bundles that are singletons can use a java package name that they own as pid. I.e. this is the reverse domain name scheme. E.g. the PID “se.ericsson.watchdog”.

5.2.4 Devices

Devices are usually organized on busses. If the bus does not provide serial number information, the address may be used. The format of the serial number should be the same as it would be printed on the housing or box to ease recognition by humans.

Bus	Description	Format	Example
USB	Universal Serial Bus. Use the standard device descriptor	idVendor (hex 4) idProduct (hex 4) iSerialNumber (decimal)	USB-0123-0002-9909873
IP	Internet Protocol	ip nr (dotted decimal)	IP-172.16.28.21
802	IEEE 802 MAC address (token ring, ethernet, ...)	MAC address with : separation	802-00:60:97:00:9A:56
ONE	1-wire bus of Dallas Semiconductor	family (hex 2) and serial nr including CRC (hex 6)	ONE-06-00000021E461

Bus	Description	Format	Example
COM	Serial ports	serial nr or type name of device	COM-krups-coffeebrewer-12323134

5.3 Configuring singletons and detected devices

A bundle that needs configuration information should register one or more ManagedService objects with a PID property. If it has a default set of properties for its configuration, it may include them as registration properties of the ManagedService so as to act as a template when a configuration is created for the first time.

The ConfigurationAdmin, when detecting a ManagedService, must retrieve the configuration stored for the registered PID. This configuration object, if there is one, is then sent to the ManagedService with the updated method. If the ConfigurationAdmin starts up after a ManagedService is registered it must call updated on this service as soon as possible.

If a ManagedService is registered and no configuration data is available, the ConfigurationAdmin must call the ManagedService.updated method with a null parameter. If the ConfigurationAdmin is registered later it must call this in the same way for all ManagedServices in the registry. This implies that a ManagedService must always get a call-back when it registers and the ConfigurationAdmin exists or is started.

The call-back from ConfigurationAdmin to ManagedService must take place on another thread than the thread that executed the ManagedService registration. This is to allow the ManagedService to finish its initialisation in a synchronized method. Care should be take not to cause deadlocks by calling the framework in a synchronized method.

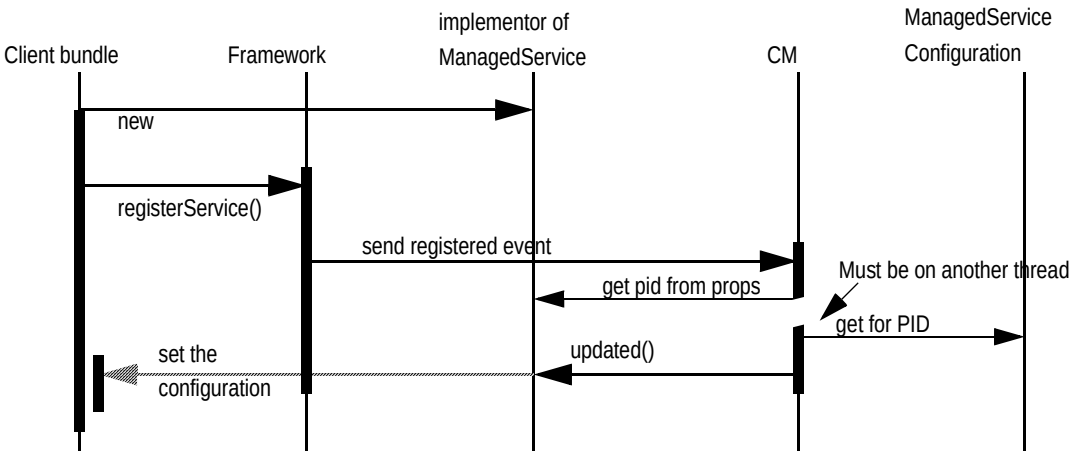

Fig. 3 Action diagram for setting the Configuration on a ManagedService

5.3.1 Race conditions

There will be a short period, when a ManagedService is registered, where the default properties are visible in the registry before they are replaced by the properties of the configuration. Care should be taken that this does not cause race conditions. In certain cases the ManagedService object must be split from the object performing the

actual service to prevent these race conditions. In that case the ManagedService is first registered, and after the configuration is received, the actual object is registered. In such cases, the use of a factory that does this more naturally, should be considered.

5.3.2 Example ManagedService

See “PIDs and external associations” on page 11. It shows an instance diagram of a live configuration example. There are three services registered under ManagedService. Each has its own PID. The ConfigurationAdmin has a database containing a configuration record for each PID. When the ManagedService with service.pid=eric.xyz is registered, the ConfigurationAdmin will retrieve the properties “name=Peter” and “size=8” from its database. The properties will stored in a configuration object and then given to the ManagedService with the updated method.

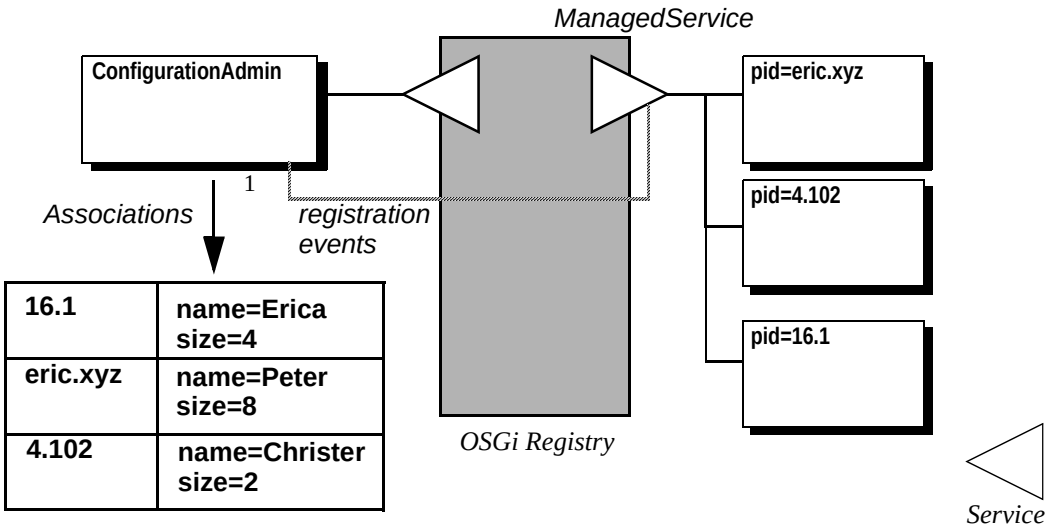

Fig. 4 PIDs and external associations

5.3.3 Sample code using a ManagedService

In this example, a bundle can a run a single console for debugging over a telnet connection. It is a singleton so it uses a ManagedService. The properties are the port and the network it should register on. The following code demonstrates the handling.

```
class SampleManagedService implements ManagedService {
     Dictionary              properties;
     ServiceRegistration     registration;
     Console                 console;

     synchronized void start(BundleContext context) throws Exception {
          properties = new Hashtable();
          properties.put( Constants.SERVICE_PID, “com.acme.console” );
          properties.put( “port”, new Integer( 2011 ) );
          registration = context.registerService(
               ManagedService.class.getName(), this, properties );
     }
```

```
        synchronized void updated( Dictionary newProperties ) {
             if ( newProperties != null ) {
                  properties = newProperties;
                  properties.put( Constants.SERVICE_PID, “com.acme.console” );
             }
             if ( console == null )
                  console = new Console();
             int port = ((Integer) properties.get(“port”)).intValue();
             String network = (String) properties.get(“network”);
             console.setPort( port, network );
             registration.setProperties( properties );
        }

        ...
    }
```

5.4 Factories

A bundle that does not a-priori know how many services are required may register a ManagedServiceFactory with a PID. A ManagedServiceFactory acts as an object factory and will receive separate configurations for each instance it may create.

