Resource Management

Draft

45 Pages

Abstract

OSGi applications need hardware and software resources to perform their features. As these resources are limited, they have to be fairly shared between applications in order to preserve the global quality of service. Up to now, OSGi platforms delegate the resource management features to the Java Virtual Machines themselves or took advantage of some external Java Resource Management solution like JVM TI or JMX. Unfortunately, all of these resource management solutions provide features at the Object or Class level. This granularity is too low level to easily monitor resources consumed by OSGi applications.

This specification proposes a Resource Management solution fitting with OSGi model and constraints. Resources (CPU, memory, disk storage space, I/O) are monitored per bundle and can be enabled and disabled at runtime. When a bundle consumes too many resources, the Resource Management solution notifies interested applications and limits resource allocations.

0 Document Information

License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

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

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

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

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

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

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

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

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

Trademarks

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

Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

Table of Contents

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.`

Revision History

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

Revision	Date	Comments
Initial	*03 06 2013*	*Initial draft.* *RINQUIN Arnaud, Orange, arnaud.rinquin@orange.com* *BOTTARO André, Orange, andre.bottaro@orange.com* *BONNARDEL Grégory, Orange, gbonnardel.ext@orange.com*
v01	*04 16 2013*	*Some modifications after the Cologne meeting:* • *Global Resource Context is now named as Platform Resource Context* • *Bandwidth Monitor → Socket Monitor* • *Resource Context persistence* • *Add a table indicating the type of Java object the ResourceMonitor.getUsage() method returns for each type of resource.* • *New paragraph about the use of the context switching operation (chapter 6.15 Resource Manager)*
V02	*04 30 2013*	*Fixes:* • *Rewritings* ◦ *Operation Summary* ◦ *CPU Monitor* ◦ *Socket Monitor* ◦ *Framework Resource Context* • *Actions are taken by authorities* *Added:* • *Resource Thresholds* • *Resource Event Types* • *Back To Normal* • *Back To Warning*

Revision	Date	Comments
V03	*05 14 2013*	*Fixes:* • *Rewritings* ◦ *Memory Monitor (OutOfMemoryException)* ◦ *Implementation of all types of Resource Monitor is optional* ◦ *Socket Monitor (existing sockets)* ◦ *BACK_TO_NORMAL Resource Event → NORMAL Resource Event* ◦ *BACK_TO_WARNING Resource Event → WARNING Resource Event* *Added:* • *A new paragraph about algorithms reducing the number of Resource Event into the Considered Alternatives chapter.* • *Socket Monitor scope in Considered Alternatives*
V04	*05 22 2013*	*Fixes:* • *Socket Monitor* ◦ *only tracks bound or connected socket* ◦ *throw a SocketException when an error threshold is reached* • *Rewritings* ◦ *Disk Storage Monitor (exception)* ◦ *Thread Monitor (exception)*
V05	*05 28 2013*	*Modifications:* • *Add Resource Monitor Factory (new paragraph + entity diagram)* • *Change resource monitor threshold list to maximum and minimum threshold attributes*

Revision	Date	Comments
V06	*06 04 2013*	*Modifications:* • *Add ResourceContext.addMonitor() and ResourceContext.removeMonitor() methods.* • *Clarify the Resource Monitor state when a new instance is created by a ResourceMonitorFactory.* • *ResourceEvent.getResourceThreshold() returns a SNAPSHOT of the Resource Threshold instance at the moment when the event was generated.* • *Resource Thresholds:* ◦ *A Resource Monitor holds an upper Resource Threshold instance and a lower Resource Threshold.* ◦ *Add a diagram showing Resource Threshold state transitions. This diagram also shows what kind of Resource Event is generated.* • *Clarify which threads are monitored by a ThreadMonitor (alive thread = all threads which are in the RUNNABLE, BLOCKED, WAITING, TIMED_WAITING java state).*
V07	*06 12 2013*	*Modifications:* • *Refactoring of the Resource Event interface:* ◦ *Add getResourceType() method* ◦ *Add getValue() method* ◦ *Add getThresholdValue() method* ◦ *Add isUppperThreshold() method* ◦ *Remove getMonitor() method* ◦ *Remove getThreshold() method* ◦ *Remove getMonitor() method*
V08	*06 19 2013*	*Updates:* • *Replace ResourceMonitor.setMonitored(boolean) by ResourceMonitor.enable()/disable()* • *Replace ResourceMonitor.isMonitored() by ResourceMonitor.isEnabled()* • *Threshold diagram (new colors + state)*

Revision	Date	Comments
V09	*06 26 2013*	*Updates:* • *Resource Monitor Factory chapter* ◦ *the newly created Resource Monitor instance is disabled by default because it should be configured before activation.* ◦ *Resource Monitor instance MUST be created only by Resource Monitor Factory instance.* • *Add ResourceMonitor,isDeleted() method.* • *Considered Alternatives* ◦ *ResourceMonitor instantiations without ResourceMonitorFactory.*
V10	*07 04 2013*	*Updates:* • *Add a new diagram for upper threshold (cpu example)* • *The upper and lower threshold diagram has been adapted to the socket resource*
V11	*07 10 2013*	• *Replaced arrows by circles into diagrams related to Resource Thresholds. Removed the text on arrows/circle. Update diagram titles.*
V12	*07 18 2013*	• *Remove old references to connected socket → in-use state sockets* • *Remove old references to active/started threads → alive threads* • *Remove old reference to Bandwidth Monitor → Socket Monitor* • *Update Extensibility clause of the Essentials chapter (5 types of resources instead of 4).* • *New use case for the context switching operation (Event Handler use case)*
V13	*07 23 2013*	• *A Socket Monitor tracks native socket file descriptors.* • *Update UML Schema:* ◦ *add Resource Listener Implementer entity* ◦ *add Resource Monitor Factory Implementer entity* • *Update Operation Summary* • *Add new section in Considered Alternatives about compatibility implementation between bundles which handles ResourceManagement api and the other ones.*
V14	*07 30 2013*	• *Add EventAdmin and HttpService use cases into compatibility section* • *Describe how a bundle can manage Resource Management features (direct implementation, weaving, service proxy)*

Revision	Date	Comments
V15	*08 06 2013*	• *ResourceManager entity registered as a service and accessible through the adapt method.*
V16	*08 14 2013*	• *Introduce Resource Context Listener and Resource Context Event* • *All Resource Context are retrieved through the Resource Manager service (Bundle.adapt() approach is deprecated)* • *Thresholds are now hosted by Resource Listener* • *Resource Threshold entities have been removed.*
V17	*09 26 2013*	• *Exceptions thrown when a threshold is reached are now optional.* • *Use of MemoryException instead of OutOfMemoryException*
V18	*11 06 2013*	• *Introduce new methods in ResourceMonitor reporting resource context operations.*
V19	*July 24, 2014*	*Antonin CHAZALET, Orange* • *Fix typos.* • *Add several comments.*
v20	*July 25th, 2014*	*André Bottaro, Orange* • *Removed context switching feature in the specification chapter part. Put the main context switching feature descriptions into 'considered alternatives' section.* • *Partially removed the link between resource context and threads. To be continued.*

1 Introduction

Applications, executed on an OSGi platform, need hardware resources (CPU, memory, disk, storage space) and software resources (sockets, threads). As these resources are limited, applications have to share them in order to preserve system quality of service. This is a general fact in the Entreprise and Residential markets.

Providing fair resource management features is crucial for the Smart Home to emerge as Residential players are opening their gateway (or box) execution environment to third party apllications. In this perspective, the framework administrator has to fairly offer the same guarantees to every actor sharing the platform.

Resource Monitoring is also vital to Cloud Computing scenarios where a management agent needs to ensure that SLAs agreed around the cloud offering are met. When a cloud node gets overloaded or fails this can affect the pre-agreed SLA and action needs to be taken. In a Cloud Computing scenario this may imply starting additional nodes, adjusting the provisioning state of the system by moving or adding deployments or indeed shutting down some nodes if the system become quiet. To be able to handle such scenarios the management agent will need to have visibility of the resource utilization of the cloud system as a whole, which encompasses a multiplicity of nodes and runtimes.

For the moment, existing OSGi specifications do not provide monitoring resource mechanism ensuring a fair resource sharing between bundles and applications. The underlying JVM provides only some standard mechanisms at a level that is too fine-grained, e.g., classes, objects, methods. The bundle being the smallest deployment unit of interest for platform administrator and application provides, this RFC defines an API compliant with the RFP [3].

Introduce the RFC. Discuss the origins and status of the RFC and list any open items to do.

2 Application Domain

Resources of environments are always limited and entities that share such environments should be aware of that. This is not different in OSGi environments. Each bundle consumes resources of different types. Some of them are required for the very basic operations, some others are nice to have, but all of them can run out and lead to situations where the bundle, a set of bundles that form an application, or even the framework as a whole is not operational anymore.

Problematic situations arise when a software unit binds a lot of resources but does not release them after normal operation. This can be caused by wrong implementations, wrong error handling or by intention in case of malware. Especially in environments with very limited resources and/or with a huge number of bundles/vendors it is crucial to monitor the state of bundles and their resource consumption and also to provide mechanisms to react on detected failures.

What are resources?

There are some obvious, basic resources like CPU, memory, disk-space, bandwidth. But new applications might introduce the need for new, different types of resources that are required for their normal operation (e.g., the presence of certain external services and devices, room temperature etc.). Because of that it is impossible to provide a complete list of potential resources here. The following figure tries to illustrate that:

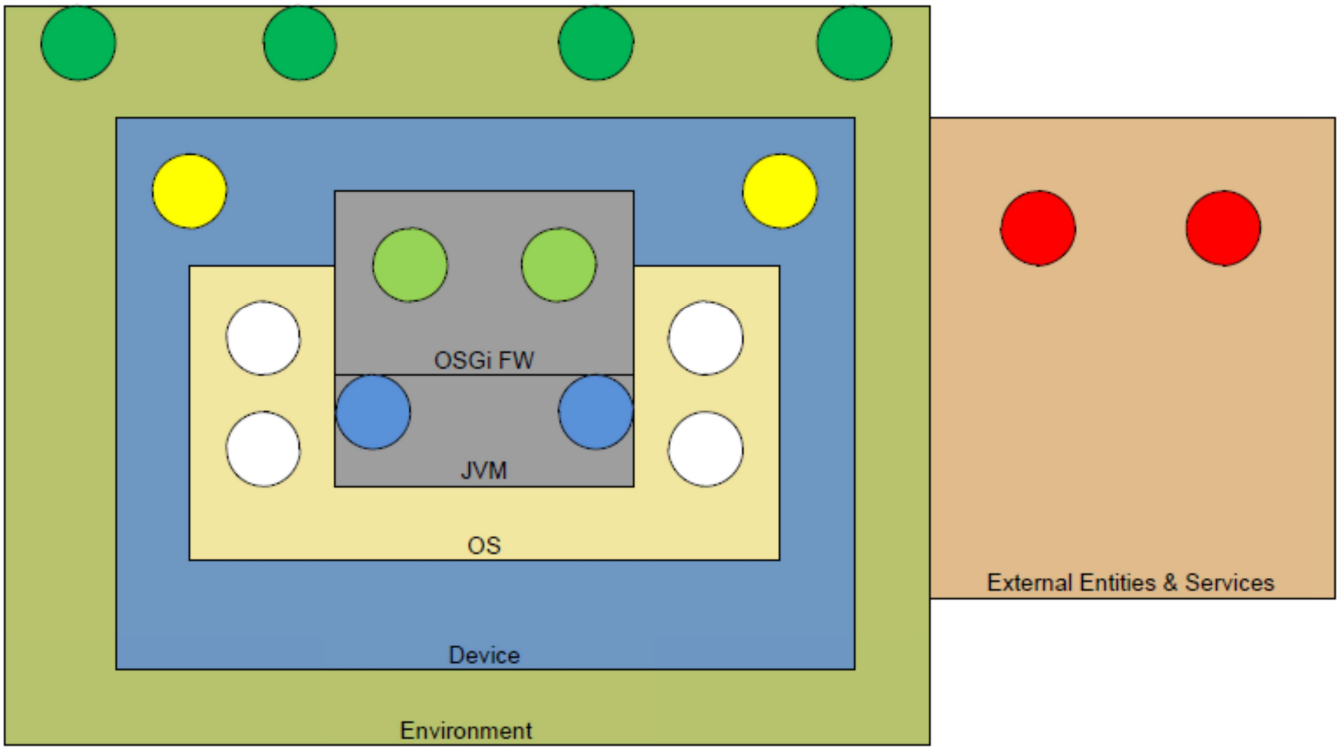


Figure 1: Origins of resources

~~Figure 1: Origins of resources~~

Every circle in this picture stands for a certain resource. As illustrated these resources can come from:

- the same OSGi Framework (e.g., service instances, exported packages …),
- the same Java VM (e.g., threads, memory …),
- the device (e.g., USB-Ports,, network interfaces/ports …),
- the local environment (e.g., room temperature, power consumption of the device, geo-location…),
- or from completely external locations (e.g., special external services like maps, dictionary …).

Most common and crucial resources

Applications use hardware and operating system resources. Targeted resources are:

- CPU
- Memory
- Disk storage space
- Bandwidth on connected networks

JVMs allocate these resources when applications call Java standard APIs. They may provide resource monitoring mechanisms such as:

- Java Management Extension (JMX), now provided by all J2SE-v5-compliant JVM

- JVM Tool Interface Interface(JVMTI) and JVM Profiler Interface (JVMPI)
- Proprietary resource management API (e.g., IBM J9, Oracle Java Embedded Client, /K/ Embedded Mika Max, Myriad Jbed)

The latter provide strict algorithms that charge bundles with consumed resources. There are

two known algorithms [4].:

- Direct accounting: the resources consumed during bundle interaction are accounted to the code provider. In other words, the CPU used by a code that belongs to bundle A will be accounted to A, even if it is the bundle B that called this code through a public interface.
- Indirect accounting: all the resources consumed by the threads belonging to a bundle are accounted to this same bundle. Therefore in service interaction there is no resource consumption accounted to service providers.

Java and OSGi enables CPU management per bundle on any VM (without any VM customization) [4].[5].

However, memory management require that standard VMs either implement JVMTI Java standard [8]. or implement custom APIs. [6].[7]. These features are not available on standard VMs, with the following definition: a standard Java platform implements the JVM Specification [9]., Java Language Specification [10]. and base class libraries (http://docs.oracle.com/javase/7/docs/api). It does not include tools like javac, javadoc, JVMTI, tools.jar which are outside the Java platform specification.

What is Healthiness?

Healthiness of an entity (service, bundle, set of bundles, or the whole framework) is meant as the state where the entity is operational as it was specified and will be for the foreseeable future. The correct operation of such an entity is often strongly related to the availability, and perhaps a certain quality, of resources that the entity needs to work. That means an entity that doesn't have or get the required resources is not healthy. There might also be intermediate states where mandatory resources are there, but some optional ones are not available.

