RFC 226 - Scheduling

Draft

12 Pages

Text in Red is here to help you. Delete it when you have followed the instructions.
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.

Abstract

10 point Arial Centered.

This RFC defines a time service that can provide time aspects like delays, timeouts, and periodic scheduling with the option to have cron like scheduling using the whiteboard pattern. The solution takes advantage of the OSGi Promise API and the date-time API introduced in Java 8.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

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

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

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

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

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

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

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

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

0.2 Trademarks

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

0.3 Feedback

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

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

```
Source code is shown in this typeface.
```

0.6 Revision History

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

Revision	Date	Comments
Initial	May 2016	David Bosschaert – initial version based on RFP 166

1 Introduction

This RFC originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

"*The only reason for time is so that everything doesn't happen at once.*" as Albert Einstein said. Though we are trying to prove Einstein wrong by adding more and more cores to our CPUs it is clear that time is an uncomfortable citizen in our software world. Simple concepts like before-after relations are surprisingly hard to work with, trying to schedule a number of actions in time, or just trying to work with dates and time in the light of the myriad of calendars and timezones further highlight the complexity of time. The primary reason of this complexity is that software has no built-in concept of time. We do not have variables that depreciate or easy primitives to spread out an operation over time (at least after we abandoned the delay loop).

The result is that there are a myriad of APIs in Java to handle delays, timeouts, and periodic scheduling. This RFP analyzes what there currently is and what is missing from an OSGi perspective.

2 Application Domain

2.1.1 Direction of Control

Time aspects of software are highly related with the *direction of control* between collaborating *actors*, where an actor is a piece of code. If an actor gets called it has no control over at what time it gets control. If the actor has control, it is expensive to delay with delay loop so the control must be temporarily handed over. Threads can be used for delaying inline because timed waits can be performed that look identical to a delay loop in the code but now the CPU could be busy with more useful work then executing NOPs.

A program that uses this threaded model is the easiest way to read and write programs since the *flow* of the code is the most concise possible in our current languages. However, the model does not scale well since threads are still expensive resources and the synchronous model requires one thread for each continuous flow. That said, a popular language like Erlang evolves around hundreds of thousands of (lightweight) threads that process messages by waiting for a message. The state of the actor is partly maintained by where the actor waits for the next message and the stack.

2.1.2 Push & Pull

A *producer* is an actor that has objects that it wants to convey to a *client*. Software allows us (surprisingly) only two options. The producer *pushes* the objects to the client or the client *pulls* the objects from the producer, the difference: time. For example, the producer can write (push) bytes to an Output Stream implemented by the client. Alternatively, the producer can implement an Input Stream and wait until the client reads (pulls) a byte. Though the same bytes are transferred, this direction of control has dramatic consequences for how we write our software as well as the runtime timing.

The more common software model in Java has been clients that pull since this model is closer to procedural languages, especially when they do not have lambdas. That is, we have iterators in Java. The client pulling allows the client to maintain the state on the call stack and procedural languages have fine grained highly optimized facilities for this model. For example, the Collections in Java support iterators that are based on the pull model: the client calls `next()` to get the next object.

2.1.3 Event Based Models

The increased complexity and distribution of applications (where the the flow timing becomes much more significant) is making the client-is-pushed model more popular despite its increased complexity. Since the client gets called whenever objects are available there is less waiting and it becomes easier to orchestrate multiple flows.

Java 8 significantly changed this landscape by introducing *lambdas*. Lambdas are objects that capture context and code, allowing the code to be passed around as an argument. Lambdas reduce the conceptual cost of using callback mechanisms because lambdas are lexically dramatically smaller then corresponding (albeit virtually identical in utility) inner classes. Java 8 also makes it easier to use the context of the lambda invocation by removing the need to mark all scoped variables as final (although it falls short of full closures).

Having lambdas made it possible to provide many push based clients in existing APIs. The `forEach` method on collections pushes the members of the collection to a lambda. Another example is the new Java 8 Stream library. It an interesting mix of a basic pull API combined with a push model that leverages lambdas. Streams are based on iterators, thus are pull, but the streams push the elements of the stream onto lambdas.

Though new Java 8 API uses a push model because that is very effective with lambda's, they did not allow an event based model where objects are not already present. For distributed applications the *event* model is more attractive where the asynchronously arriving events are pushed to the client.