The configuration objects for a factory are identified by a PID, just like the a ManagedService. When a configuration object for a factory is created (ConfigurationAdmin.createFactoryConfiguration), a new PID is created for this object by the ConfigurationAdmin and it is unique. A configuration object also has a factory PID. This is the pid from the associated factory and is used to group all factory configuration objects together.

When the ConfigurationAdmin detects the registration of a ManagedServiceFactory it must retrieve all configurations for this factory and must then sequentially call ManagedServiceFactory.updated for each. The service should then create instances of the associated class. The service may register new services in the framework using the PID given in the configuration object.

The ConfigurationAdmin must guarantee that the configurations are not deleted before they are given to the service and must assure that no race conditions exist between initialization and updates.

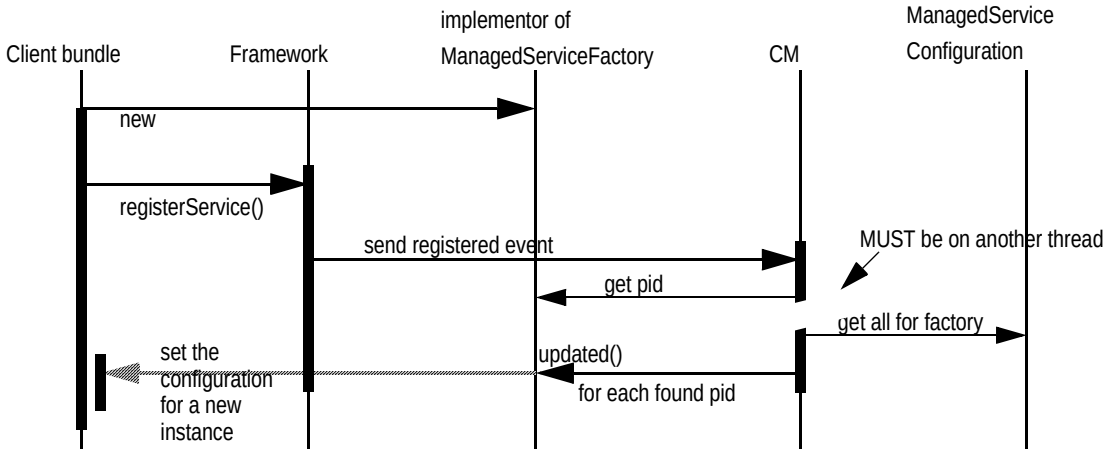

Fig. 5 Action diagram factories

There is only one update method in a ManagedServiceFactory, updated. This method can be called any number of times as configurations are created or updated. The implementation of a ManagedServiceFactory must detect

whether a PID is being used for the first time, in which case it should instantiate a new object, or a subsequent time, in which case it should update an existing object.

The ConfigurationAdmin must call the updated method on a different thread. This allows implementations of ManagedServiceFactories to use “synchronized” to assure that the call-backs do not arrive while the managed-ServiceFactory is registered.

5.4.1 Example factory

This example elaborates on the difference between ManagedServices and ManagedServiceFactories. It shows how the ManagedServiceFactory implementation receives the configuration information that was created before. In this example (“Factory example” on page 13), there is a bundle implementing a mail fetcher. It registers a ManagedServiceFactory with PID=eric.mf. The ConfigurationAdmin notices the registration and consults its database. It finds 3 Configuration objects where the *factory* pid is set to “eric.mf”. The ConfigurationAdmin will call updated for each of these objects on the newly registered ManagedServiceFactory. For each configuration received, the factory will create a new instance of a MailFetcher, one for “erica” (PID=16.1), one for “anna” (PID=16.3) and one for “peter” (PID=16.3). In this case, the mail fetchers are registered under some “Topic” service so their results can be viewed by an online display. The registration is optional. If the object is registered, it may safely be registered under the PID of the configuration object because the ConfigurationAdmin must guarantee its uniqueness.

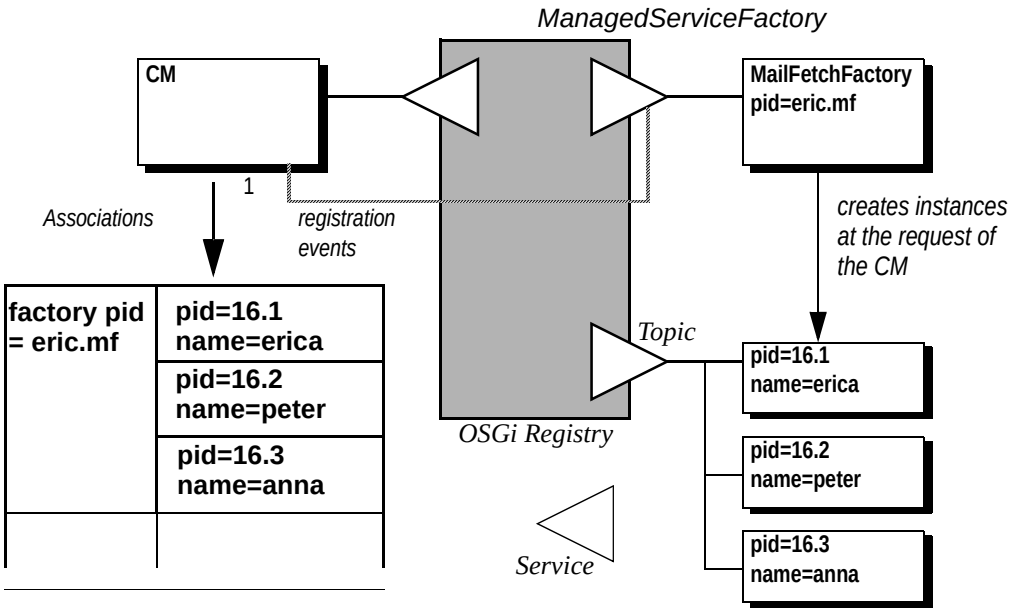

Fig. 6 Factory example

5.4.2 Example factory code

This example allows multiple consoles to be run, each of which has its own port and interface.

```
class ExampleFactory implements ManagedServiceFactory {
     Hashtable              consoles = new Hashtable();

     public void start( BundleContext context ) throws Exception {
          Hashtable local = new Hashtable();
          local.put( Constants.SERVICE_PID, “com.acme.console.server” );
```

```
            context.registerService(
                ManagedServiceFactory.class.getName(), this, local );
        }
        public void updated( String pid, Dictionary config ){
            Console console = (Console) consoles.get( pid );
            if ( console == null ) {
                console = new Console(context);
                consoles.put( pid, console );
            }
            int port = getInt( config, “port”, 2011 );
            String network = getString( config, “network”, null /*all*/ );
            console.setPort( port, network );
        }
        public void deleted( String pid ) {
            console = (Console) consoles.get( pid );
            if ( console != null ) {
                consoles.remove( pid );
                console.close();
             }
        }
    }
```

