RFP 132 – Asynchronous Services

Draft

10 Pages

Abstract

The original requirements document for Distributed OSGi, RFP 88, included a requirement to support asynchronous communication protocols. This was deferred to the next release due to time constraints. However, this remains a significant requirement for enterprise applications using the OSGi Framework, especially to address application requirements for loose coupling, scalability, and reliability. This document expands on the requirements for asynchronous OSGi Services and how they would be added to the OSGi Framework. Asynchronous Services are expected to be a very elegant way to fit asynchronous distributed computing into the OSGi programming model.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

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 6.1.

`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	January 2010	David Bosschaert (david.bosschaert@progress.com) Separated Aysnc Services from the general asynchronous communications RFP 124.
0.1	March 11th, 2010	David Bosschaert (dbosscha@progress.com) Rewrote requirements section following focus group conference call (March 9th)
0.2	September 21st, 2013	Reactivate RFP focussing on asynchronous P2P execution, rather than PubSub and messaging, which is covered under other RFPs
0.3	September 26th, 2013	Update Logo and Licence to match the current RFP template

1 Introduction

Over the last decade there has been a significant shift in computing hardware. Most systems now have access to multiple CPU cores, and can therefore only achieve optimal throughput when processing work on mulitple threads. When mixed workloads are being processed there are further benefits to having multiple threads, computational jobs can continue while other jobs are blocked, for example waiting on IO devices.

Historically these improvements in hardware and throughput have been were achieved through “Request level parallelism”. In this case multiple client requests (e.g. HTTP GET requests) are processed simultaneously on separate threads, but each request is executed in a single-threaded way. This works well – up to a point. If the individual requests are long-running, or result in multiple further requests, then these requests may proceed slowly, despite CPU resource being available.

To best leverage the capabilities of modern systems programmers are making more use of asynchronous execution. Some single threaded systems, such as JavaScript engines, make heavy use of asynchronous processing to allow multiple functions to appear to run concurrently. Several different mechanisms exist for exploiting asynchronous execution.

- Fire and forget – In the case that the return value of a method is not needed in subsequent execution, calling the method has no local side-effects, and need not complete before the caller returns, then it can be started and left running. Good tasks for this sort of execution model include
  - Submitting an “order” to a back-end system, which will be batch processed later
  - Logging to a central log store
- Blocking Promises – In some cases the return value from one or more methods is needed before execution can complete, however if these methods do not have a “happens before” dependency relationship then they can be executed asynchronously and potentially in parallel. Once submitted the caller receives a “promise” which can be queried to find out the eventual result. These are known as “blocking” promises because the initiating thread blocks until the result is available. Java's Future/ExecutorService API is an example of this sort of asynchronous model. The Blocking Promises model has two significant advantages over the fire and forget model; it supports returning values and it allows the caller to wait until the asynchronous action has completed. Blocking the calling thread can, however, lead to the same sorts of problems seen with request-level parallelism. The calling thread is “lost” to the system until the necessary asynchronous tasks complete. If the asynchronous tasks also make use of blocking promises then the asynchronous threadpool can be exhausted, leading to deadlock.
- Promises with callbacks – JavaScript makes heavy use of promises, but due to JavaScript's single-threaded execution model it is not possible for callers to block without deadlocking the whole process. As a result JavaScript promises work differently, rather than providing a blocking get method javascript promises are passed a function to execute on successful completion, or on failure. This callback mechanism ensures that deadlock does not occur, but still allows access to the return value of the asynchronous execution. The most popular promise implementations also provide ways to chain and aggregate promise returns, delaying callbacks until the relevant tasks are complete. Because of the way that promises with callbacks work they have typically been less popular in Java, where declaring an inner class as a callback is quite verbose. This is likely to change with the introduction of lambda expressions in Java 8.

In OSGi services are used as a way for loosely coupled bundles to interact, and represent a natural boundary for execution. Services declare an interface contract (which may be a Java class or interface), which can be referenced and called without having to directly load or instantiate the implementation. Many service calls are short, but some may be long-running, or delegate to remote implementations. In general OSGi services execute synchonously, and only offer asynchronous behaviour when it is explicitly built in to both the API and implementation. This obviously limits the use of asynchronous programming paradigms in OSGi.

The current Remote Services specification defines a set of properties to add to an OSGi service to enable it for remote communication across address spaces using existing distributed computing software systems, using synchronous request/response semantics. In this way Remote Services behave much like local OSGi services.

The synchronous semantics of Remote Services meet many requirements for distributed computing, but due to their higher latency and cost they can benefit even more from the capabilities of asynchronous communication. One need only look at frameworks such as AKKA to see how asynchronous protocols can be used to improve the simplicity and scaling performance of distributed systems. Given that Remote Services can benefit so greatly from asynchronous execution additional requirements should be included to ensure that asynchronous communication models and protocols can be levaraged where possible.