Other reasons for non-healthy entities are potential failure situations either inside the entity itself or in their environment. Sometimes such conditions cause shortage of other resources, which at the end affects other entities as well.

So, in order to ensure the healthiness of entities the first step that should be done is to ask themselves, "how do you feel?", and the second step is to know the entities resources requirements, and to monitor their availability.

Terminology and abbreviations

Application

A set of bundles needed to render a full application to the user.

Observable

An entity that is subject of Health monitoring. In the scope of this document this can be a framework, a bundle or a set of bundles.

Health

The state of an observable that describes its ability to work as specified.

Resource

A limited source or supply of physical or virtual goods that are used by bundles in order to provide their service(s).

Fault

The term fault is usually used to name a defect at the lowest level of abstraction, e.g., a memory cell that always returns the value 0.

Error

A fault may cause an error, which is a category of the system state.

Failure

An error, in effect, may lead to a failure, meaning that the system deviates from its correctness specification.

This section should be copied from the appropriate RFP(s). It is repeated here so it can be extended while the RFC authors learn more subtle details.

3 Problem Description

OSGi platforms host several applications which are executed concurrently. These applications have to share limited resources between them.

Cooperative applications

These mechanisms should also allow to estimate the severity of the situation and to decide for required actions to recover the intended state. Ideally, this should be done in cooperation with the bundle that causes the failure. If a failure situation is detected and can be assigned to a certain bundle, then first this bundle should have the chance to take actions to come back to a healthy state. If this is not successful, then appropriate actions must be taken by another entity.

Due to the wide range of potential failures and the definition of resources as very generic and application specific, this can not be achieved by a fixed and inflexible mechanism that handles a fixed set of predefined problems.

What is needed is a flexible framework that allows dynamic provisioning of modules to:

- collect information about resource requirements, and further, the normal, intended states of the monitored entities,
- monitor those resources (as defined above) and ask services for their health status,
- warn interested and legitimate applications when monitored consumptions are above thresholds,
- evaluate the severity of deviations of the currently monitored state from the intended state,
- take decisions and perform actions to recover the intended state,
- control/monitor the success of the actions taken.

Less cooperative or legacy applications

In case of an application consumes too much resources, it may affect the quality of service of the other applications installed on the platform. Those situations have to be prevented by OSGi platforms.

As described in the previous chapter, JVMs may provide resource management mechanisms. However, all these solutions are designed to monitor low granularity elements: e.g., threads, classes, objects or methods.

As such, these data are of limited interest and there is a need to raise the abstraction to the primitive deployment unit in OSGi, bundles and applications (or sets of bundles). This encourages the specification of a standard unified OSGi-level API managing resources of bundles and sets of bundles installed on the platform.

This section should be copied from the appropriate RFP(s). It is repeated here so it can be extended while the RFC authors learn more subtle details.

4 Requirements

R1: The solution MUST provide at least one resource accounting algorithm (e.g., direct accounting algorithm).

R2: The solution MUST monitor resources per bundle or per bundle set.

R3: The resource monitoring solution MUST be configurable, enabled and disabled at runtime per bundle or per bundle set.

R4: The solution MUST monitor the following resources, if relevant on the underlying (hardware and software) platform:

- CPU
- Memory
- Disk storage space
- Bandwidth on any connected network

R5: The solution MUST provide a mechanism to list the resource types that can be monitored on the underlying (hardware and software) platform.

R6: The solution MUST allow the setting of a warning threshold and an error threshold per bundle or set of bundles.

R7: The solution MUST send events while a bundle or a bundle set is exceeding one of the two thresholds defined by R6.

R8: The solution MUST define CPU thresholds as a percentage of use over a configurable period.

R9: The solution MUST define memory thresholds as bytes.

R10: The solution MUST define disk storage space thresholds as bytes.

R11: The solution MUST define thread thresholds as a number of threads.

R12: The solution MUST define socket thresholds as a number of opened sockets.

R13: The solution MUST be able to lower bundle thread priorities while CPU error threshold is reached.

R14: The solution MUST raise an error (e.g., OutOfMemoryError) and MUST prevent further memory allocation while memory error threshold is reached.

R15: The solution MUST raise an error (e.g., IOException) and MUST prevent further disk storage space allocation while disk storage space error threshold is reached.

R16: The solution MUST raise an error (e.g., InternalError) and MUST prevent further thread activation while thread error threshold is reached.

R17: The solution MUST raise an error (e.g., IOException) and MUST prevent further connected-state socket while socket error threshold is reached.

R18: The solution MUST define means for bundles to define their intend resource usage.

R19: The solution MUST allow OSGi applications to monitor bundles, evaluate their states and take decisions to react gracefully.

R20: The solution MAY define optional means for a bundle to resolve its own conflicts based on the decisions of the entity introduced in R19.

R21: Thanks to notification from R6, an application able to monitor the success of R20 MAY take actions, if the conflicts are not resolved after a period of time. Default action MAY be that the framework mechanism resolve this conflict.

R22: The solution MUST provide a mechanism that allows to plug application specific components to evaluate application specific resources.

R23: Special (non standard, see standard Java Runtime definition in section 2) Java platform implementations MAY be necessary to support management of certain resource types.

5 Initial Specification Chapter

Essentials

Monitoring – Bundle execution resource usage is monitored.

~~Quota assignment – The Resource Manager limits the usage of resources according to resource quotas as declared by Resource Manager Authority.~~

Granular activation – The resource manager can be activated and deactivated per bundle or per bundle set.

Extensibility – five resource types are specified (CPU, memory, disk storage, alive thread and in-use sockets). The list of monitored resource types is extensible and query-able.

~~Coercion – the resource manager executes restrictive actions when resource limits are exceeded.~~

Eventing – the resource manager notifies interested entities of exceeded limits.

Entities

Resource Context – A logical entity for resource accounting. A context may be related to a single bundle or a set of bundles.

System Resource Context – Resource context of the core framework.

Platform Resource Context – A Resource context monitoring the resource usage of the platform as a whole.

Resource Monitor – Monitors the usage of a specific resource type for a specific Resource Context. Resource Monitors track resource usage. They hold Resource Thresholds instances. Resource Monitor object implementation may depend on standard or proprietary JVM APIs, and on operating system features.

Resource Monitor Factory – a factory creating Resource Monitor instances for every Resource Context.

CPU Monitor – Resource Monitor used to monitor CPU.

Memory Monitor – Resource Monitor used to monitor memory.

Socket Monitor – Resource Monitor used to monitor socket resource.

Disk Storage Monitor – Resource Monitor for disk storage usage.

Thread Monitor – Resource Monitor used to monitor alive Java Thread objects.

Resource Listener – A Resource Listener receives resource threshold notifications.

Resource Event – A Resource Event defines a notification to be sent to Resource Listener instances.

Resource Context Listener – A Resource Context Listener receives notifications about resource context creation and configuration.

Resource Context Event – A Resource Context Event defines a notification to be sent to Resource Context Listeners instances.

Resource Manager – This is a singleton entity which manages Resource Context instances. It is used to create new Resource Context instances and to enumerate existing contexts. ~~It also provides methods to make context switching at runtime.~~

Resource Management Authority – Makes any decision to ensure the quality of the service of the system. They use the Resource Manager to create Resource Context instances. It configures them by adding bundles and Resource Monitors.

~~*Resource management class diagram specification*~~

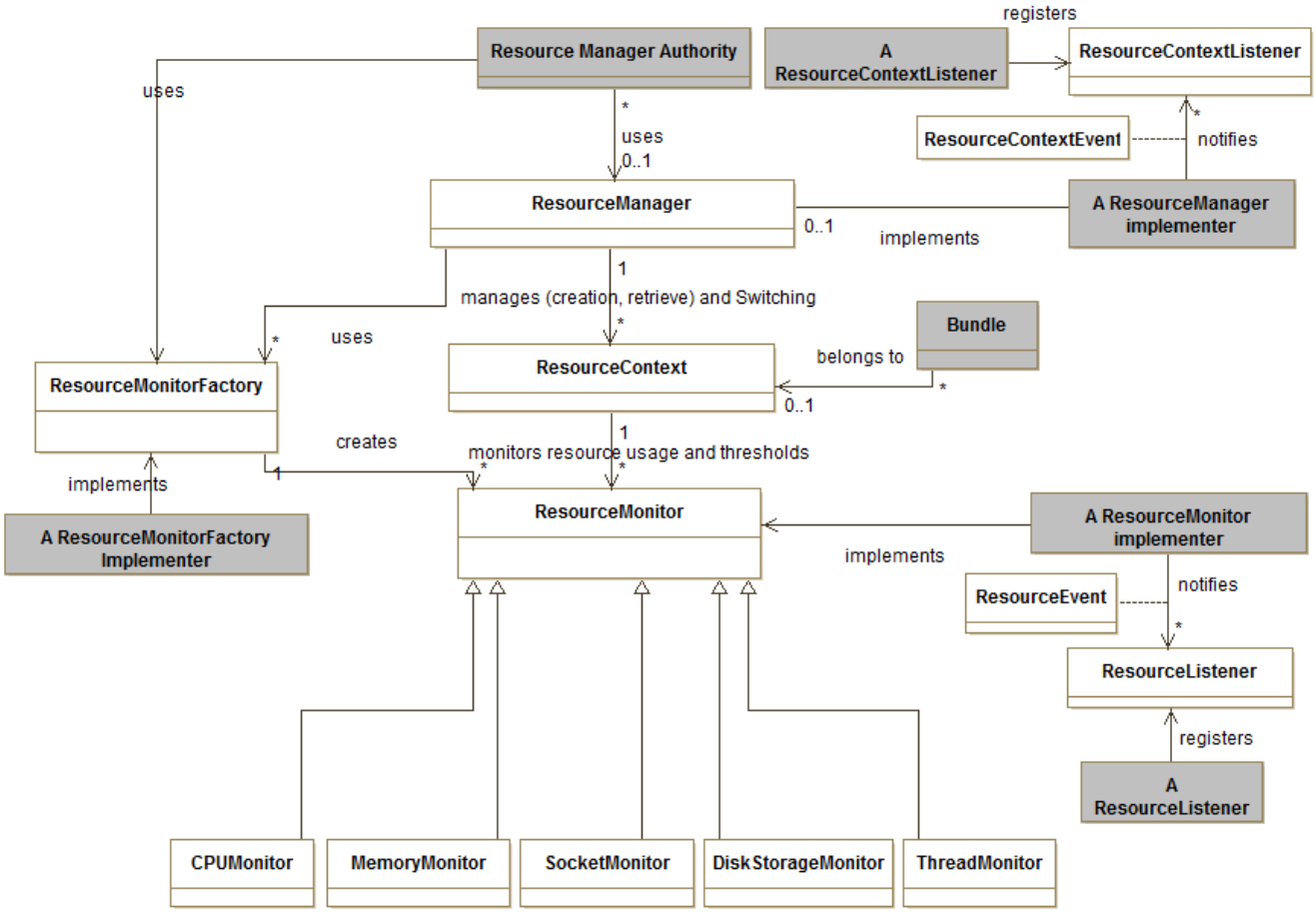


Figure 2: Resource management class diagram specification

Operation summary

Resource Management Authorities use the Resource Manager service to create Resource Contexts. These authorities set bundles or group of bundles to Resource Contexts. They also request every Resource Monitor Factory to create Resource Monitors for a resource type. These Resource Monitors are associated to a single Resource Context.

When activated, Resource Monitors provide the current resource usage per Resource Context. Then, they check whether the current resource usage is compatible with the thresholds held by their associated Resource Listeners. When one of these thresholds is violated, the related Resource Monitor notifies the Resource Listener holding this threshold.

The Resource Manager manages the set of Resource Contexts. Resource Contexts are persistent between platform restarts. Resource Context Listeners are notified when a Resource Context is created or deleted or when a Resource Context configuration (i.e., adding or removing of bundle) is updated.

~~The Resource Manager service allows for resource context switching. This feature is used by authorized entities to set the right Resource Context to be charged for consumed resources at any time.~~

Resource Context

A Resource Context instance is a logical entity used to account resource usage. Every Resource Context defines a bundle scope which can be either a single bundle or a set of bundles. Once the bundle scope is defined, resources used by those bundles are monitored through a set of per-resource-type Resource Monitor instances.

Resource Context instances are persistent. The persistence of those instances is directly managed by the Resource Manager instance.

Each Resource Context is uniquely identified by a name. It can be retrieved through the getName() method. It can not be changed, i.e. it is definitively set when the Resource Context instance is created.

The Resource Context bundle scope is retrieved through the getBundles() method. This bundle scope can be extended through the addBundle(Bundle) method. Bundles can also be removed from a Resource Context through the removeBundle(Bundle, ResourceContext) method. For this last method, a Resource Context instance MAY be specified in order to associate the removed bundle to another Resource Context instance.

Resource Monitor instances are retrieved through getResourceMonitor(String resourceType) method or the getMonitors() method. The list of available resource types is retrieved through the Resource Manager singleton instance.

Resource Monitor instances are added to and removed from a Resource Context instance by calling either addMonitor() method or removeMonitor() method. Both methods SHOULD only be called by ResourceMonitorFactory instances (see ResourceMonitorFactory.createMonitor() method).

A Resource Context is retrieved through the Resource Manager service.

A Resource Context instance can be deleted through removeContext(ResourceContext) method. The Resource Context input argument then defines a destination Resource Context instance for the bundles belonging to the to-be-removed Resource Context instance.

System Resource Context

The System Resource Context is the Resource Context of the core framework. It is retrieved through the Resource Manager service.

The name of this context is “system”.

Framework Resource Context

The Framework Resource Context is a Resource Context monitoring resources of the platform as a whole. It is retrieved through the Resource Manager service. This Resource Context holds all hosted bundles allowing access to the whole platform resource consumption.

The name of this context is “framework”.

Resource Monitor

A Resource Monitor instance monitors a resource type consumed by the bundles of a specific Resource Context instance.

A Resource Context instance holds at most one Resource Monitor instance per monitor-able resource type. Resource Monitor instances are retrieved through their related Resource Context instance. Resource Monitor instances give access to their related Resource Context instance through a call to getContext() method.

The monitored resource type is retrieved through getType() method.

The current usage of a resource consumed by a Resource Context instance is given through getUsage() method. This method returns a Java Object to be casted to the appropriate Java object type depending on the Resource type. The next table provides the expected Java Object type for each specified resource type:

Type of Resource	Expected Java Object type	Value description
CPU	Long	Cumulative CPU time in ns
Memory	Long	Allocated memory in bytes
Threads	Long	Number of alive thread.
Socket	Long	Number of in-use socket.
Disk storage space	Long	Bytes on the bundle persistent storage area

For example, for a Memory Monitor instance, a call to MemoryMonitor.getUsage() returns a Long java object indicating the amount of memory the related Resource Context instance is consuming.

A Resource Monitor instance is enabled and disabled through enable() and disable() methods. The state (enabled or disabled) of a Resource Monitor is retrieved through a call to isEnabled() method.