5.5 The configuration properties

The set of configuration properties are coded as properties in a dictionary. The values that must be used are the same types as the properties supported in the OSGi service registry. This is defined as:

```
type          = String | Integer | Long | Float | Double | Byte | Short | BigInteger
                | BigDecimal | Character | Boolean | vector | arrays
primitive     = long | int | short | char | byte | boolean | double | float
arrays        = primitive ‘[]’ | type ‘[]’
vector        = Vector of type
```

The name of a property must always be a String and is not case sensitive in lookup but must preserve the original case. The use of nested Vector and arrays is strongly discouraged.

An implementor of ManagedService should copy all the properties of the configuration object, known or unrecognized, into its service properties using ServiceRegistration.setProperties(). This convention will allow services to be developed that leverage the OSGi service registry more extensively. However, cooperation with this mechanism is advised but not mandatory. A service may ignore any properties it does not recognize. It may also change the values of the properties from the values in the configuration before these are registered. This implies that properties in the framework registry are not strictly related to configuration properties.

Applications that cooperate with this propagation of properties can then participate in horizontal services. An example of such a service is an application that maintains location information. It offers support by finding out

where a particular device is located in the house. This service could use a property “location” and provide functions that leverage this property.

5.6 Maintaining the configuration

The ConfigurationAdmin API is used to modify the configuration. Maintenance may be done by a remote management system or by the bundle itself. The ConfigurationAdmin has methods for creating/accessing Configurations based on a PID, and methods for creating new Configurations for a ManagedServiceFactory.

5.6.1 Creation of a ManagedService configuration

A bundle can create a managed service configuration with the ConfigurationAdmin.getConfiguration(String pid). Assuming that there is not already an existing Configuration object associated with the speciifed PID, a new Configuration object is created for this PID and placed in persistent storage. The Configuration object has a call getFactoryPid() which will return null in this case.

5.6.2 Creation of a ManagedServiceFactory configuration

To create a new instance via a factory, the ConfigurationAdmin.createFactoryConfiguration(String factoryPid) is called. It creates a new Configuration object, associates it with a newly generated PID, and places it in persistent storage. The Configuration object has a method getFactoryPid() which will return factoryPid in this case.

5.6.3 Location Binding

When a Configuration object is created by either getConfiguration(String pid) or createFactoryConfiguration(String factoryPid), it is *bound* to the location of the calling bundle. This means that any attempt to access this Configuration using getConfiguration(String pid) will result in a SecurityException, unless the caller is either the same bundle that created it or else has AdminPermission. Also, if a ManagedService is registered when there is already a Configuration object created for its PID, the normal call-back to ManagedService.updated does not take place if the Configuration is bound to a location different than that of the registering bundle.

There are also two-argument versions of getConfiguration and createFactoryConfiguration which take a location String as their second parameter. These methods require AdminPermission, and they create Configuration objects bound to the specified location instead of the location of the calling bundle. As a special case, a null location parameter can be used to create Configuration objects that are not bound. In this case, the objects become bound to a specific location the first time that they are used.

For example, an administrative bundle might create a ManagedService Configuration object before the corresponding ManagedService has been registered. It might use a null location to avoid any dependency on the actual location of the bundle which will register this service.. Then, when the ManagedService is registered, the Configuration object will be bound to the location of the registering bundle, and its properties will then be passed to the the ManagedService.updated method.

5.6.4 Accessing existing configurations

Configuration objects may be accessed via a filter expression or directly via a PID. The filter has the same syntax and rules as the filter in the framework. The filter will inspect the properties of the configuration objects and only return the ones that match. If the caller has AdminPermission, all configuration objects are searched. In other cases, only configuration objects bound to the calling bundle’s location are returned.

```
Configuration []
     listConfigurations( String filter );
Configuration getConfiguration( String pid );
```

5.6.5 Updating a configuration

The process of updating a configuration is the same for managed services and factories. Once a Configuration object is obtained, the existing properties can be obtained with Configuration.getProperties(). When no update has happened yet, getProperties() returns a null.

New properties can be set by calling Configuration.update(Dictionary properties). The ConfigurationAdmin first stores the properties and then calls the target. This is either ManagedService.updated where the ManagedService is registered with the configuration's PID, or ManagedServiceFactory.updated where the ManagedServiceFactory is registered with the configuration's factory PID. If this target service is not registered, the updated configuration will be set when the target registers later.

The update calls in Configuration are not synchronous with the target service update. The actual updated call to the target service happens on a different thread some time after the properties have been updated. However, the ConfigurationAdmin must have updated the persistent storage before the function returns.

If a Configuration is no longer needed it can be deleted with Configuration.delete(). This removes the Configuration from the database. The database must be updated before the target service is called.

If the target service was a factory, the factory is informed of the deletion by a call to ManagedServiceFactory.deleted. It should then remove the associated instance. The ManagedServiceFactory.deleted call is done on a separate thread some time after the configuration has been deleted, and hence is asynchronous with respect to the Configuration.delete() method.

When a configuration of a ManagedService is deleted, the ManagedService.updated is called with a null for the properties parameter. This may be used to clean up, revert to default values or stop servicing.

5.6.6 Updating a bundle's own configuration

The model does not distinguish between updates via an official management bundle or a bundle updating its own configuration objects. Even if a bundle updates its own configuration the ConfigurationAdmin must call-back the associated target service.

As a rule, a bundle's UI wanting to update its own configuration should only update the configuration properties and never its internal structures directly. This will have the advantage that the events from the implementation point of view look similar for internal updates, remote management update and initialization.

5.7 Plugins

Plugins allow sufficiently privileged bundles to intercept configuration data before it is passed to the ManagedService or ManagedServiceFactory for which it is intended. Plugins are services registered in the framework registry under the ConfigurationPlugin interface. This interface has only one method: void ConfigurationPlugin.modifyConfiguration(ServiceReference,Dictionary).

All plugins in the registry are traversed and called **before** the properties are passed to the target service. Each `ConfigurationPlugin` gets a chance to inspect the existing properties, look at the target object (which can be a ManagedService or a ManagedServiceFactory). It can also modify the properties.

Obviously, ConfigurationPlugin's should not modify properties that belong to the configuration properties of the target service unless the implications are understood. ConfigurationPlugin functionality is mainly intended to provide functions that leverage the OSGi Framework registry. For example, a ConfigurationPlugin can add a physical location property to a service. This property can be leveraged by applications that want to know where a service is located. This type of scenario could be played without any further support of the service itself, except for the general requirement that the service should store the properties it receives from the ConfigurationAdmin in the registry.

