RFP-166 Time

Draft

10 Pages

Abstract

This RFP requests 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 must take advantage of the OSGi Promise API and the date-time API introduced in Java 8.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	*26-11-14*	*Initial* *Peter.Kriens@aQute.biz*

1 Introduction

This RFP 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 Use Cases

4.1.1 Cron Job

Al Bundle needs to execute a statistic report every first Monday of the month at 6.00 AM. Al creates a Task service with a service property that specifies a cron schedule:

"0 0 6 ? * 2 *"

When the time comes, the execute method on the Task service is called. Unfortunately, the DNS does not work and the Task fails with an Exception. After a delay, the Task is executed again, fortunately, this time it succeeds.

4.1.2 Periodic Execution

Al Bundle needs to poll a remote server so he gets the Scheduler service and gets a callback every 5 minutes. When his bundle closes, the schedule is automatically removed.

4.1.3 Delay in Promises

Al Bundle has a Promise but must delay 50 seconds after that promise is satisfied before the next step in his sequence can be executed. He goes to the Scheduler service and gets a new Promise that will be resolved 50 seconds from now.

4.1.4 Timeouts in Promises

Al Bundle has a Promise but wants a failure when the Promise is not resolve before 50 seconds from now. He goes to the Scheduler service and gets a Promise that will be resolved or get be failed if the resolving takes more than 50 seconds.

5 Requirements

5.1.1 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.

5.1.2 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.

5.1.3 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.

6 Document Support

6.1 References

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

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

[3]. https://github.com/ReactiveX/RxJava/tree/1.x/src/main/java/rx/subjects

[4]. http://www.quartz-scheduler.org/

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document