A Resource Monitor instance can also be deleted (delete() method). isDeleted() method returns true if the ResourceMonitor instance has been deleted.

~~A ResourceMonitor is notified when a context switching operation occurs. When a thread joins a ResourceContext, all ResourceMonitors of the incoming ResourceContext are notified through a call to ResourceMonitor.notifyIncomingThread(thread). On the opposite, when a thread leaves a ResourceContext, all ResourceMonitors of the outgoing ResourceContext are notified through a call to ResourceMonitor.notifyOutgoingThread(thread).~~

Five types of Resource Monitor are specified:

- CPU Monitor
- Memory Monitor
- Socket Monitor
- Disk Storage Monitor
- Thread Monitor

The support of any Resource Monitor is optional. This list MAY be extended by the solution vendor. The list of the types that are supported on the OSGi platform can be computed by querying ResourceMonitorFactory services.

Resource Monitor Factory

A Resource Monitor Factory is a service that provides Resource Monitor instances of a specific resource type (e.g., CPUMonitor, MemoryMonitor...) for every Resource Context.

Every Resource Monitor Factory service is registered with the org.osgi.resourcemanagement.ResourceType mandatory property. This property indicates which type of Resource Monitor a Resource Monitor Factory is able to create. The type can also be retrieved through a call to ResourceMonitorFactory.getType(). The type MUST be unique (two Resource Monitor Factory instances MUST not have the same type).

New Resource Monitor instances are created by a call to createResourceMonitor(ResourceContext). This method returns a new Resource Monitor instance associated to the provided Resource Context instance (the ResourceMonitorFactory MUST call ResourceContext.addMonitor() to associate the newly created ResourceMonitor with the provided ResourceContext instance). The newly created Resource Monitor is disabled, i.e., it is initially not monitoring the Resource Context resource consumption. It can be activated through a call to ResourceMonitor.enable().

Resource Monitor instances are deleted by calling ResourceMonitor.delete() method.

A Resource Monitor instance MUST only be created through its ResourceMonitorFactory.

Resource Monitor Factory instances should be only used by the Resource Manager singleton instance. The Resource Manager singleton instance performs a service lookup on all existing Resource Monitor Factories ~~and computes the list of supported type (ResourceManager.getSupportedTypes())~~. It uses a Resource Monitor Factory instance when it has to create a new Resource Context instance and their associated Resource Monitor instances.

CPU Monitor

A CPU Monitor instance is a Resource Monitor used to monitor the CPU usage of the bundles belonging to a Resource Context.

CPU usage and thresholds are expressed as a cumulative number of nanoseconds (Long). The encapsulated~~is~~ value can be retrieved with the CPUMonitor.getCPUUsage() method.

In case where a threshold is reached, the CPU Monitor instance generates an event triggering Resource Authorities defined corrective actions (e.g., decrease thread priority).

Memory Monitor

A Memory Monitor instance monitors and limits the memory used by the bundles of a Resource Context instance.

Memory is accounted as bytes. Memory usage and thresholds are Long java objects. ~~Memory usage~~ The encapsulated value can be retrieved through the getMemoryUsage() method.

When an error threshold is reached, the next memory allocation MAY throw a MemoryException in the associated context.

Socket Monitor

A Socket Monitor instance monitors and limits the number of existing sockets (e.g., TCP, UDP) which are considered to be in use (e.g., listening for incoming packet, bound, or sending outgoing packets).

A Socket is considered to be in-use state when a native socket file descriptor is created. It leaves this state when this socket file descriptor is deleted.

The number of in-use sockets is a Long. The~~is~~ encapsulated value can be retrieved using SocketMonitor.getSocketUsage() method.

When an ERROR threshold is reached, the next socket file descriptor creation in the associated context MAY throw a SocketException.

Disk Storage Monitor

A Disk Storage Monitor instance monitors and limits the use of persistent storage within Bundle Persistent Storage Area a Resource Context (the bundles actually belonging to it) consumes.

Disk Storage is expressed as a number of bytes of type Long. The encapsulated value can be retrieved using DiskStorageMonitor.getUsedDiskStorage() method.

A IOException MAY be thrown in the associated context when an error threshold is reached.

Thread Monitor

A Thread Monitor instance monitors and limits the number of alive Java Thread objects for a Resource Context instance. A Thread is considered to be alive when it is in the RUNNABLE, BLOCKED, WAITING or TIMED_WAITING java state.

Usage and thresholds are Java ~~Integer~~Long objects. The encapsulated value can be retrieved using ThreadMonitor.getAliveThreads() method.

When an error threshold is reached, any further thread activation will be prevented in the associated context. An InternalError exception MAY also be thrown in the associated context.

Resource Listener

A Resource Listener receives notifications about resource usage for a specific Resource Context and a specific type of resource. A notification will be sent to a Resource Listener when one of its thresholds is violated.

A Resource Listener holds two types of threshold:

- A lower threshold type. This kind of threshold is reached when the monitored resource usage decreases below the threshold.
- An upper threshold type. An upper threshold is reached when the monitored resource usage exceeds this threshold.

Each of them have two levels:

- a WARNING level
- an ERROR level.

A threshold has the following state diagram, which transitions are associated to events:

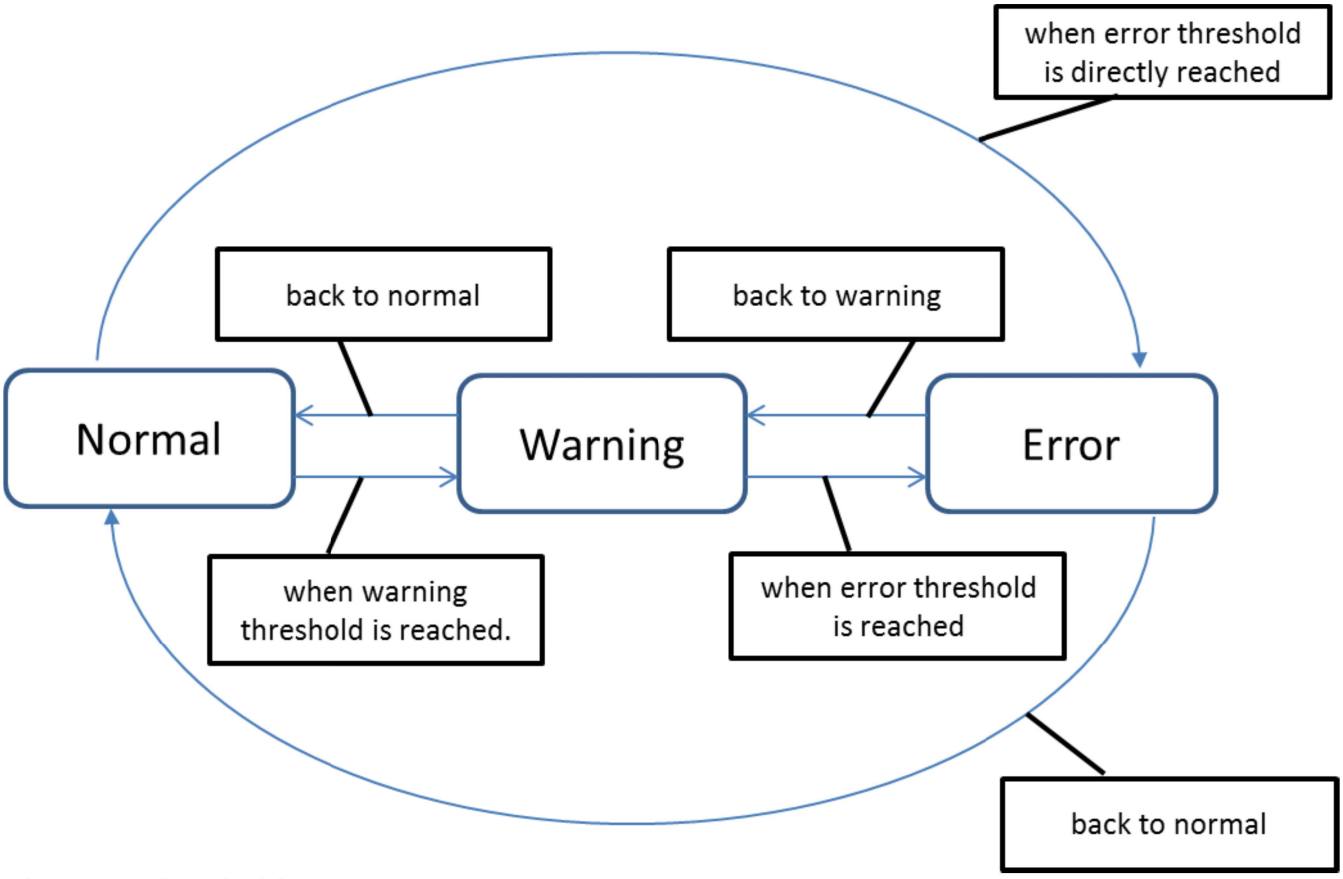


Figure 3: Threshold state diagram

A threshold state depends on the current consumption of resource and the type of threshold (upper or lower threshold).

A Resource Listener is registered as an OSGi service. The implementer must provide the two following mandatory properties:

- RESOURCE_CONTEXT property – a String defining the name of Resource Context for which the Listener want to receive threshold notifications.
- RESOURCE_TYPE property – a String defining which type of resource the listener wants to monitor.

It also has to provide at least one of these four properties when registered as an OSGi service:

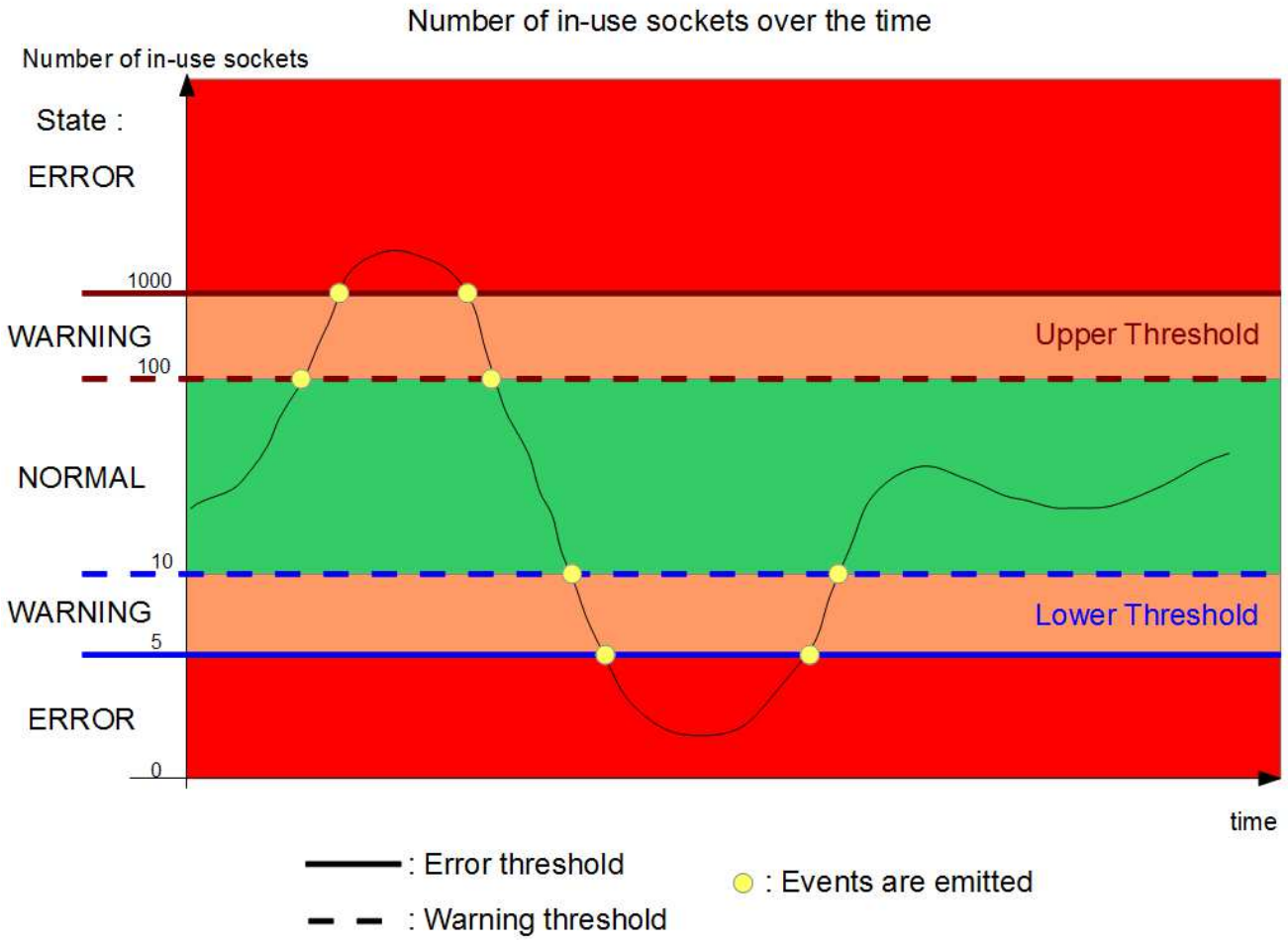
- UPPER_WARNING_THRESHOLD
- UPPER_ERROR_THRESHOLD
- LOWER_WARNING_THRESHOLD
- LOWER_ERROR_THRESHOLD

These properties are mapped to the four types of threshold values a Resource Listener may support. The service properties are used to notify the associated Resource Monitor when one of these threshold values is modified.

Threshold values can also be retrieved through a set of getter methods. All of these methods returns a Comparable object used by the associated Resource Monitor in order to determine the current state of the current usage.

RESOURCE_CONTEXT and RESOURCE_TYPE properties are used by Resource Monitors to identify their associated Resource Listeners. Once associated, a Resource Monitor retrieves the threshold settings using service properties. When one of its thresholds is reached, the Resource Monitor calls ResourceListener.notify(ResourceEvent).

Two examples of resource consumption are explained below, first with in-use sockets monitoring, second with CPU monitoring. The next picture shows the state diagram of the number of in-use state socket over the time.