A Plugin may optionally specify a "cm.target" registration property whose value is the PID of the ManagedService or ManagedServiceFactory whose configuration updates the Plugin is intended to intercept. The Plugin will then only be called with configuration updates that are targetted at the ManagedService or ManagedServiceFactory with the specified PID. Omitting the cm.target registration property means that it is called for all configuration updates.

5.7.1 Example: property expansion

There is a target ManagedService which has a configuration property "to" with value "(objectclass=Alarm)". When the ConfigurationAdmin goes to set this property on the target service, a Plugin will replace the (objectclass=Alarm) filter with an array of existing alarm systems's pids:

```
ID  "to=[32434,232,12421,1212]"
```

Now a new Alarm service with pid=343 is registered, so this list needs to be updated. The bundle which registered the Plugin therfore wants to set new properties on the target service. It does not do this by calling ManagedService.updated directly for several reasons: it should not have the permission to do that, and it could get into race conditions with the ConfigurationManager if it did. Instead it gets the Configuration object from the ConfigurationAdmin and calls update on it.

ConfigurationAdmin will schedule a new update cycle and somewhere in the future will call Plugin.modifyProperties(). The Plugin will then set "to" to "[32434,232,12421,1212, 343]. After that, ConfigurationAdmin will call updated on the target service with the right "to" list.

5.7.2 Example: dynamic lists property

A Plugin watches Bluetooth devices. The property bluetooth.computer is a list of bluetooth addresses of computers that are in the neighborhood. The Plugin automatically sets this list. When a new device enters the neighborhood or leaves it, the Plugin will have to call update to be able to set the new address in the list.

5.7.3 Modifications

Modifications to this Dictionary are still under control of the ConfigurationAdmin. It is at the discretion of the ConfigurationAdmin to accept these changes, hide critical variables or deny changes for other reasons.

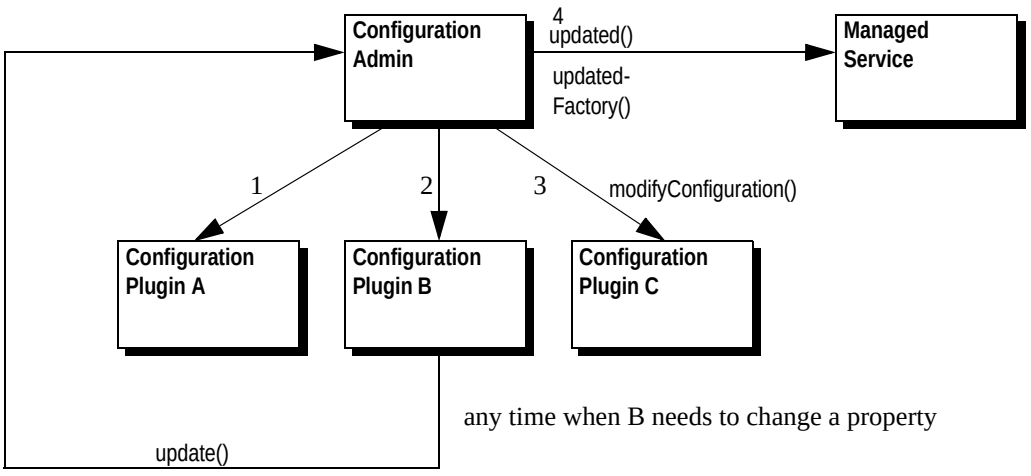

Fig. 7 Plugins

The ConfigurationPlugin interface will also allow plugins to detect configuration updates to the service via the call-back. This allows them to synchronize the configuration updates with transient information.

If a bundle needs to force a plugin to be called again, because something has changed that might cause it to do something different than it did before, it must fetch the appropriate Configuration from the ConfigurationAdmin and call Configuration.update on this object. This starts an update with the current configuration properties, so that all applicable plugins get called again.

The order in which the ConfigurationPlugin's are called will depend on the "service.cmRanking" property of the ConfigurationPlugin service. The following ordering applies:

service.cmRanking < 0, should not modify properties and shall be called **before** any modifications

service.cmRanking >= 0 && <= 1000, modifies data sorted on order

service.cmRanking > 1000, should not modify data and shall be called **after** all modifications are made

6 Meta typing

This RFC uses the meta data from RFC 16, see [11]. This chapter discusses how RFC 16 is used in the context of configuration management.

6.1 Connecting meta types to the CM

6.1.1 Runtime

When a ManagedService or ManagedServiceFactory is registered, the service object can also implement the MetaTypeProvider interface. The bundle may register the service under this interface but this is not required. If the ManagedService or ManagedServiceFactory implements this MetaTypeProvider interface, an implementation may assume that the associated ObjectClassDefinition can be used to configure the service.

The ObjectClassDefinition and AttributeDefinitions contain sufficient information to automatically build rudimentary user interfaces. They can also be used to augment dedicated interfaces with accurate validations.

6.1.2 Management systems

This RFC does not address how the meta type is made available to a management system due to the many open issues regarding remote management.

7 Remote Management

7.1 Concerns

This RFC does not attempt to define a remote management interface for the OSGi framework. The purpose of the RFC is to define a minimal interface for bundles that is complete enough to test. However, configuration man-

agement is a primary aspect of remote management and any configuration management standard should be compatible with common remote management standards. This chapter points out some of the issues of using this RFC with the Common Information Model (CIM) [4] and Simple Network Management Protocol (SNMP) [5], the most likely candidates for remote management today.

These discussions are not complete nor comprehensive and are only intended to point in relevant directions. Further RFC's are needed to make a more concrete mapping.

7.2 CIM

CIM defines the managed objects in an Interface Definition Language (IDL) [7] like language which was developed for CORBA. There is a syntax for the data types and a syntax for the data values. There exists a mapping from this language to XML. Unfortunately, this XML mapping is very different from the very applicable XSchema [6] XML data type definition. It is currently not thoroughly investigated but it seems that the OSGi framework registry property types are a proper subset of the CIM data types.

In this proposal, a factory would map to a CIM class definition. The primitives create, delete and set are supported in this RFC via the ManagedServiceFactory interface. The possible data types in CIM are richer than the framework supports and should thus be limited when CIM classes for bundles are defined.

An important conceptual difference between this RFC and CIM is the naming of properties. CIM properties are scoped inside a class. In this RFC properties are primarily scoped inside the ManagedServiceFactory but are then placed in the registry where they have global scope. This mechanism is like LDAP [8], where the semantics of the properties are defined globally and where a class is a collection of globally defined properties.

The RFC does not address the non-configuration management primitives like notifications and method calls.

7.3 SNMP

SNMP defines the data model in ASN.1. This is a very rich and powerful data typing language that supports many types that are difficult to map to the data types supported in this RFC. However, there is a large overlap and it should be feasible to design a data type that is applicable in this context.