2.1.4 Promises & Java RX

In Enterprise Edition Release 6, the OSGi Alliance introduced a Java *Promise*, based on the Javascript promises. A Promise removes any synchronicity between the producer and the client. The Promise represents a value that will arrive in the future or it will signal an error. Promises make it easier to sequence a number of steps in time without blocking. That is, a a function can define a multi-step sequence of actions, keep its state local to the function, handle errors that occur in any of the actions centrally, and still return immediately. That is, a Promise acts as a broker between a producer and a client to remove the synchronicity between them.

It is common that sequence of steps can have time outs (execute before) or require intermediate delays (do not execute before). Timeouts and delays can be implemented with intermediate promises that use a *scheduler*.

Promises are about a single return value. However, many problems require a (continuous) stream of events. Netflix promotes an open source project RXJava that provides push based stream model. The model uses a similar API as the Java 8 Stream API to process the objects but behind the scenes uses an event model to push objects into it.

2.1.5 Timers

Java has had a java.util.Timer class since the 1.0. Clients can create a Timer object, which creates a background thread, and then the client can register *tasks* with a *schedule*, a specification of when to *invoke* the task. A task is some object that can be executed, in this case a Timer Task. The tasks are then efficiently *scheduled* and invoked when they expire. An invocation is done in the scheduler thread. Long running tasks can therefore postpone the invocation of other tasks if previous tasks take a long time or the system is busy.

Tasks are *scheduled*. A schedule could be a *delay*. In that case the task was executed after the given *duration*. A duration is fixed length of time, for example 2 hours. Durations in the Timer are specified in a long representing milliseconds. A task could also be scheduled at a certain *instance time*, using a Date object. Instance time is defined in Java 8 as a number of nanoseconds from the *epoch*. The epoch is a fixed moment in time Jan 1 1970 0:0 UTC.

A schedule could also be *periodic*. A periodic schedule executes the Timer Task continuously with an *interval*. The interval is the duration between two invocations. Since Timer tasks take time to execute and there can be other causes for upholding the Timer Scheduler the actual interval can *skewed* and thus be longer than the given interval. Periodic schedules can provide an initial delay or a specific date for *initial scheduling*.

Periodic schedules can also be scheduled at a *fixed rate*. A fixed rate periodic schedule ensures that the interval is not skewed. Each invocation is scheduled at a multiple interval duration from the initial scheduling instance time.

Timer tasks could also be *canceled* by the clients. Since a task can be canceled at any moment in time, there is a potential race condition between the execution of the task and the cancelation. This in general requires the task to use locks or atomic booleans to verify executing against an expired context. Since a scheduled task has very little cost and the task can still be executed in an expired context a cancelation is not always opportune.

Since the Timer is a Java class, it includes the maintenance methods together with the collaboration methods. In general, each actor in an application tends to create their own timer and timers are generally not shared.

2.1.6 Java Util Concurrent

Java 5 introduced the java.util.concurrent package that provided a number of services to run tasks in a background thread and/or schedule at a given date-time or delay. The Scheduled Executor Service specifies a service that has the same facilities as the Timer class. However, durations are specified with a 2 parameters: a magnitude and a Time Unit and did not allow the initial scheduling to be scheduled at a date-time. It did provide background scheduling and any task invocations were executed away from the scheduler thread.

The Scheduled Executor Service is an interface. This interface contains the scheduling methods as well as life cycle and maintenance methods. The Executors class provides a number of standard implementations.

2.1.7 Quartz

The Timer and Scheduled Executor Service are limited to *instance* time. However, many tasks require scheduling based on dates and times. For example, some tasks must be executed every third Wednesday of the week or at 5 AM Sunday morning. The Linux cron jobs used a specific format to specify such a scheduling by creating a *mask* for seconds, minutes, hours, day of the week, day of the month, month, and year. When the mask matches, the given task is executed.

In the Java world, this facility was mainly provided by the Quartz library. It allows the specification of a cron like schedule and then executes predefined tasks. Tasks in this context are specified by their class name, they are then loaded and instantiated when needed.

2.1.8 Java 8 Date and Time

After having been shown the way by Joda Time Java 8 finally has a mature date and time library. The library provides now abstractions for local times and dates (i.e. 4 o'clock) that can be mapped to a specific instance time later when the day and location are known.