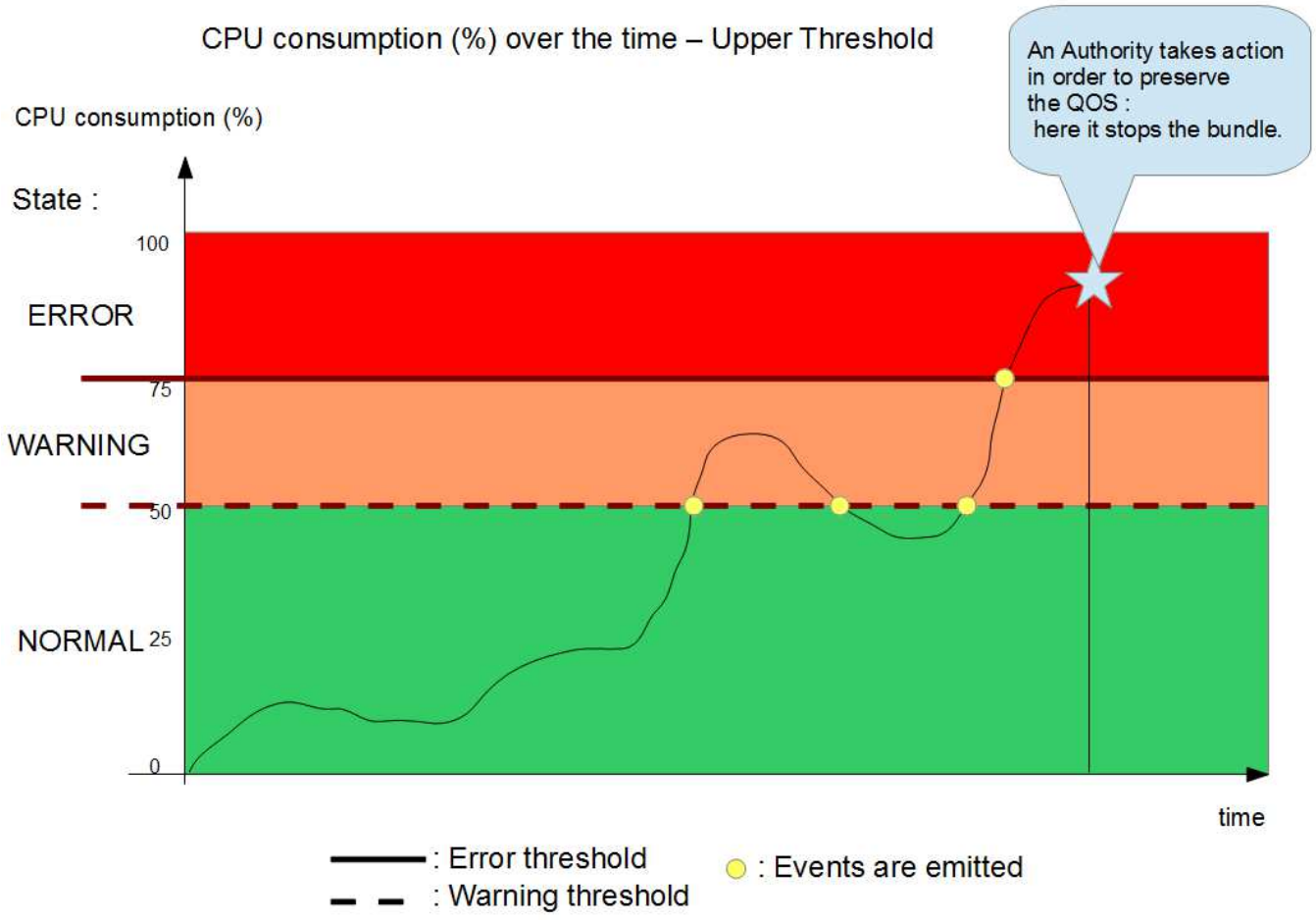


In our example, the lower warning threshold and the lower error threshold of the Resource Listener are respectively set to 10 and 5. When the number of in-use sockets decreases under 10, the usage goes from the NORMAL state to the WARNING state and the Resource Listener receives a WARNING event. If the number of in-use state sockets decreases again and goes down to 5, the usage goes from the WARNING state to the ERROR state and the Resource Listener receives a ERROR Resource Event.

The upper threshold is also set. The upper warning threshold and the upper error threshold are respectively set to 100 and 1000 in-use state sockets. When the number of sockets reaches 100, the usage goes from the NORMAL state to the WARNING state and the Resource Listener receives a WARNING Resource Event. If this number is still increasing and exceeds 1000, then the usage goes from the WARNING state to the ERROR state and the Resource Listener receives an ERROR Resource Event.

This is a typical use case for a Java Web server. Indeed, one of the most important quality of service indicator is the number of in-use state sockets a java web server is handling. A low number of in-use state sockets may indicate the java web server encounters network problems. On the contrary, a high number of in-use state socket may be the result of an external network attack or it could also indicates the java web server is overused and its administrator should take actions to load-balance the charge to another java web server instance.

For other resource types, only upper thresholds may be useful. The next diagram shows the CPU consumption a Resource Context is using over the time:



In this example, only the upper threshold is set. The upper warning threshold is set to 50%, the error one is set to 75%. CPU consumption fluctuates between 0 and 50%, the usage is in the NORMAL state. Then it increases and reaches 50%. The usage then goes from the NORMAL state to the WARNING state and the Resource Listener holding the threshold receives a WARNING Resource Event.

Afterwards, CPU consumption decreases under 50%; the usage goes from the WARNING state to the NORMAL state. The related Resource listener receives a NORMAL Resource Event.

It then increases again and exceeds 50%. The usage goes to the WARNING state. CPU consumption is still increasing and exceeds 75%. At this moment, the usage goes from the WARNING state to the ERROR state and the related Resource Listener receives an ERROR Resource Event.

After some seconds in the ERROR state, the Resource Listener implementation stops the bundle in order to preserve the quality of service.

The choice of the type of threshold (lower or upper, or both of them) depends on the type of resource and the needs of the Resource Management Authorities providing the Resource Listener. Other resources like the free memory may take advantage of a lower threshold.

Resource Event

A Resource Event instance is an event sent to a Resource Listener when one of its thresholds is reached. This event is notified to a Resource Listener through a call to ResourceListener.notify(ResourceEvent).

A Resource Event has a type among the following ones:

- ERROR – The resource consumption reaches either the upper or the lower error threshold of the Resource Listener receiving this event,
- WARNING – The resource consumption reaches either the upper or the lower warning threshold of the Resource Listener receiving this event.
- NORMAL – The resource consumption is back from warning or error state to normal state.

The Resource Listener instance analyzes this event by calling the following methods:

- getValue() method returns the resource consumption at the time when the Resource Event instance was generated.
- isUpperThreshold() method returns true if the reached threshold is an upper threshold type. If this method returns false, this is a lower threshold.
- getType() method indicates the state (WARNING, ERROR, or NORMAL) of the resource usage.
- getContext() method returns the Resource Context instance related to this event. The Resource Listener can use it to retrieve the Resource Monitor instance (e.g., event.getContext().getMonitor(event.getResourceType()).

Resource Context Listener

A Resource Listener instance receives notifications about Resource Context lifecycle and configuration.

A notification will be sent when:

- A Resource Context is created.
- A Resource Context is updated, i.e., a bundle has been added or removed from a Resource Context instance.
- A Resource Context is deleted.

An application which is interested in notifications has to register a Resource Context Listener instance as an OSGi service. The application may provide a set of properties at registration time to reduce the number of notifications a Resource Listener instance will receive. The available properties are:

- RESOURCE_CONTEXT property – An array of String defining the name of Resource Context instances. If defined, a Resource Listener instance will only receive notifications related to these specified Resource Context instances.
- RESOURCE_TYPE property – an array of integers defining the type of notifications a Resource Context Listener instance will receive (see types defined in section Resource Context Event).

A Resource Context Listener instance is notified through a call to notify(ResourceContextEvent) method.

Resource Context Event

A Resource Context Event instance is an event sent to Resource Context Listener instances through a call to ResourceContextListener.notify(ResourceContextEvent) method.

A Resource Context Event has a type among the four following ones:

- RESOURCE_CONTEXT_CREATED – A new Resource Context instance has been created.
- RESOURCE_CONTEXT_REMOVED – A Resource Context instance has been deleted.
- BUNDLE_ADDED – A bundle has been added in the scope of a Resource Context instance
- BUNDLE_REMOVED – A bundle has been removed from the scope of a Resource Context instance.

In the case of a RESOURCE_CONTEXT_ADDED event or a RESOURCE_CONTEXT_REMOVED event, a call to getContext() returns the targeted Resource Context instance.

In the case of a BUNDLE_ADDED type or BUNDLE_REMOVED type, getBundle() returns the Bundle object to be added to or removed from. The related Resource Context instance is given by a call to getContext().

Resource Manager

The Resource Manager service manages the Resource Context instances. ~~Moreover the Resource Manager service informs about the type of resource that the framework is able to monitor.~~

~~It also provides Resource Context switching mechanism.~~ The Resource Manager service is available through the OSGi service registry.

This service holds the existing Resource Context instances. Resource Context instances are created by calling the createContext(String, ResourceContext) method. The caller provides a context name as a string and optionally a template as a ResourceContext object.

The list of existing Resource Context instances can be retrieved through the following methods:

- getContext(String) – retrieve a Resource Context instance by name.
- getContext(Thread) – retrieve the Resource Context instance related to a Thread.
- getCurrentContext() - retrieve the Resource Context instance based on the current thread. ~~This method is equivalent to getContext(Thread.currentThread()) if context switching has not been used.~~
- listContexts() - retrieve all existing Resource Context instances as an array.

The Resource Manager singleton manages the persistence of the Resource Context instances. The following properties are stored:

- name of the Resource Context.
- list of the bundles belonging to the Resource Context.

- list of the Resource Monitor instances. For each one:
  - sampling period.
  - monitoring period.

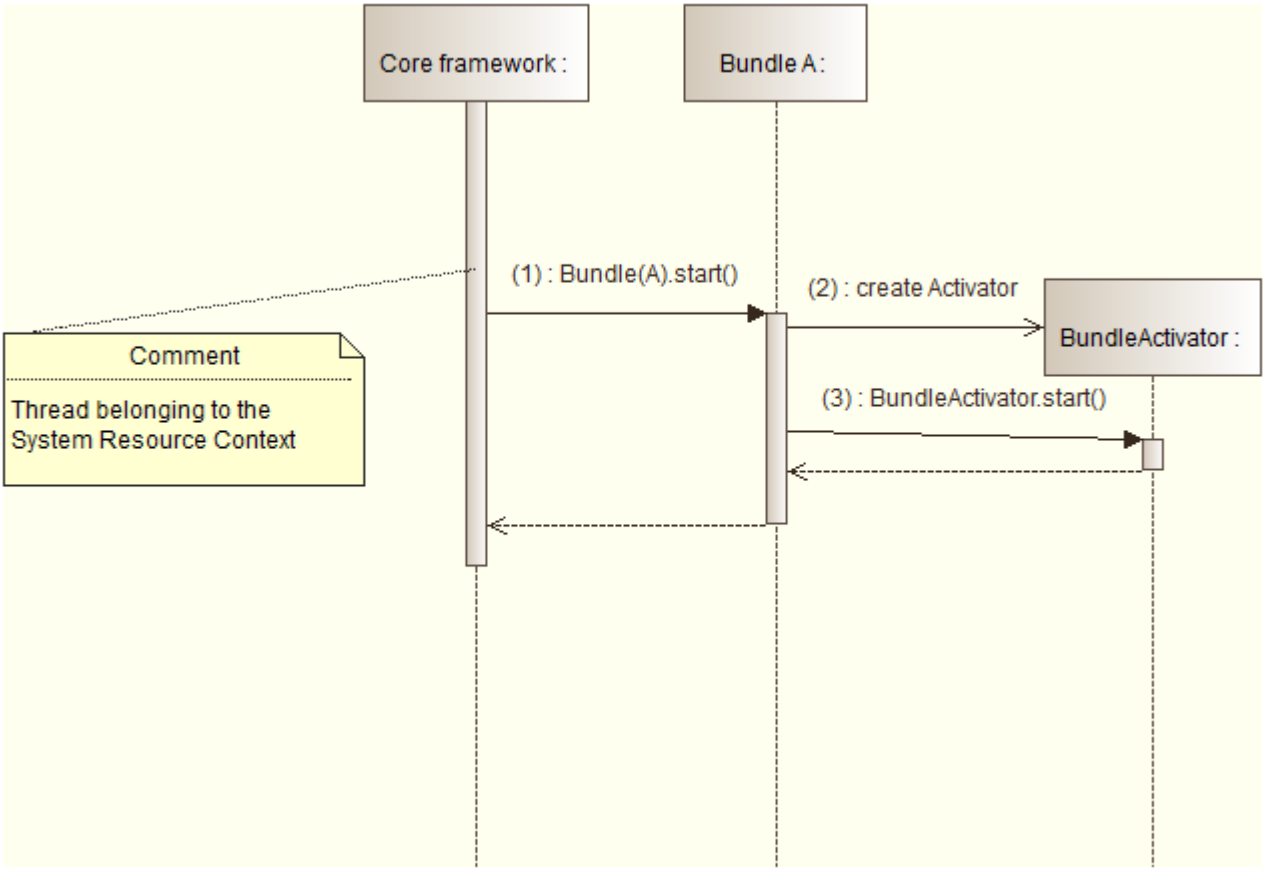
The way the Resource Manager persists the Resource Context instances is implementation specific. The implementer is free to use any file format and file location it wants. At startup, the Resource Manager will load the persisted Resource Context instances to restore shutdown state.

~~The Resource Manager service also allows to perform Resource Context switching. This feature may rely on the association of Java threads with a Resource Context instance. In that case, every Java thread is associated with a single Resource Context instance. A Resource Context switching operation then reallocates a Java thread to another Resource Context instance. Therefore, all further resource allocation made during the execution of the Java thread are accounted to the new Resource Context instance.~~

~~This switching feature is executed by a call to the switchContext(Thread, ResourceContext) or switchCurrentContext(ResourceContext).~~

~~Below are described three use cases showing an appropriate usage of the context switching feature.~~

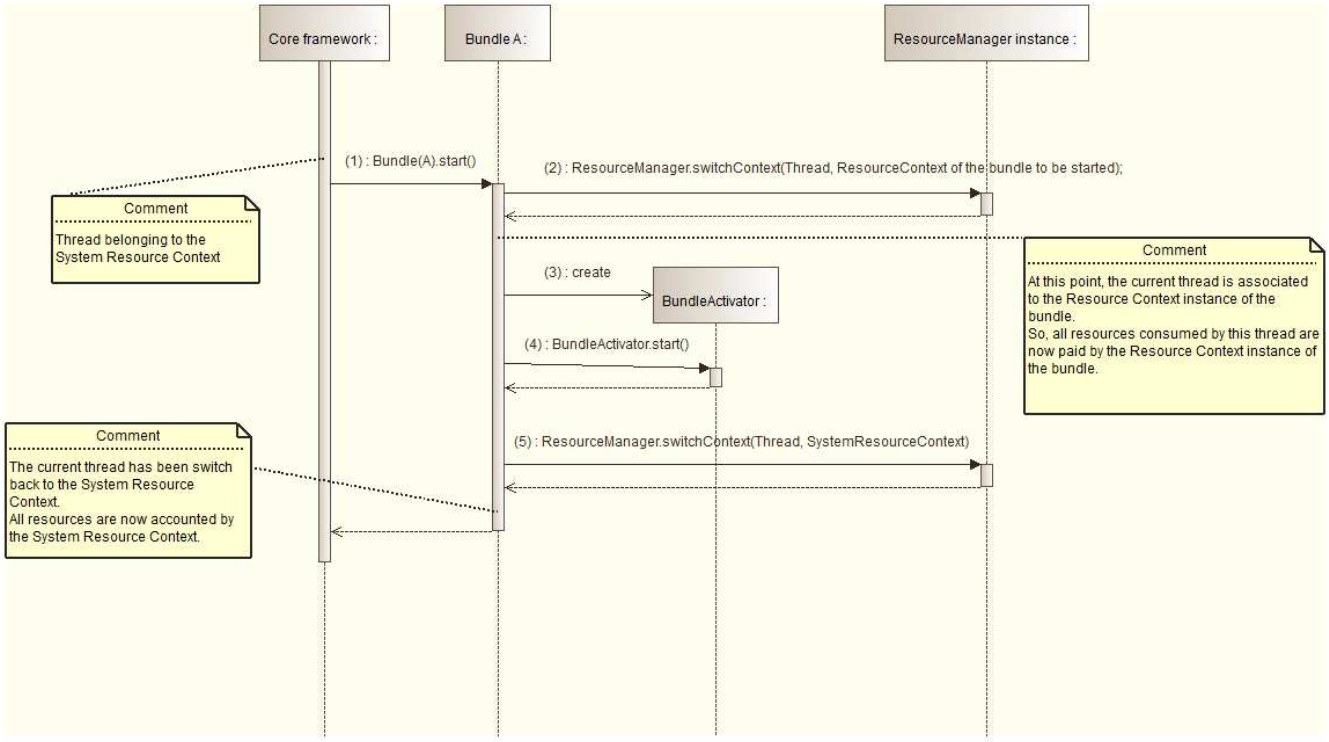
~~When the platform is starting, the core framework may start automatically bundles. The next sequence diagram describes the actions performed to start a bundle:~~



~~For every bundle instance to be started, the core framework calls Bundle.start() method (1). Bundle.start() method then creates a new instance of the BundleActivator implementation class of the Bundle (2) and calls BundleActivator.start() method (3). BundleActivator.start() method is generally used to allocate all resources a bundle needs, start threads and get and/or register services.~~

The object starting the bundle (actually the thread calling Bundle.start() method) belongs to the System Resource Context. As a consequence, resources consumed during start() call, including the resources used for the activation of the bundle, are accounted by the System Resource Context.