In this case the PID of a ManagedService will map to the SNMP object identifier (OID). ManagedServiceFactories are mapped to tables in SNMP though this creates an obvious restriction in data types because table can only contain scalar values. This would imply that the property values of the Configuration would have to be limited to scalar values.

Similar scoping issues as seen in CIM arise for SNMP because properties have a global role in the registry.

SNMP does not support the concept of method calls or function calls. All information is conveyed as the setting of values. This would map closely to this RFC.

This RFC does not address the non-configuration management primitives like traps.

8 Security Considerations

8.1 Permissions

Security is implemented using ServicePermissions and AdminPermission. The following table summarizes the permissions needed by the Configuration bundle itself, as well as those needed by the bundles with which it interacts.

Bundle registering	ServicePermisson	Admin-Permission	Mode
ConfigurationAdmin	REGISTER ConfigurationAdmin	yes	exclusive for ConfigurationAdmin
	GET ManagedService		exclusive for ConfigurationAdmin
	GET ManagedServiceFactory		exclusive for ConfigurationAdmin
	GET ConfigurationPlugin		exclusive for ConfigurationAdmin
ManagedService	REGISTER ManagedService	no	done by many
	GET ConfigurationAdmin		optional, by many
ManagedService-Factory	REGISTER ManagedServiceFactory	no	done by many
	GET ConfigurationAdmin		optional, by many
ConfigurationPlugin	REGISTER ConfigurationPlugin	no	done by few
	GET ConfigurationAdmin		optional, done by many

The ConfigurationAdmin bundle must have REGISTER ServicePermission for the ConfigurationAdmin service. It will also be the only bundle that needs GET ServicePermission for ManagedService, ManagedServiceFactory and ConfigurationPlugin. No other bundle should be allowed to have GET permission on these services. The ConfigurationAdmin bundle must also hold AdminPermission.

Configurable bundles must have REGISTER permission for ManagedService or ManagedServiceFactory.

Bundles registering ConfigurationPlugins require REGISTER permission for that service. The ConfigurationAdmin must trust all services registered with ConfigurationPlugin. Only ConfigurationAdmin should have a GET permission on ConfigurationPlugin to prevent malicious bundles.

If ManagedService or ManagedService factory is implemented in an object that is also registered under another interface, it would be possible, although inappropriate, for a bundle other than the ConfigurationAdmin to call the updated method. Security aware bundles can avoid any problem with this by having their updated methods check for AdminPermission.

Bundles that want to change their own configuration need GET permission for ConfigurationAdmin. A bundle with AdminPermission is allowed to access and modify any Configuration. Note that doing pre-configuration of bundles requires AdminPermission.

8.2 Forging PIDs

There is a risk of an unauthorized bundle forging a PID in order to obtain and possibly modify the configuration information of another bundle. To mitigate this risk, Configuration objects are generally *bound* to a specific bundle location, and are not passed to any ManagedService or ManagedServiceFactory registered by a different bundle.

Bundles with AdminPermission can create Configuration objects that are not bound, that is that have their location set to null. This can be useful for pre-configuring bundles before they are installed, without having to know their locations. In this scenario, the Configuration will becomed bound to the first bundle which registers a ManagedService (or ManagedServiceFactory if it is a factory configuration) with the right PID. There is still a possibility that a bundle could obtain another bundle's configuration, by registering a ManagedService with the right PID before the victim bundle does so. This can be seen as a kind of denial-of-service attack, since the victim bundle would never receive its configuration information. Such an attack can be avoided by always binding configurations to the right locations.

8.3 Relationship between Configuration and Permission Administration

Configuration information has a direct influence on the permissions needed by a bundle. For example, when the ConfigurationAdmin orders a bundle to use port 2011 for a console, then that bundle also needs permission for listening to incoming connections on that port.

There is a simple and complex case. First, the bundle has a set of permissions that do not define specific values but allow a range of values. For example, a bundle could listen to ports above 1024 freely. All these ports could then be used for configuration.

The other case is more complicated. In an environment where there is very tight security, the bundle would only be allowed a a specific port. This situation requires an atomic update of both the configuration data and the permissions. If this update was not atomic there would be a potential security hole while the set of permissions would not match the configuration.

The following scenario can be used to update a configuration AND the security permissions.

1 Stop the bundle

2 Update the configuration information in the ConfigurationAdmin

3 Update the permissions in the framework

4 Refresh the packages to assure the exported code is also having the right permissions

5 Start the bundle

This scenario would achieve atomicity from the point of view of the bundle.

9 org.osgi.service.cm

Package

org.osgi.service.cm

Class Summary	
Interfaces	
`Configuration`	The configuration information for a ManagedService or ManagedServiceFactory.
`ConfigurationAdmin`	Service for administering configuration data.
`ConfigurationPlugin`	A service interface for processing Configuration data before the update.
`ManagedService`	A service that can receive configuration data from a ConfigurationAdmin.
`ManagedServiceFactory`	Manage multiple service instances.
Exceptions	
`ConfigurationException`	An Exception class to inform the ConfigurationAdmin of problems with configuration data.

org.osgi.service.cm

Configuration

Syntax

```
public interface Configuration
```

Description

The configuration information for a ManagedService or ManagedServiceFactory. The ConfigurationAdmin uses this interface to represent the configuration information for a ManagedService or for a service instance of a Managed-ServiceFactory.

A Configuration contains a set of configuration properties and allows the properties to be updated via this object. Bundles wishing to receive configuration information do not need to use this class - they register a ManagedService or ManagedServiceFactory. Only administrative bundles, and bundles wishing to update their own configurations need to use this class.

The properties handled in this configuration have case insensitive String keys. However, case is preserved from the last set key/value. The value of the property may be of the following types:

```
type        = String | Integer | Long | Float | Double | Byte | Short | BigInteger
              | BigDecimal | Character | Boolean | vector | arrays
primitive   = long | int | short | char | byte | double | float
arrays      = primitive '[]' | type '[]'
vector      = Vector of type
```

A configuration can be *bound* to a bundle location (Bundle.getLocation()). The purpose of binding a Configuration to a location is to make it impossible for another bundle to forge a PID that would match this configuration. When a configuration is bound to a specific location, and a bundle with a different location registers a corresponding Managed-Service or ManagedServiceFactory, then the configuration is not passed to the updated method of that object.

If a configuration's location is null, it is not yet bound to a location. It will become bound to the location of the first bundle that registers a ManagedService or ManagedServiceFactory with the corresponding PID.

The same Configuration object is used for configuring both ManagedServiceFactory and ManagedService. When it is important to differentiate between these two the term "factory configuration" is used.

Member Summary

Methods

void	delete()
String	getBundleLocation()
String	getFactoryPid()
String	getPid()
Dictionary	getProperties()
void	setBundleLocation(String)
void	update()
void	update(Dictionary)

Methods

delete()

```
public void delete()
            throws IOException, IllegalStateException
```

Delete the associated configuration. Removes this configuration object from the persistent store. Notify on a different thread the corresponding ManagedService or ManagedServiceFactory. A ManagedService is notified by a call to its updated method with a null properties argument. A ManagedServiceFactory is notified by a call to its deleted method.