One relevant aspect of this library is the Temporal Adjuster interface. The Temporal Adjuster is a strategy for adjusting a *temporal* object. A temporal object is an instance time, a local date or time, or any other object that understands an actual time. *Adjusters* exist to externalize the adjusting. For example, an adjuster that sets the next date-time avoiding weekends, or one that sets the date to the last day of the month.

Practically, Temporal Adjusters can be used to abstract the concept of the delay, date, or periodic interval in the Timer/Scheduled Executor Service but they also encompass the cron mask.

2.2 Terminology + Abbreviations

3 Problem Description

Currently time is quite chaotic in Java. It is not clear what API to use to schedule tasks and for relatively simple task like cron scheduling it is necessary to escape to a significant library like Quartz.

- A major problem is that it is hard to share schedulers. One of the possible solutions would be to registers a Timer or Scheduled Executor Service in the service registry and then share this object between all components. However, both APIs include the life cycle and maintenance methods on their interfaces. In an OSGi environment, life cycle and interface must be clearly separated from the collaboration API.
- The existing Java Timer and Scheduled Executor Service also do not work with the very advanced Java 8 Date Time API. The Timer is limited to milliseconds or Date and the ScheduledExecutorService has the awkward kludge called TimeUnit.
- The Promise API lacks any time awareness.

Therefore, the solution this RFP seeks is a service that provides time support in the OSGi way leveraging Java 8 and our Promise API. This service should make it easy to handle timeouts, delays, and periodic schedules with very flexible interval specifications through a shared service.

4 Requirements

4.1.1 Scheduler

- S0010 – Provide a way to execute a lambda in another thread with an optional schedule
- S0020 – A schedule must at least support:
  - A delay
  - A cron/Quartz like syntax
  - An interval with an initial delay
  - An temporal object

  - Temporal Adjuster based

- S0060 – Allow the scheduled lambda to throw Exceptions. Periodic tasks must always continue to be executed even if they throw Exceptions.
- S0070 – It must be possible to specify intervals so that they are at a fixed rate.
- S0080 – Provide a Temporal Adjuster implementation for the Quartz cron syntax
- S0090 – Provide a Temporal Adjuster implementation for an interval based on durations.
- S0100 – Periodic schedules must be able to provide a push model for the objects the lambda returns on each invocation. This will allow periodic schedules to act as a timed event producer.
- S0110 – For a given registration and scheduled instance time the lambda must be executed only once.
- S0120 – Lambdas that take too much time to execute (at least 10 secs) must be black listed. This should be configurable.
- S0130 – Exceptions in lambda execution must be logged
- S0140 – It must be possible to stop a periodic schedule
- S0150 – One time schedules should not be canceable since this creates complex race conditions.

4.1.2 Promises

- P0010 – Wrap a promise so that it will fail if it is not resolved before a given delay or instance time.
- P0020 – Wrap a promise so that it will delay after it is resolved. Failures are reported immediately.
- P0030 – Wrap a promise so that it will not resolve before a given instance time. Failures are reported immediately.
- P0040 – Lambdas that are scheduled once must be able to return a value through a promise.

4.1.3 Whiteboard

- W0020 – Provide a whiteboard service model to schedule a service
- W0030 – The syntax for the whiteboard schedule must be Quartz cron syntax
- W0040 – Tasks must be repeated a configurable number of times (minimally 3) if they fail with an exception.
- W0050 – A Task must only be executed once for a given scheduled instance time.

5 Technical Solution

First give an architectural overview of the solution so the reader is gently introduced in the solution (Javadoc is not considered gently). What are the different modules? How do the modules relate? How do they interact? Where do they come from? This section should contain a class diagram. Then describe the different modules in detail. This should contain descriptions, Java code, UML class diagrams, state diagrams and interaction diagrams. This section should be sufficient to implement the solution assuming a skilled person.

Strictly use the terminology a defined in the Problem Context.

On each level, list the limitations of the solutions and any rationales for design decisions. Almost every decision is a trade off so explain what those trade offs are and why a specific trade off is made.

Address what security mechanisms are implemented and how they should be used.

6 Data Transfer Objects

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

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

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

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

7 Javadoc

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

8 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.

9 Security Considerations

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

10 Document Support

10.1 References

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

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

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.***

10.2 Author's Address

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document