This situation may not be suitable. In usual situations, these resources may be accounted by the Resource Context instance of the bundle. A context switching operation has thus to be performed to switch to the Resource context instance of the bundle. The next diagram summarizes the actions to perform:



The ResourceManager.switchContext() operation (action 2) switches to the Resource Context of the bundle. At this point, all resources consumed by the current thread are accounted by the Resource Context instance of the bundle. Then, a new BundleActivator instance is created (action 3) and the BundleActivator.start() method is called (action 4). BundleActivator.start() method allocates all resources needed by the bundle. Before the completion of the Bundle.start() method, a context operation is again executed to switch back to the System Resource Context (action 5).

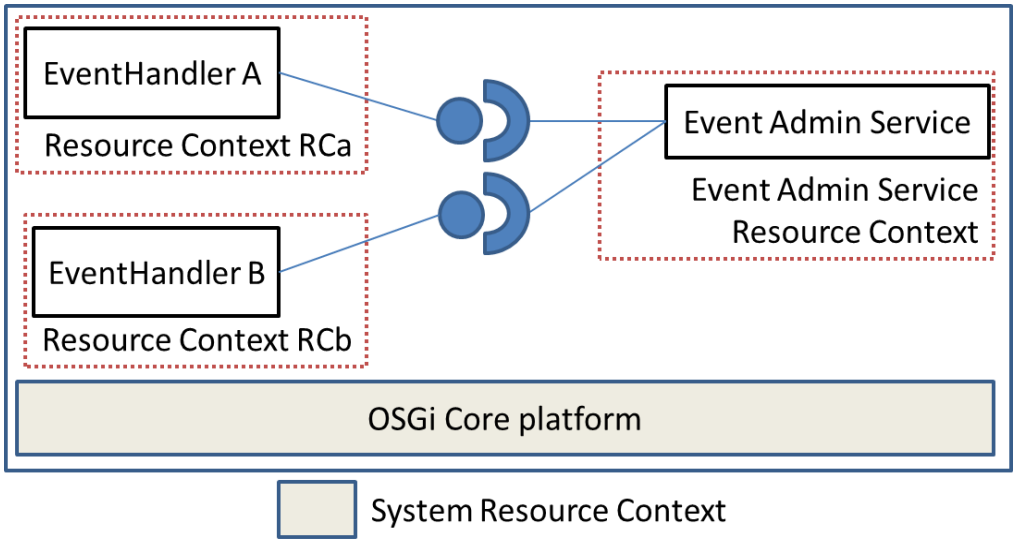
It is important to note that a context switching operation must be executed twice. The first context switching sets the context to which resources must be accounted. The second switches back to the initial context. The developer MUST ensure that these two operations are correctly achieved in order to avoid resource miscellaneous accounting.

This use case clearly shows that a context switching operation is needed at bundle startup to account resource in the appropriate Resource Context instance. Other framework operations requires a context switching execution:

- BundleActivator.start() - switch to the to-be-activated bundle Resource Context instance.
- BundleActivator.stop() - switch to the to-be-stopped bundle Resource Context instance.
- ServiceListener.serviceChanged() - switch to the Resource Context instance of the bundle hosting the ServiceListener instance.
- BundleListener.bundleChanged() - switch to the Resource Context instance of the bundle hosting the BundleListener instance.

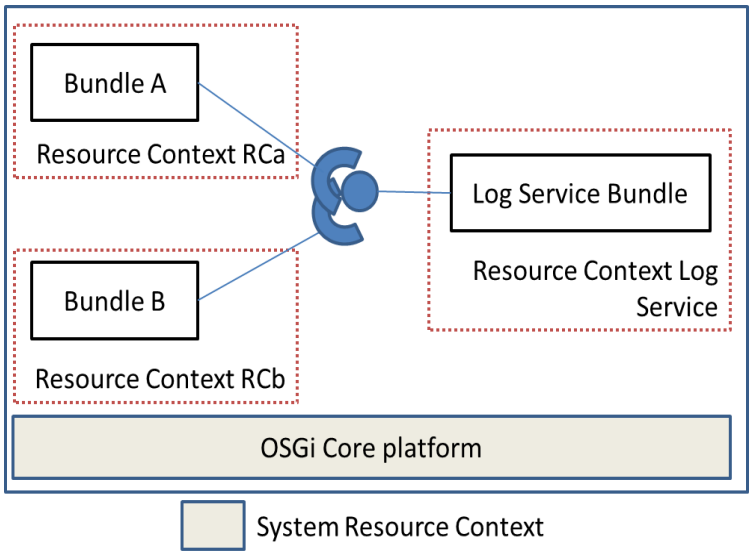
- ~~FrameworkListener.frameworkEvent() - switch to the Resource Context instance of the bundle hosting the FrameworkListener instance.~~
- ~~ServiceFactory.getService() - switch to the Resource Context instance of the bundle requesting the OSGi service.~~
- ~~ServiceFactory.ungetService() - switch to the Resource Context instance of the bundle calling ungetService.~~

~~Some other Compendium OSGi service like Http Service or Event Admin service should also take advantage of the context switching operation. The next diagram shows the use case for the Event Admin service:~~



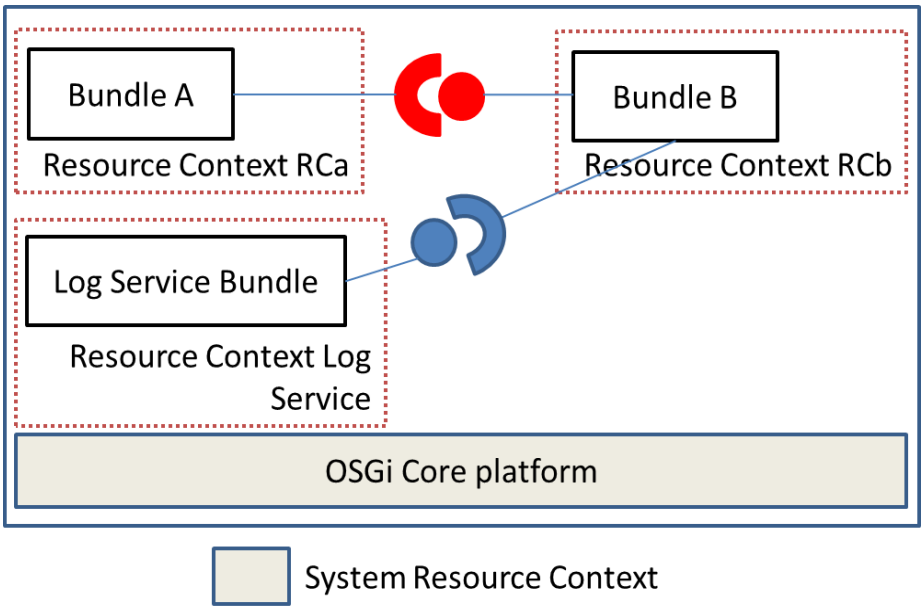
~~The EventAdmin Service should execute a context switching operation before and after calling EventHandler.handleEvent(event). This operation switches the current context to the context of the bundle providing the Event handler. Then, the resources consumed by the EventHandler,handleEvent() method are accounted by the Resource Context associated to the bundle providing the Event Handler and are not paid by the Event Admin service.~~

~~Now, the LogService case is considered. Any bundle about to log requires the LogService instance and calls LogService.log() method. The next schema shows two bundles A and B belonging respectively to Resource Context RCa and RCb. These two bundles requires the LogService (which belongs to Resource Context Log Service):~~



~~In such situation, the resources used by the LogService.log() method due to the bundle A are paid by RCa (the Resource Context of A). So, if bundle A overuses the LogService, it may exceeds the resource quotas of RCa which may lead to stop the bundle A (for example). Bundle B is not impacted and can still use the LogService.~~

~~Now, the following example is considered:~~



~~Bundle A belongs to the Resource Context named RCa. It requires the Red service provided by Bundle B. The Red service provides a single public method called m(). Red.m() method uses the LogService to log useful data. Bundle B belongs to Resource Context RCb, LogService bundle belongs to Resource Context Log Service.~~

~~When bundle A calls Red.m() method, all the resources consumed by Red.m() method are paid by Rca (if, for instance, the owner is defined as the owner of the current thread). As Red.m() calls also the LogService.log() method, the resource consumed by LogService.log() method are also paid by RCa.~~

Now, if the Red.m() method executes a context switching operation to switch to RCb (resource context instance of bundle B), then all resources used by Red.m() method are paid by RCb including the resources used by the call to the LogService.log() method. In this case, a call to Red.m() method costs nothing to RCa. However, Red.m() method MUST switches back to RCa at the end of its execution. If it is not the case, all resources consumed afterwards could be still paid by RCb. One more time, it is very important for a method that need to execute a switch context operation to switch back to the initial context at the end of its execution.

The three use cases described above show that context switching operation are not suitable in all cases. The most important thing is finally to determine which is the context executing the code. In all cases, a switching context operation notify all ResourceMonitor instances of both the outgoing ResourceContext and the incoming ResourceContext. The ResourceMonitor instances of the leaving ResourceContext are notified through ResourceMonitor.notifyOutgoingThread(thread) where the ones of the incoming ResourceContext are notified through ResourceMonitor.notifyIncomingThread(thread).

Resource Management Authority

A Resource Management Authority uses the Resource Manager singleton instance to apply Resource Management policies. These entities MAY:

- create and configure Resource Context instances (resource thresholds, bundle scope)perform some context switching
-
- take any decisions (stop a bundle, uninstall a bundle) if a Resource Context exceeds resource limit.

These policies are out of the scope of this specification.

6 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

7 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

Resource Manager inside the Core framework or in a bundle?

Some framework operations like Bundle.start() or Bundle.stop() requires a context switching to account resource usage in the context of the related bundle (instead of using the context of the caller of the method). These context switching have to be done automatically by the framework. As a consequence, the Resource Manager solution must be implemented inside the core framework. However, the Resource Manager may take advantage of the extension bundle mechanism.

Adapt pattern or OSGi service?

Each bundle is belonging to one specific resource context. So, the Bundle.adapt() method is an easy to use way to get access to its related ResourceContext object. It also avoid service management code necessary to require and release a service reference.

Moreover, as the Core framework has to be modified in order to perform automatic context switching on specific framework methods, the adapt pattern is definitely the best approach.

03/22/2013: Evgeni (Prosyst) seems not to be 100% sure of the adapt pattern. He indicates that the service approach could be better. Then the discussion moves the reasons why the Start Level Service uses the adapt pattern.

08/20/2013: David Bosschaert proposes to remove the adapt pattern for retrieving Resource Manager and Resource Contexts. From the Enterprise Expert Group point of view, the Resource Manager capability may be implemented outside the OSGi core framework. Introducing the adapt pattern supposes to be a part of the OSGi core framework and implies a high coupling between the Resource Management solution and the core framework. David proposes to use the service mechanism to make available the Resource Manager instance. Then from this Resource Manager service, Resource Context can easily be retrieved.

Eventing paradigms

03/22/2013: Several eventing mechanism have been discussed:

- The Event Admin service. This is the logical service to send notifications on OSGi platform. However, this service is optional and may not be accessible all the time. What to do in these cases?
- Use of the core eventing system:
  - Notifications through BundleEvent objects. Notifications MUST be sent when a Resource Context instance exceeds one of its resource usage thresholds. As a Resource Context is not limited to a single bundle (it could be a set of bundle), the BundleEvent approach seems not to be a good approach.
  - Notifications through FrameworkEvent objects. Those events have been designed to notify about general events of the OSGi environment (e.g., The framework has started,...). FrameworkEvent instances contains a few fields like the bundle associated to the event, the exception causing this event and a type. Moreover, it is not possible to define filters when registering the FrameworkListener.
  - Notifications through a new ResourceEvent object. To be defined.
- Whiteboard pattern approach. Some ResourceListener instance (to be defined) may be registered as an OSGi service or any other pattern. Each time a ResourceMonitor instance detects a resource usage exceeding, it requests for the list of existing ResourceListener instances and notify them. Some filters may be applied in order to reduce the number of ResourceListener to be notified.

Resource event classes

04/03/2013: Resource Event instances notify applications about threshold exceeding as well as Resource Management configuration updates (e.g., adding/removing Resource Context instance, adding/removing bundles

from Resource Context instance). A Resource Listener instance receiving a notification then has to identify which kind of notification it receives (through a call to ResourceEvent.getType()) and treat it accordingly.

So it could be interesting to separate concerns by creating two different Event/Listener interfaces. Resource Event and Resource Listener can still deal with threshold exceeding while Resource Context Event and Resource Context Listener will manage any events about Resource Context configuration updates (adding or removing Resource Context, adding or removing bundles).