Throws:

`IOException` - If delete fails

`IllegalStateException` - if this configuration has been deleted

getBundleLocation()

```
public java.lang.String getBundleLocation()
            throws SecurityException, IllegalStateException
```

Get the bundle location. Returns the bundle location to which this configuration is bound, or null if it is not yet bound to a bundle location.

This call requires AdminPermission.

Returns: location to which this configuration is bound, or null

Throws:

`SecurityException` - if the caller does not have AdminPermission

`IllegalStateException` - if this configuration has been deleted

getFactoryPid()

```
public java.lang.String getFactoryPid()
            throws IllegalStateException
```

For a factory configuration return the PID of the corresponding ManagedServiceFactory, else return null.

Returns: factory PID or null

Throws:

`IllegalStateException` - if this configuration has been deleted

getPid()

```
public java.lang.String getPid()
            throws IllegalStateException
```

Get the PID for this configuration.

Returns: the PID for this configuration.

Throws:

`IllegalStateException` - if this configuration has been deleted

getProperties()

```
public java.util.Dictionary getProperties()
            throws IllegalStateException
```

Return the properties of this Configuration. The dictionary returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type String.

If called just after the configuration is created and before update has been called, this method returns null.

Returns: A private copy of the properties for the caller or null.

Throws:

`IllegalStateException` - if this configuration has been deleted

setBundleLocation(String)

```
public void setBundleLocation(java.lang.String bundleLocation)
            throws SecurityException, IllegalStateException
```

Bind this configuration to the specified bundle location. If the bundleLocation parameter is null then the configuration will not be bound to a location. It will be set to the bundle's location before the first time a ManagedService/ ManagedServiceFactory receives this Configuration via the updated method and before any plugins are called.

This method requires AdminPermission.

Parameters:

`bundleLocation` - a bundle location or null

Throws:

`SecurityException` - if the caller does not have AdminPermission

`IllegalStateException` - if this configuration has been deleted

update()

```
public void update()
            throws IOException, IllegalStateException
```

Update the configuration with the current properties. Initate the updated call-back to the ManagedService or ManagedServiceFactory with the current properties on a different thread.

This is the only way for a bundle that uses a ConfigurationPlugin to initate a call-back. For example, when that bundle detects a change that requires an update of the ManagedService or ManagedServiceFactory via its ConfigurationPlugin.

Throws:

`IOException` - if update cannot access the properties in persistent storage

`IllegalStateException` - if this configuration has been deleted

See Also: `ConfigurationPlugin`

update(Dictionary)

```
public void update(java.util.Dictionary properties)
            throws IOException, IllegalArgumentException, IllegalStateException
```

Update the properties of this Configuration. Stores the properties in persistent storage after adding or overwriting the following properties:

- “service.pid” : is set to be the PID of this configuration.
- “service.factoryPid” : if this is a factory configuration it is set to the factory PID else it is not set.
- “service.bundleLocation” is set to the location to which this configuration is bound. If the location is null, this property is not set.

These system properties are all of type String.

If the corresponding ManagedService/ManagedServiceFactory is registered, its updated method will be called on another thread. Else, this call-back is delayed until aforementioned registration occurs.

Parameters:

`properties` - the new set of properties for this configuration

Throws:

`IOException` - if update cannot be made persistent

`IllegalArgumentException` - if the dictionary contains invalid configuration types

`IllegalStateException` - if this configuration has been deleted

org.osgi.service.cm
ConfigurationAdmin

Syntax
```
public interface ConfigurationAdmin
```

Description
Service for administering configuration data.

The main purpose of this interface is to store bundle configuration data persistently. This information is represented in Configuration objects. The actual configuration data is a Dictionary of properties inside a Configuration object.

There are two principally different ways to manage configurations. First there is the concept of a ManagedService, where configuration data is uniquely associated with an object registered with the service registry.

Next, there is the concept of a factory where the ConfigurationAdmin will maintain 0 or more configuration objects for a ManagedServiceFactory that is registered with the framework.

The first concept is intended for configuration data about “things/services” whose existence is defined externally, e.g. a specific printer. Factories are intended for “things/services” that can be created any number of times, e.g. a configuration for a DHCP server for different networks.

Bundles that require configuration should register a ManagedService or a ManagedServiceFactory in the service registry. A registration property named “service.pid” (persistent identifier or PID) must be used to identify this ManagedService or ManagedServiceFactory to the ConfigurationAdmin.

When the ConfigurationAdmin detects the registration of a ManagedService, it checks its persistent storage for a configuration object whose PID matches the PID registration property (service.pid) of the ManagedService. If found, it calls ManagedService.updated method with the new properties. The implementation of a ConfigurationAdmin must run these call-backs one a thread that differs from the initiating thread to allow proper synchronization.

When the ConfigurationAdmin detects a ManagedServiceFactory registration, it checks its storage for configuration objects whose factoryPid matches the PID of the ManagedServiceFactory. For each such configuration objects, it calls the ManagedServiceFactory.updated method on a different thread with the new properties. The calls to the updated method of a ManagedServiceFactory must be executed sequential and not overlap in time.

In general, bundles having permission to use the ConfigurationAdmin service can only access and modify their own configuration information. Accessing or modifying the configuration of another bundle requires AdminPermission.

Configurations can be *bound* to a specified bundle location. In this case, if a matching ManagedService or ManagedServiceFactory is registered by a bundle with a different location, then the ConfigurationAdmin must not do the normal call-back, and it should log an error. In the case where a configuration is not bound, its location field is null, the ConfigurationAdmin will bind it to the location of the bundle that registers the first ManagedService or ManagedServiceFactory that has a corresponding PID property.

The method descriptions of this class refer to a concept of “the calling bundle”. This is a loose way of referring to the bundle which obtained the ConfigurationAdmin service from the service registry. Implementations of ConfigurationAdmin must use a ServiceFactory to support this concept.

Member Summary	
Methods	
Configuration	createFactoryConfiguration(String)
Configuration	createFactoryConfiguration(String, String)
Configuration	getConfiguration(String)
Configuration	getConfiguration(String, String)
Configuration[]	listConfigurations(String)

Methods

createFactoryConfiguration(String)

```
public Configuration createFactoryConfiguration(java.lang.String factoryPid)
            throws IOExceptionif, SecurityException
```

Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered ManagedServiceFactory.

The Configuration is bound to the location of the calling bundle.

Parameters:

`factoryPid` - PID of factory (not null)

Returns: a new Configuration object

Throws:

`IOException` - if access to persistent storage fails

`SecurityException` - if caller does not have AdminPermission and factoryPid is bound to another bundle

createFactoryConfiguration(String, String)

```
public Configuration createFactoryConfiguration(java.lang.String factoryPid,
            java.lang.String location)
            throws IOExceptionif, SecurityException
```

Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered ManagedServiceFactory.