CPU Monitor

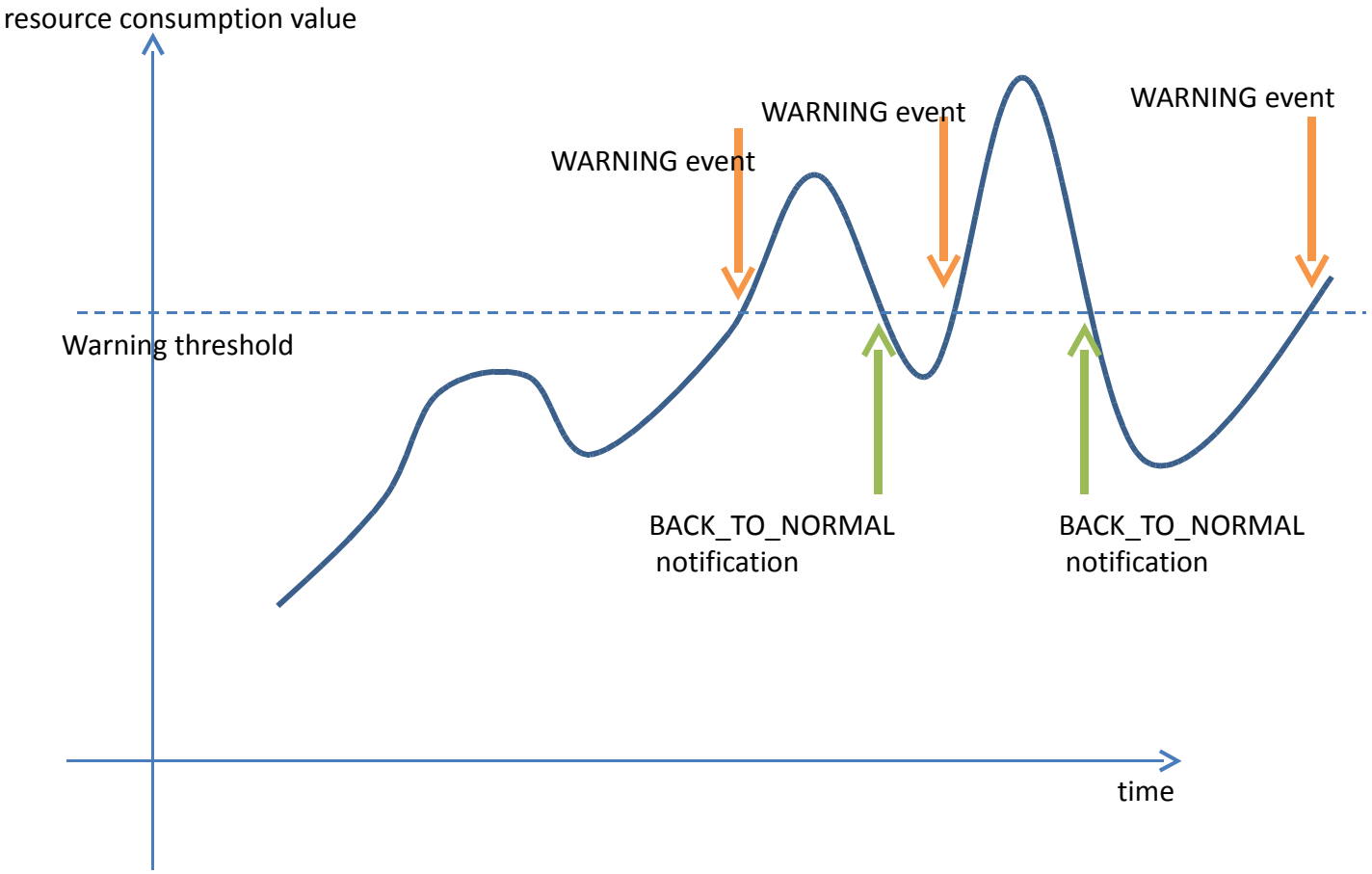
CPU Monitor instances monitor the CPU usage of Resource Context. The CPU usage is expressed as a percentage of usage over the monitoring period.

This percentage may be evaluated using the raw cpu data like the number of nanoseconds a Resource Context uses CPU. CPU Monitor instance evaluates the percentage by making a difference between cumulative values periodically retrieved (sampling period).

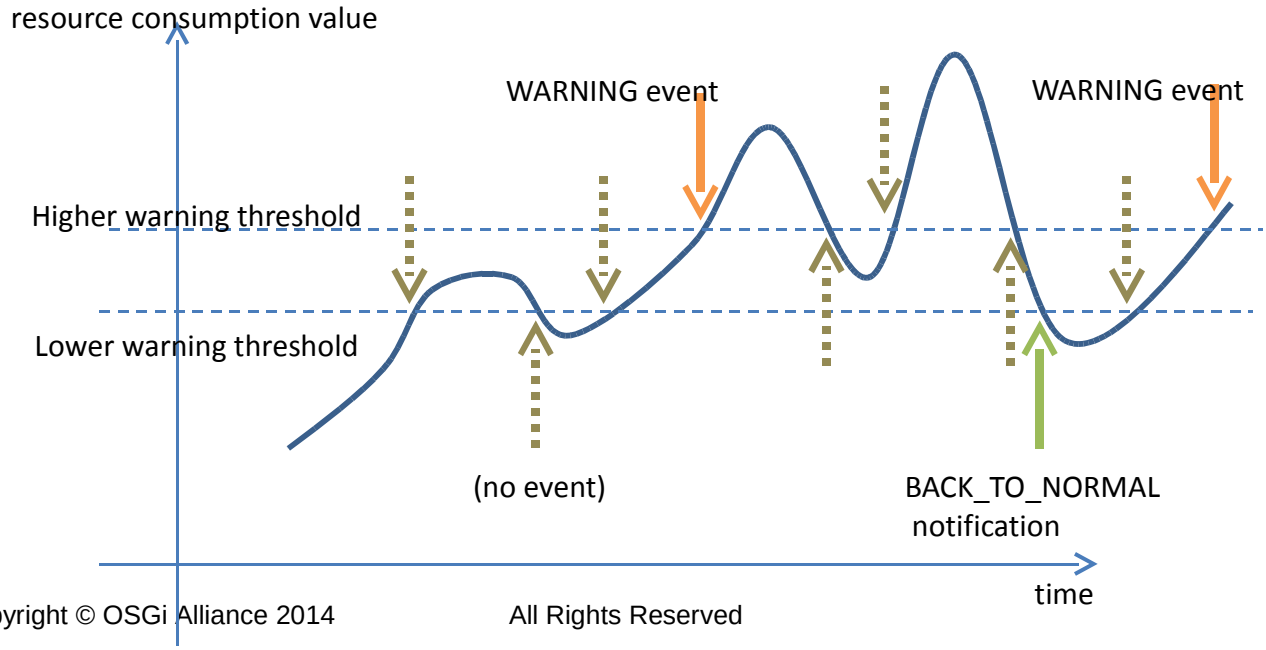
It could be interested to leave access to the CPU cumulative values. A getCumulativeUsage() method may be added to the CPU Monitor interface.

Resource Threshold algorithm and eventing

When the resource consumption exceeds the WARNING threshold, the Resource Threshold instance goes from the NORMAL state to the WARNING state and generates WARNING Resource Event. When the resource consumption decreases below the WARNING threshold, the Resource Threshold instance goes back to the NORMAL state and generates a NORMAL Resource Event notification. In some cases, the resource consumption may fluctuate around the threshold generating several WARNING or NORMAL Resource Event notification. The next chart summarizes the situation:



NTT proposes the Resource Threshold instances are not a fixed straight value but rather a floating value with a lower warning threshold and a higher warning threshold (some kind of range). The NTT's solution is summarized below:



WARNING Resource event are generated only when higher warning threshold is exceeded. The same for NORMAL Resource Event when the resource consumption decreases under the lower warning threshold.

This kind of algorithms decreases the number of Resource Event notifications but increases the implementation complexity.

The RFC does not take any strong position on that particular question. Resource Management solution providers are free to implement such algorithms.

Resource Monitor Factory

ResourceMonitor instances are created by ResourceMonitorFactory. Some ResourceMonitor implementation (e.g., those provided by the framework itself) might not have ResourceMonitorFactory. This is strongly encouraged to provide Resource Monitor Factory instance for all kind of Resource Monitor.

Resource accounting and context switching

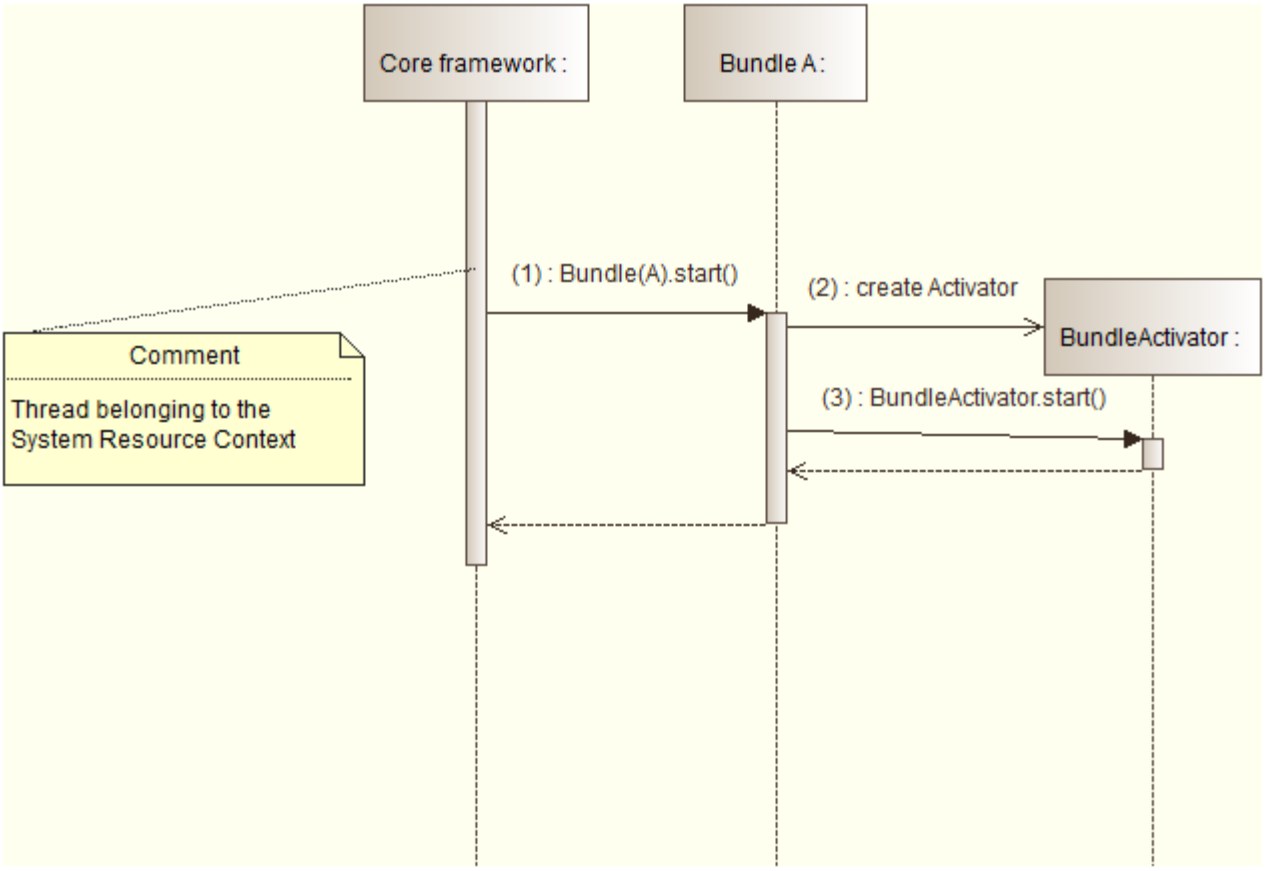
In a previous version, the Resource Manager service also allowed to perform Resource Context switching. The switching feature relied on a specific design for resource monitoring and accounting. For generality, it was decided to remove this feature (after CPEG review-).

This feature relies on the association of Java threads with a Resource Context instance. In that case, every Java thread is associated with a single Resource Context instance. A Resource Context switching operation then reallocates a Java thread to another Resource Context instance. Therefore, all further resource allocation made during the execution of the Java thread are accounted to the new Resource Context instance.

This switching feature is executed by a call to the switchContext(Thread, ResourceContext) or switchCurrentContext(ResourceContext). Resource Monitors are notified when a context switching operation occurs. When a thread joins a ResourceContext, all ResourceMonitors of the incoming ResourceContext are notified through a call to ResourceMonitor.notifyIncomingThread(thread). On the opposite, when a thread leaves a ResourceContext, all ResourceMonitors of the outgoing ResourceContext are notified through a call to ResourceMonitor.notifyOutgoingThread(thread).

Below are described three use cases showing an appropriate usage of the context switching feature.

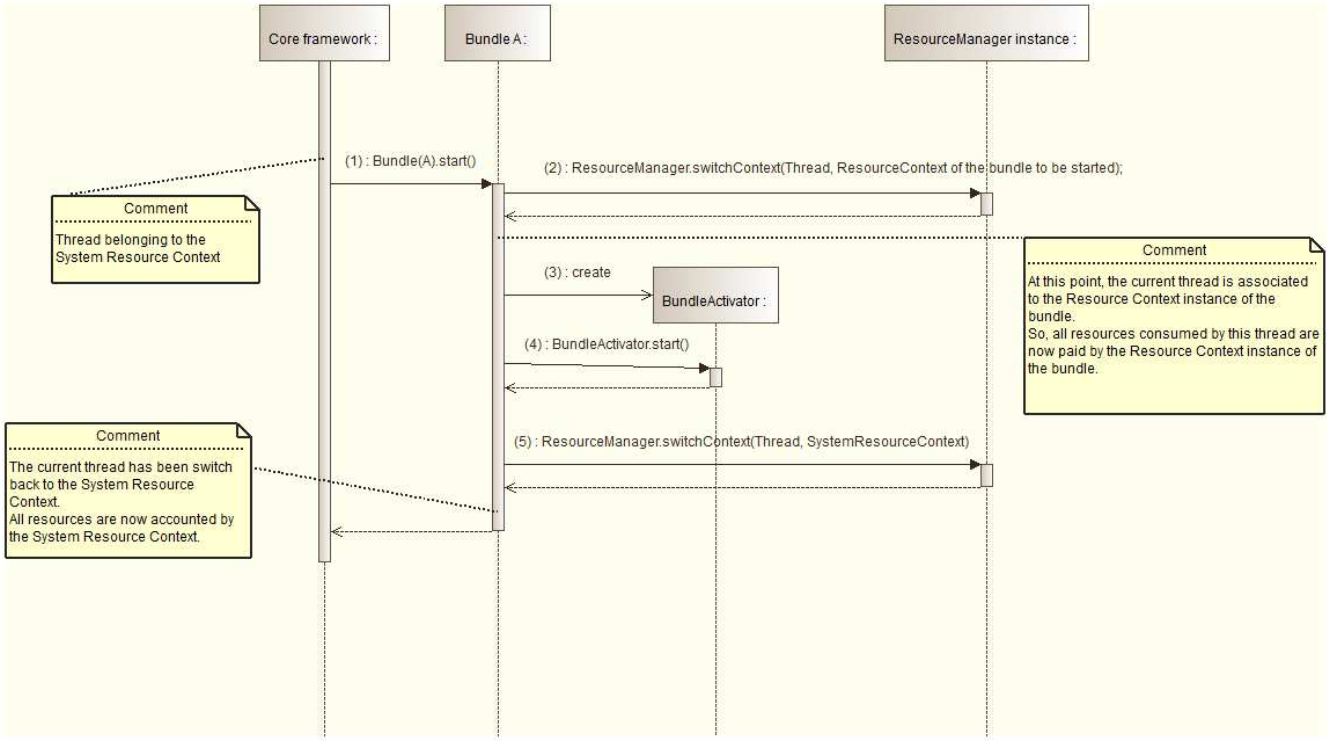
When the platform is starting, the core framework may start automatically bundles. The next sequence diagram describes the actions performed to start a bundle:



For every bundle instance to be started, the core framework calls Bundle.start() method (1). Bundle.start() method then creates a new instance of the BundleActivator implementation class of the Bundle (2) and calls BundleActivator.start() method (3). BundleActivator.start() method is generally used to allocate all resources a bundle needs, start threads and get and/or register services.