The Configuration is bound to the location specified. If this location is null it will be bound to the location of the first bundle that registers a ManagedServiceFactory with a corresponding PID.

This method requires AdminPermission.

Parameters:

`factoryPid` - PID of factory (not null)

`location` - a bundle location string, or null

Returns: a new Configuration object

Throws:

`IOException` - if access to persistent storage fails

`SecurityException` - if caller does not have AdminPermission

getConfiguration(String)

```
public Configuration getConfiguration(java.lang.String pid)
            throws IOException, SecurityException
```

Get an existing or new Configuration from the persistent store. If the Configuration for this PID does not exist, create a new Configuration for that PID, where properties are null. Bind its location to the calling bundle's location.

Else, if the location of the existing Configuration is null, set it to the calling bundle's location.

If the location of the Configuration does not match the calling bundle, throw a SecurityException.

Parameters:

`pid` - persistent identifier

Returns: an existing or new Configuration matching the PID

Throws:

`IOException` - if access to persistent storage fails

`SecurityException` - if the Configuration is bound to a location different from that of the calling bundle and it has no AdminPermission

getConfiguration(String, String)

```
public Configuration getConfiguration(java.lang.String pid, java.lang.String location)
            throws IOException, SecurityException
```

Get an existing Configuration object from the persistent store, or create a new Configuration object.

If a Configuration with this PID already exists in ConfigurationAdmin return it. The location parameter is ignored in this case.

Else, return a new Configuration object. This new object is bound to the location and the properties are set to null. If the location parameter is null, it will be set when a ManagedService with the corresponding PID is registered for the first time.

This method requires AdminPermission.

Parameters:

`pid` - persistent identifier

`location` - the bundle location string, or null

Returns: an existing or new Configuration object

Throws:

`IOException` - if access to persistent storage fails

`SecurityException` - if the caller does not have AdminPermission

listConfigurations(String)

```
public Configuration[] listConfigurations(java.lang.String filter)
            throws IOException, InvalidSyntaxException
```

List the current Configurations which match the filter.

Normally only Configuration objects that are bound to the location of the calling bundle are returned. If the caller has AdminPermission, then all matching Configurations are returned.

The syntax of the filter string is as defined in the OSGi framework's Filter. The filter can test any configuration parameters including the following system properties:

1. service.pid - String - the PID under which this is registered
2. service.factoryPid - String - the factory if applicable
3. service.bundleLocation - String - the bundle location

The filter can also be null, meaning that all Configurations should be returned.

Parameters:

`filter` - an OSGi filter, or null to retrieve all Configurations

Returns: all matching Configurations, or null if there aren't any

Throws:

`IOException` - if access to persistent storage fails

`InvalidSyntaxException` - if the filter string is invalid

org.osgi.service.cm

ConfigurationException

Syntax

```
public class ConfigurationException extends java.lang.Exception

java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--org.osgi.service.cm.ConfigurationException
```

All Implemented Interfaces: java.io.Serializable

Description

An Exception class to inform the ConfigurationAdmin of problems with configuration data.

Member Summary		
Constructors		
		`ConfigurationException(String, String)`
Methods		
	`String`	`getProperty()`
	`String`	`getReason()`

Inherited Member Summary
Methods inherited from class java.lang.Throwable
`getMessage, getLocalizedMessage, toString, printStackTrace, printStackTrace, print-StackTrace, fillInStackTrace`
Methods inherited from class java.lang.Object
`getClass, hashCode, equals, clone, notify, notifyAll, wait, wait, wait, finalize`

Constructors

ConfigurationException(String, String)

```
public ConfigurationException(java.lang.String property, java.lang.String reason)
```

Create a ConfigurationException object.

Parameters:

`property` - name of the property that caused the problem, null if no specific property was the cause

`reason` - reason for failure

Methods

getProperty()

```
public java.lang.String getProperty()
```

Return the property name that caused the failure or null.

getReason()

```
public java.lang.String getReason()
```

Return the reason for this exception.

Returns: reason of the failure

org.osgi.service.cm

ConfigurationPlugin

Syntax

```
public interface ConfigurationPlugin
```

Description

A service interface for processing Configuration data before the update.

A bundle registers a ConfigurationPlugin in order to process configuration updates before they reach the ManagedService or ManagedServiceFactory. The ConfigurationAdmin will detect registrations of ConfigurationPlugin services and must call these services every time before it calls the ManagedService or ManagedServiceFactory updated method. The ConfigurationPlugin thus has the opportunity to view and modify the properties before they are passed to the ManagedService or ManagedServiceFactory.

ConfigurationPlugins have full read/write access to all configuration information. Therefore, bundles using this facility should be trusted. Access to this facility should be limited with ServicePermission REGISTER for the ConfigurationPlugin service. Implementations of ConfigurationPlugin should assure that they only act on appropriate configurations.

The Integer “service.cmRanking” registration property may be specified. Not specifying this registration property, or setting it to something other than an Integer, is the same as setting it to the Integer zero. The service.cmRanking property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones. In the event of more than one plugin having the same value of service.cmRanking, then the ConfigurationAdmin arbitrarily chooses the order in which they are called.

By convention, plugins with `service.cmRanking< 0` or `service.cmRanking > 1000` should not make modifications to the properties.

The ConfigurationAdmin has the right to hide properties from plugins, or to ignore some or all the changes that they make. This might be done for security reasons. Any such behavior is entirely implementation defined.

A Plugin may optionally specify a “cm.target” registration property whose value is the PID of the ManagedService or ManagedServiceFactory whose configuration updates the Plugin is intended to intercept. The Plugin will then only be called with configuration updates that are targetted at the ManagedService or ManagedServiceFactory with the specified PID. Omitting the cm.target registration property means that it is called for all configuration updates.

Member Summary		
Fields		
	`String`	`CM_TARGET`
Methods		
	`void`	`modifyConfiguration(ServiceReference, Dictionary)`

Fields

CM_TARGET

```
public static final java.lang.String CM_TARGET
```

Methods

modifyConfiguration(ServiceReference, Dictionary)

```
public void modifyConfiguration(org.osgi.framework.ServiceReference reference,
            java.util.Dictionary properties)
```

View and possibly modify the a set of configuration properties before they are sent to the ManagedService or the ManagedServiceFactory. The ConfigurationPlugins are called in increasing order of their service.cmRanking property. If this property is undefined or is a non-Integer type, 0 is used.

This method should not modify the properties unless the service.cmRanking of this plugin is in the range `0 <= service.cmRanking <= 1000`.

If this method throws any Exception, the ConfigurationAdmin must catch it and should log it.

Parameters:

`reference` - reference to the ManagedService or ManagedServiceFactory

`configuration` - the configuration properties

org.osgi.service.cm
ManagedService

Syntax
```
public interface ManagedService
```

Description
A service that can receive configuration data from a ConfigurationAdmin.

A ManagedService is a service that needs configuration data. Such an object should be registered with the framework registry with the “service.pid” property set to some unique identitifier called a PID.

If the ConfigurationAdmin has a Configuration object corresponding to this PID, it will call-back the updated() method of the ManagedService, passing the properties of that Configuration object.

If it has no such Configuration object, then it calls back with a null properties argument. Registering a Managed-Service will always result in a call-back to the updated() method provided the ConfigurationAdmin is or becomes active. This call-back is always done on a different thread than the thread that performs the registration.

Else, every time that either of the updated() methods is called on that Configuration, the ManagedService.updated() with the new properties is called. If the delete() method is called on that Configuration, ManagedService.updated() is called with a null for the properties parameter. All these call-backs are done a different thread.

The following example shows the code of a serial port that will create a port depending on configuration information.

```
class SerialPort implements ManagedService {
    ServiceRegistration registration;
    Hashtable configuration;
    CommPortIdentifier id;

    synchronized void open(CommPortIdentifier id, BundleContext context) {
        this.id = id;
        registration = context.registerService(
            ManagedService.class.getName(),
            this,
            getDefaults()
        );
    }

    Hashtable getDefaults() {
        Hashtable defaults = new Hashtable();
        defaults.put( “port”, id.getName() );
        defaults.put( “product”, “unknown” );
        defaults.put( “baud”, “9600” );
        defaults.put( Constants.SERVICE_PID,
                      “com.acme.serialport.” + id.getName() );
        return defaults;
    }

    public synchronized void updated( Dictionary configuration  ) {
        if ( configuration == null )
            registration.setProperties( getDefaults() );
        else
        {
            setSpeed( configuration.get(“baud”) );
            registration.setProperties( configuration );
        }
    }
    ...
}
```

As a convention, it is recommended that when a ManagedService is updated, it should copy all the properties it does not recognize into the service registration properties. This will allow the ConfigurationAdmin to set properties on services which can then be used by other applications.

Member Summary	
Methods	
void	updated(Dictionary)

Methods

updated(Dictionary)

```
public void updated(java.util.Dictionary properties)
            throws ConfigurationException
```

Update the configuration for a ManagedService.

When the implementation of updated(Dictionary) detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem. This can allow a management system to provide useful information to a human administrator.

If this method throws any other Exception, the ConfigurationAdmin must catch it and should log it.

The ConfigurationAdmin must call this method on a thread other than the thread which initiated the call-back. This implies that implementors of ManagedService can be assured that the call-back will not take place during registration when they execute the registration in a synchronized method.

Parameters:

`properties` - configuration properties, or null

Throws:

`ConfigurationException` - when the update fails

org.osgi.service.cm
ManagedServiceFactory

Syntax
```
public interface ManagedServiceFactory
```

Description
Manage multiple service instances. Bundles registering this interface are giving the ConfigurationAdmin the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.

Each of these *service instances* is represented, in the persistent storage of the ConfigurationAdmin, by a factory Configuration that has a PID. When such a Configuration is updated, the ConfigurationAdmin calls the ManagedServiceFactory updated method with the new properties. When updated is called with a new PID, the ManagedServiceFactory should create a new service instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information.

In general it is expected that the implementation of this interface will maintain a data structure that maps PIDs to the service instances that it has created. The semantics of a a service instance are defined by the ManagedServiceFactory. However, if the service instance is registered in the OSGi service registry its PID should match the PID of the corresponding Configuration.

An example that demonstrates the use of a factory. It will create serial ports under command of the ConfigurationAdmin.

```
class SerialPortFactory implements ManagedServiceFactory {
    ServiceRegistration registration;
    Hashtable ports;

    void start(BundleContext context) {
        Hashtable properties = new Hashtable();
        properties.put( Constants.SERVICE_PID, “com.acme.serialportfactory” );
        registration = context.registerService(
            ManagedService.class.getName(),
            this,
            properties
        );
    }

    public void updated( String pid, Dictionary properties  ) {
        String portName = (String) properties.get(“port”);
        SerialPortService port = (SerialPort) ports.get( pid );
        if ( port == null ) {
            port = new SerialPortService();
            ports.put( pid, port );
            port.open();
        }
        if ( port.getPortName().equals(portName) )
            return;

        port.setPortName( portName );
    }
    public void deleted( String pid ) {
        SerialPortService port = (SerialPort) ports.get( pid );
        port.close();
        ports.remove( pid );
    }
    ...
}
```

Member Summary	
Methods	
void	deleted(String)
String	getName()
void	updated(String, Dictionary)

Methods

deleted(String)

`public void` **`deleted`**`(java.lang.String pid)`

Remove a service instance. Remove the service instance associated with the PID. If the instance was registered with the service registry, it should be unregistered.

If this method throws any Exception, the ConfigurationAdmin must catch it and should log it.

The ConfigurationAdmin must call this method on a thread other than the thread which called delete() on the corresponding Configuration object.

Parameters:
`pid` - the PID of the service to be removed

getName()

```
public java.lang.String getName()
```

Return a descriptive name of this factory.

Returns: the name for the factory, which might be localized

updated(String, Dictionary)

```
public void updated(java.lang.String pid, java.util.Dictionary properties)
            throws ConfigurationException
```

Create a new instance, or update the configuration of an existing instance. If the PID of the configuration object is new for the ManagedServiceFactory, then create a new service instance, using the configuration properties provided. Else, update the service instance with the provided properties.

If the service instance is registered with the framework, then the configuration properties should be copied to its registry properties. This is not mandatory and security sensitive properties should obviously not be copied.

If this method throws any Exception, the ConfigurationAdmin must catch it and should log it.

When the implementation of updated detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem.

The ConfigurationAdmin must call this method on a thread other than the thread which necessitated the call-back. This implies that implementors of ManagedServiceFactory can be assured that the call-back will not take place during registration when they execute the registration in a synchronized method.

Parameters:
`pid` - the PID for this configuration

`properties` - the configuration properties

Throws:
`ConfigurationException` - when the configuration is invalid

10 Document Support

10.1 References

[1]. RFP 0005 Request For Proposal for Remote Management

[2]. RFP 0006 Request For Proposal for Execution Profile

[3]. OSGi Framework specification http://www.osgi.org/about/spec1.html

[4]. Common Information Model http://www.dmtf.org

[5]. SNMP RFCs http://directory.google.com/Top/Computers/Internet/Protocols/SNMP/RFCs

[6]. XSchema, http://www.w3.org/TR/xmlschema-0/

[7]. IDL, http://www.omg.org

[8]. LDAP, http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

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

[10]. Understanding and Deploying LDAP Directory services, Timothy Howes et. al. ISBN 1-57870-070-1, Mac-Millan Technical publishing.

[11]. RFC 0016 Meta data for properties

10.2 Author's Address

Name	Peter Kriens
Company	Ericsson
Address	Finnasandsvagen 22, 43933 Onsala, Sweden
Voice	+46 70 5950899
e-mail	Peter.Kriens@aQute.se
Name	David Bowen
Company	Sun Microsystems Inc.
Address	901 San Antonio Road, M/S UCUP01-203, Palo Alto, CA 94303. ISA
Voice	+1 408 343 1407
e-mail	David.Bowen@eng.sun.com