The object starting the bundle (actually the thread calling Bundle.start() method) belongs to the System Resource Context. As a consequence, resources consumed during start() call, including the resources used for the activation of the bundle, are accounted by the System Resource Context.

This situation may not be suitable. In usual situations, these resources may be accounted by the Resource Context instance of the bundle. A context switching operation has thus to be performed to switch to the Resource context instance of the bundle. The next diagram summarizes the actions to perform:



The ResourceManager.switchContext() operation (action 2) switches to the Resource Context of the bundle. At this point, all resources consumed by the current thread are accounted by the Resource Context instance of the bundle. Then, a new BundleActivator instance is created (action 3) and the BundleActivator.start() method is called (action 4). BundleActivator.start() method allocates all resources needed by the bundle. Before the completion of the Bundle.start() method, a context operation is again executed to switch back to the System Resource Context (action 5).

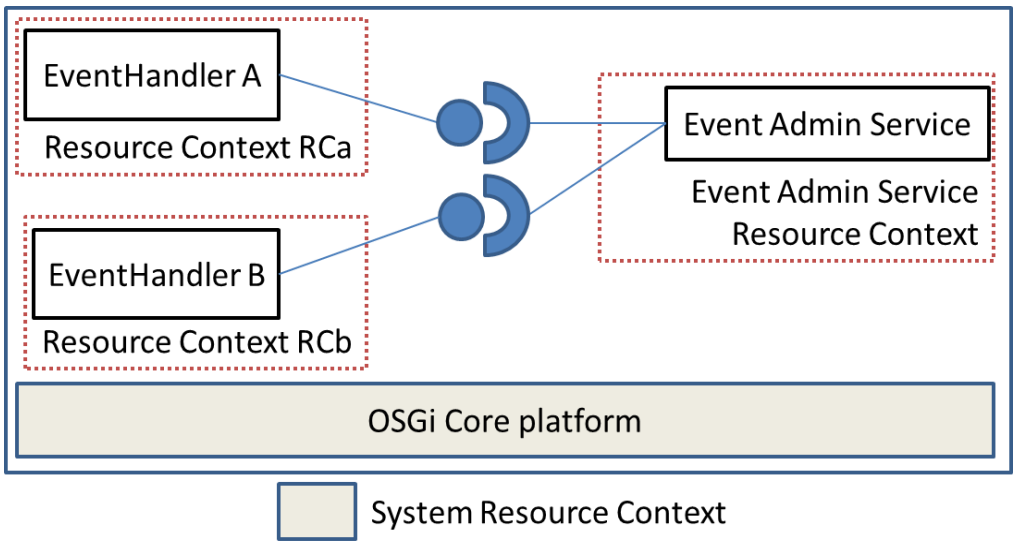
It is important to note that a context switching operation must be executed twice. The first context switching sets the context to which resources must be accounted. The second switches back to the initial context. The developer MUST ensure that these two operations are correctly achieved in order to avoid resource miscellaneous accounting.

This use case clearly shows that a context switching operation is needed at bundle startup to account resource in the appropriate Resource Context instance. Other framework operations requires a context switching execution:

- BundleActivator.start() - switch to the to-be-activated bundle Resource Context instance.
- BundleActivator.stop() - switch to the to-be-stopped bundle Resource Context instance.
- ServiceListener.serviceChanged() - switch to the Resource Context instance of the bundle hosting the ServiceListener instance.
- BundleListener.bundleChanged() - switch to the Resource Context instance of the bundle hosting the BundleListener instance.
- FrameworkListener.frameworkEvent() - switch to the Resource Context instance of the bundle hosting the FrameworkListener instance.
- ServiceFactory.getService() - switch to the Resource Context instance of the bundle requesting the OSGi service.

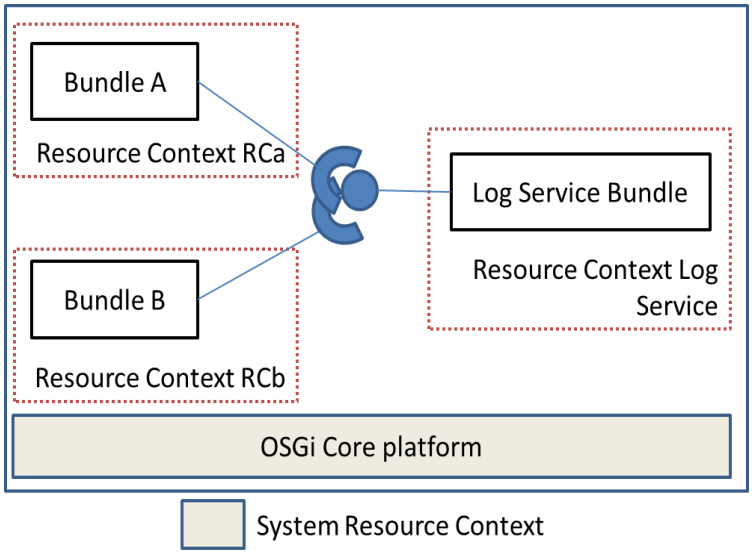
- ServiceFactory.ungetService() - switch to the Resource Context instance of the bundle calling ungetService.

Some other Compendium OSGi service like Http Service or Event Admin service should also take advantage of the context switching operation. The next diagram shows the use case for the Event Admin service:



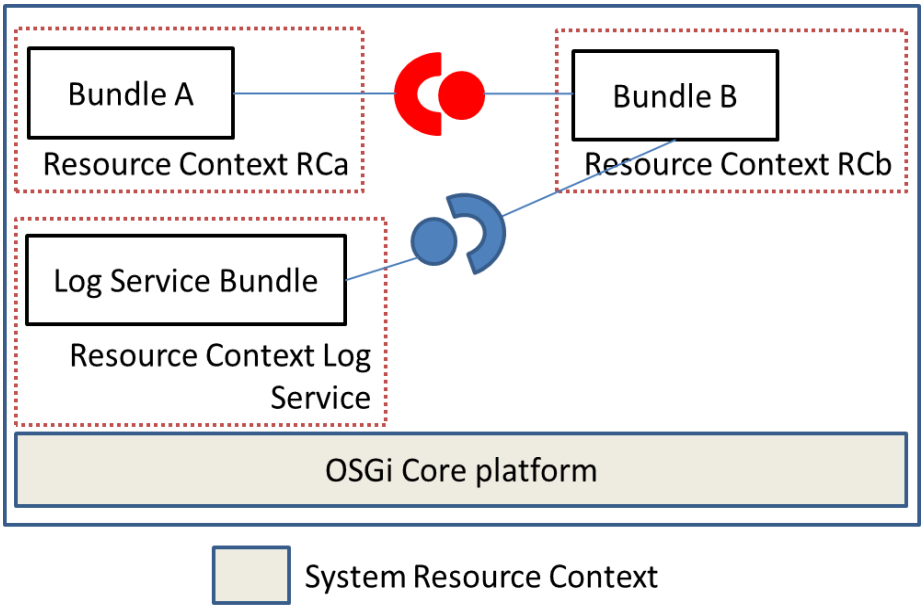
The EventAdmin Service should execute a context switching operation before and after calling EventHandler.handleEvent(event). This operation switches the current context to the context of the bundle providing the Event handler. Then, the resources consumed by the EventHandler,handleEvent() method are accounted by the Resource Context associated to the bundle providing the Event Handler and are not paid by the Event Admin service.

Now, the LogService case is considered. Any bundle about to log requires the LogService instance and calls LogService.log() method. The next schema shows two bundles A and B belonging respectively to Resource Context RCa and RCb. These two bundles requires the LogService (which belongs to Resource Context Log Service):



In such situation, the resources used by the LogService.log() method due to the bundle A are paid by RCa (the Resource Context of A). So, if bundle A overuses the LogService, it may exceeds the resource quotas of RCa which may lead to stop the bundle A (for example). Bundle B is not impacted and can still use the LogService.

Now, the following example is considered:



Bundle A belongs to the Resource Context named RCa. It requires the Red service provided by Bundle B. The Red service provides a single public method called m(). Red.m() method uses the LogService to log useful data. Bundle B belongs to Resource Context RCb, LogService bundle belongs to Resource Context Log Service.

When bundle A calls Red.m() method, all the resources consumed by Red.m() method are paid by Rca (if, for instance, the owner is defined as the owner of the current thread). As Red.m() calls also the LogService.log() method, the resource consumed by LogService.log() method are also paid by RCa.

Now, if the Red.m() method executes a context switching operation to switch to RCb (resource context instance of bundle B), then all resources used by Red.m() method are paid by RCb including the resources used by the call to the LogService.log() method. In this case, a call to Red.m() method costs nothing to RCa. However, Red.m() method MUST switches back to RCa at the end of its execution. If it is not the case, all resources consumed afterwards could be still paid by RCb. One more time, it is very important for a method that need to execute a switch context operation to switch back to the initial context at the end of its execution.

The three use cases described above show that context switching operation are not suitable in all cases. The most important thing is finally to determine which is the context executing the code. In all cases, a switching context operation notify all ResourceMonitor instances of both the outgoing ResourceContext and the incoming ResourceContext. The ResourceMonitor instances of the leaving ResourceContext are notified through ResourceMonitor.notifyOutgoingThread(thread) where the ones of the incoming ResourceContext are notified through ResourceMonitor.notifyIncomingThread(thread).

Compatibility between bundles capabilities

Some bundle implementation should naturally use the Resource Management features and in particular the switching context operation in order to account resource properly. For example, the Http Service implementation switch to the Context of the bundle providing the servlet before executing any service methods when receiving an HTTP request.

An open issue is the deployment of resource management non-aware bundles into a resource management aware context. In that particular case, some Resource Context may not be charged correctly.

The present specification gives all the features necessary to implement bundles and framework resource management aware. As a consequence, the platform operator is free to deploy suitable bundles depending on its needs (i.e., in a resource management aware context or not).

Implementation of resource management aware bundles

There are different ways to implement a resource management aware bundle.

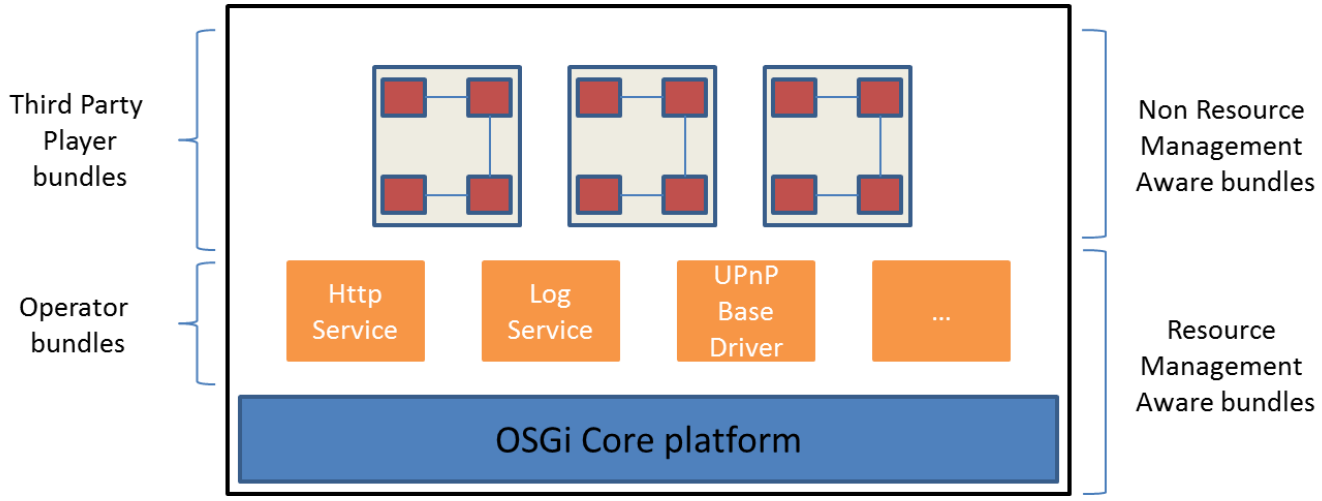
The first approach is the direct implementation into the bundle. The bundle implementer is free to execute context switching operation when it is required. The bundle implementer is fully responsible to choose the appropriate Resource Context to charge and it is also responsible to switch back to the initial Resource Context. This approach has some drawbacks. First, it implies to handle Resource Management features into a business features which is sometimes not suitable. Moreover, the platform operator must be confident in the way the bundle implementer uses context switching operations (i.e., permissions). Finally, as the context switching operations are made by the bundle itself, the chosen accounting policy is statically defined. This accounting policy may not be convenient in all situations.

The second approach is to take advantage of the weaving feature. The weaving feature allows to inject some byte-code into existing bundles. In this case, the injected byte-code deals with the context switching features. The byte-code injection can be done at runtime or at the compilation time. The advantage of this approach is that the bundle implementer does not need to handle any context switching feature. Moreover, the accounting policy can be changed with a new byte-code injection. The main drawback of this approach is the complexity and acceptability of the use of bytecode injection.

Finally, the last approach is to implement service proxies. This kind of service proxyfies a service which is not resource management aware. The service proxy implements the same interface as the one of the to-be-proxified service and handles all the resource management stuff. This approach is very similar to the weaving approach.

Example 1

Operators involved into the Residential market plan to provide a Resource Management aware OSGi platform on which Third Party Players will deploy bundles:



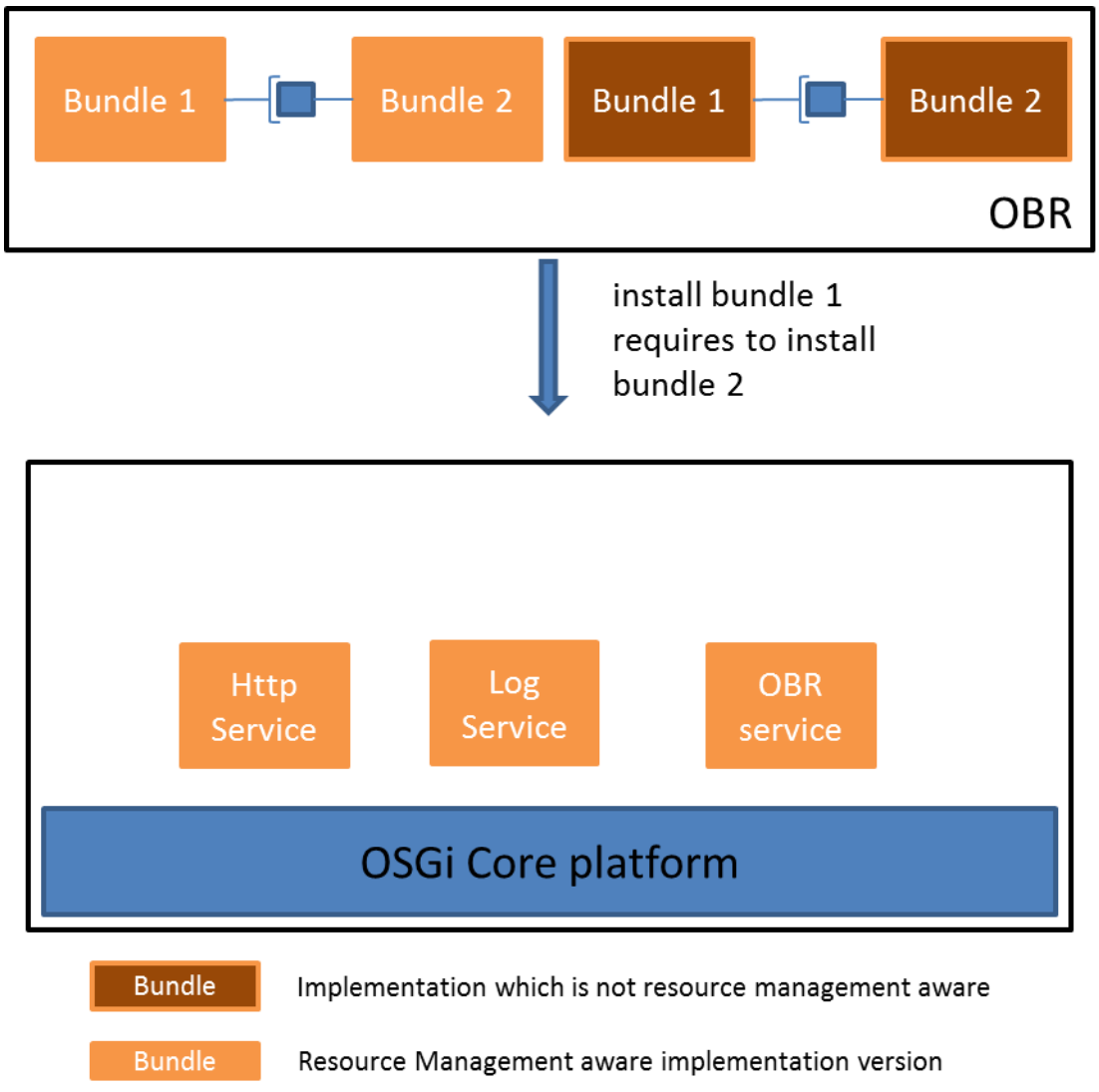
The OSGi platform hosts some operator bundles like HttpService or LogService. All of them are resource management aware, i.e., they automatically execute switch context operation in order to charge resource to the appropriate Resource Context.

All these operator services are used by Third Party Player bundles. Each Third Party Player is isolated from another (they do not share services between them) and the platform operator creates a Resource Context per Third Party Player. Moreover, as Third Party Players may not be trusted, their bundles do not have access to the Resource Management API. This does not prevent the resources to be accounted to the Resource Context belonging to the Third Party Player.

In such situations, Resource Management is transparent for the Third Party Player. Authorities and operator bundles handle resource context switching at the appropriate moment.

Example 2

In the Enterprise context, bundles are deployed using the OBR service. The OBR allows to deploy bundles and all their dependencies computed based on package and service dependencies of to be-installed bundle:



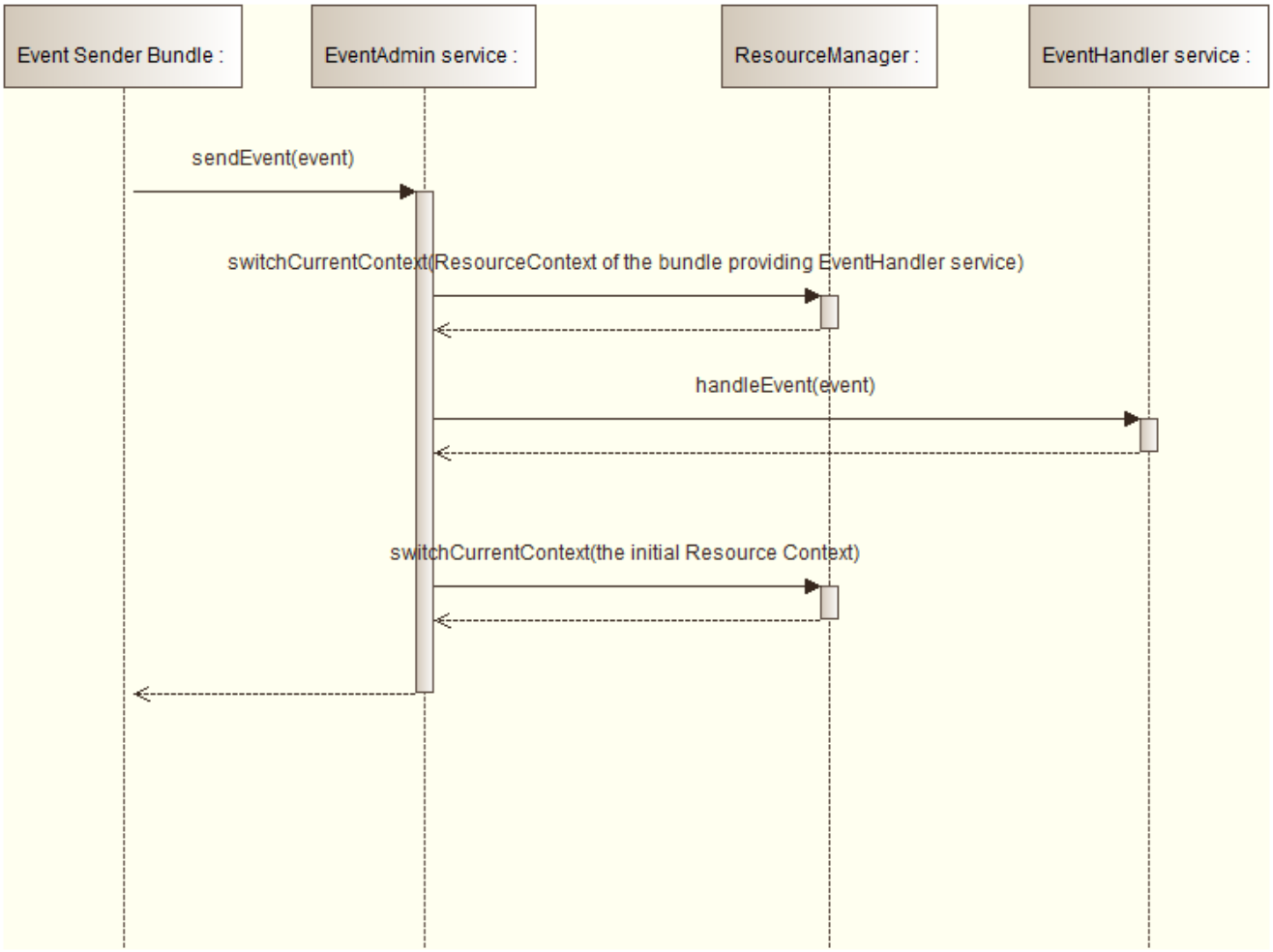
Here, the OBR service is requested to install the bundle 1 which depends on a package provided by bundle 2 (which is not installed on the platform). The OBR then installs the bundle 1 as well as the bundle 2 in order to satisfy the missing dependency.

In the case where the OSGi Core platform is resource management aware, the operator may have to deploy bundles which should be resource management aware. So the OBR service should take care between a bundle implementation which is resource management aware (light orange box on the schema) and a one which is not (dark orange box).

In such cases, either the OBR service is smart to deploy the appropriate version or the operator should do it itself.

Event Admin use-case

The next sequence diagram shows the use-case of the Event Admin:

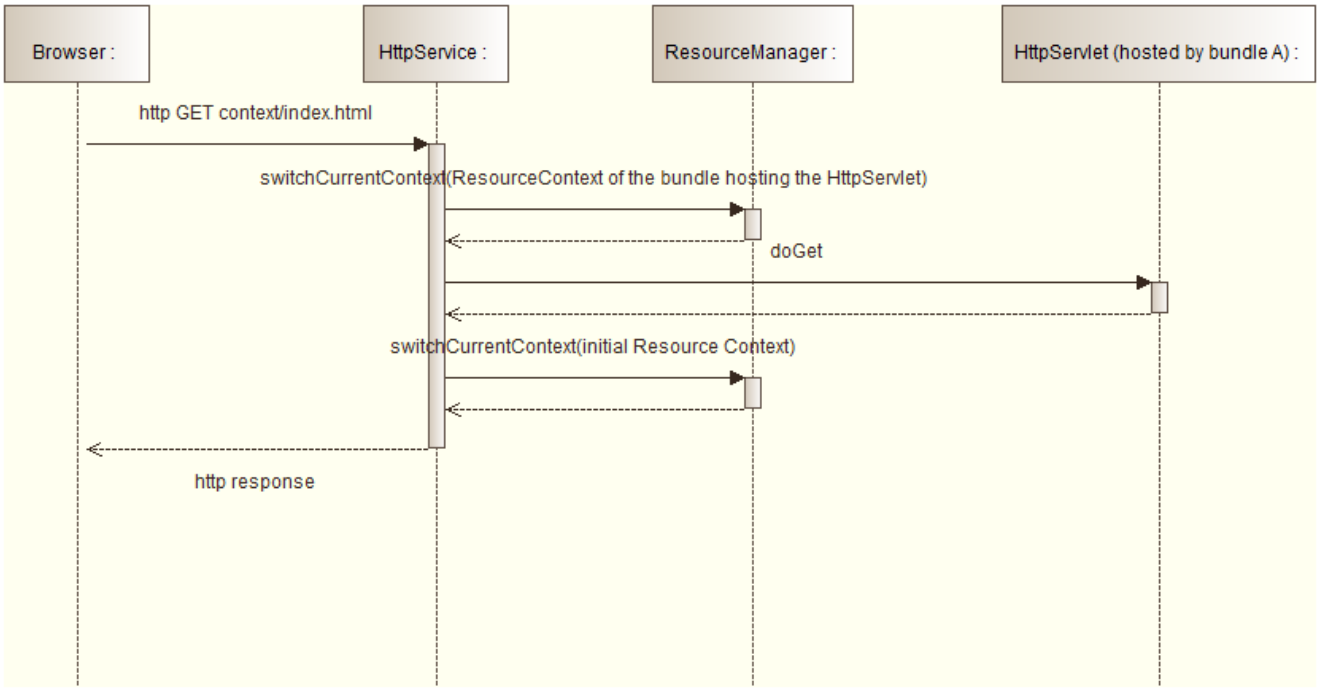


A bundle uses the EventAdmin.sendEvent() method to emit an event. The EventAdmin service then performs a lookup into the OSGi service registry to find out all the available EventHandler services. For every to-be-notified EventHandler, the EventAdmin performs a switching context operation to be in the Resource Context of the bundle providing the EventHandler service. Then it calls the EventHandler.handleEvent() method and switches back to the initial Resource Context (the one of the bundle sending the event).

The context switching operation made here allows to charge the bundle receiving the event instead of the bundle sending the event.

Http Service use-case

The Http Service use-case is shown below:



A browser requests an HTTP GET on context/index.html. The HttpService identifies the HttpServlet to be-invoked based on the context of the request. Once identified, the HttpService executes a switch context operation to the Resource Context of the bundle provided the HttpServlet. Then, the HttpService calls the HttpServlet.doGet() method and switches back to the initial context.

The switching context operation allows to charge the bundle providing the HttpServlet instead the bundle exposing the HttpService.

8 Security Considerations

~~*Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.*~~

9 Document Support

References

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

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

[3]. RFP 153 – Resource Management for OSGi Platform, OSGi Alliance

[4]. T. Miettinen, D. Pakkala, and M. Hongisto. A method for the resource monitoring of osgi-based software components. In Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference, pages 100 {107, 3-5 2008.

[5]. Y. Maurel, A. Bottaro, R. Kopetz, K. Attouchi. Adaptive Monitoring of End-user OSGi based Home Boxes. In Component Base Software Engineering, 15th ACM SigSoft International Symposium on Component-Based Software Engineering (CBSE 2012), Bertinoro, Italy, June 2012.

[6]. C. Larsson and C. Gray. Challenges of resource management in an OSGi environment. In OSGi Community Event 2011, Darmstadt, Germany, September 2011.

[7]. N. Geofray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a java virtual machine for component isolation in osgi. In Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, pages 544{553. IEEE, 2009.

[8]. Java Community Process, Java Specification Request 163, Java Platform Profiling Architecture, final release, September 2004.

[9]. *The Java Virtual Machine Specification*, Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3.

[10]. *The Java Language Specification, Third Edition*, May 2005, ISBN 0-321-24678-0.

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

Author's Address

Name	BONNARDEL Gregory
Company	France Telecom Orange
Address	28 Chemin du Vieux Chêne, 38240 Meylan, France
Voice	+33 4 76 76 44 49
E-mail	gbonnardel.ext@orange.com

Name	BOTTARO Andre
Company	France Telecom Orange
Address	28 Chemin du Vieux Chêne, 38240 Meylan, France
Voice	+33 4 76 76 41 03
e-mail	andre.bottaro@orange.com

Name	DIMOV Svetozar
Company	Prosyst
Address	
Voice	
e-mail	s.dimov@prosyst.com

Name	GRIGOROV Evgeni
Company	Prosyst
Address	
Voice	
e-mail	e.grigorov@prosyst.com

Name	RINQUIN Arnaud
Company	France Telecom Orange
Address	28 Chemin du Vieux Chêne, 38240 Meylan, France
Voice	+33 4 76 76 45 59
e-mail	arnaud.rinquin@orange.com

Name	CHAZALET Antonin
Company	France Telecom Orange
Address	28 Chemin du Vieux Chêne, 38240 Meylan, France
Voice	+33 4 76 76 41 03
e-mail	antonin.chazalet@orange.com

Acronyms and Abbreviations

End of Document