Requirements for an asynchronous programming model should be considered additional to the existing Remote Services Specification's request/response model, not as a replacement.

2 Application Domain

This section explores various aspects of adding support for asynchronous execution. Asynchronous execution typically is achieved via the introduction of a queuing mechanism for "tasks" which are pulled in and executed by one or more Threads. In the case of remote invocations, the task queue is often on the remote machine, allowing the request to be sent and executed without occupying a local Thread. These mechanisms are often also used to handle events, for example the OSGi Event Admin Service provides an asynchronous communication model.

The ability to deal with more fine-grained failure scenarios is another important reason for the introduction and use of asynchronous communication protocols in enterprise applications. Asynchronous protocols provide the ability to restart a request without asking the user to re-enter the data and resubmit when a communication failure occurs.

Synchronous invocations are typically easier to program, but once a client makes a request, either local or remote, the client is blocked waiting for execution to complete and return control to the client. While asynchronous execution may be more complex to program, it offers many benefits and advantages.

For example, synchronous remote invocations depend on the availability of the network during request execution. If a client or server fails during the execution of a request, the request typically has to be resubmitted. This may not be a problem for some applications, where it's easy to re-create the request input. But for other applications, such as an ATM, gas pump, or electronic funds transfer, it may not be easy to recapture the input data and create another request message, and asynchronous protocols meet the requirement better. Even when it is possible to recreate a request message, it is not always easy to know at which point the server failed – i.e. whether or not an update was performed as a result of executing the request, and if so, whether performing the update a second time might cause data inconsistency. And in this case asynchronous protocols can also offer some advantages.

Synchronous invocations operate on a first-come, first-served scheduling mechanism (i.e. the computer has to process requests as they are made by the caller). This means that it's not easy to treat some invocations with higher priority than others, although this is a common application requirement (for example, a bank wants to process the outstanding $1M deposits ahead of the $10 deposits near the end of the banking day). As they have work queues, asynchronous processing engines can process work in an arbitrary order if they choose.

2.1 Asynchronous programming models

A variety of asynchronous programming models are successfully used in enterprise applications today, including ExecutorServices, Async EJBs, REST/HTTP, store-and-forward, pub-sub, and broadcast/multicast to name a few. These programming models assume that a task is visible to a program using one or more asynchronous submission mechanisms (for example, JMS) and that the program is responsible for explicitly creating or retrieving a response using the API and then may acts upon it in a way that is visible to another program using the same API.

For example, a store and forward system has one program submitting a message to a queue using a SEND or SUBMIT command, and another program retrieving the message from a queue using a RECEIVE or DEQUEUE command from the asynchronous programming model API. The sending program is responsible for packing, or serializing, the message, and the receiving program is responsible for unpacking, or de-serializing the message. (Some APIs define a wire format while others do not.)

In each case, management utilities are required to configure the capabilities of the asynchronous implementation being used so that they are able to reject work when overloaded, make best use of the resources available, and to identify, report and resolve any errors that may occur.

2.2 Mixture of programming models

Many enterprise applications require both synchronous and asynchronous execution models for different types of IT functions. For a reserved ticket purchase, for example, it may be necessary to synchronize the database update with the reply to the user to indicate the ticket was purchased, since only one person can have a given seat. For a book purchase, however, it may be sufficient to reply to the user that the order was received, and that it would be fulfilled later. Some of the fulfillment operations for a book order might also use synchronous communications, for example to debit inventory while packing the order for shipment.

2.3 Terminology + Abbreviations

3 Problem Description

The current OSGi programming model for communications among components and bundles is based on the OSGi service interface, which implies a synchronous semantic (i.e. the client invokes on the interface and waits for the reply), and language objects as parameters. These characteristics are typical of local invocations and distributed RPC and meet many requirements, but we want to extend these capabilities to support asynchronous invocation.

3.1 Asynchronous Services

We propose that the EEG evaluate options for specifying the following areas.

Asynchronous invocation of services – specifically the ability for a client to issue an invocation on a service interface without waiting for completion, and relying on a later notification or polling to check completion and retrieve results. For illustration, a low-level equivalent of such a framework is provided in

J2SE by the Future interface. Other technologies (such as CORBA) provide asynchronous 'one-way' support on their remote interfaces. There are significant design considerations involved in selecting whether this may be defined within the "OSGi Services" architecture, and/or "Blueprint", and/or Remote Services; and how a particular choice of solution relates to all three architectures.

4 Use Cases

The use cases describe situations in which requests are received and processed at some later time, which are typically the kind of use cases for which asynchronous execution is a better solution. Similarly, the use cases include situations in which the sending and receiving programs communicate indirectly using some type of intermediate store.

1. A web store receives an order for a book and submits the order immediately into a queue, letting the customer know the order was received. Later, perhaps hours or even days later, the order fulfillment program retrieves the order from the queue and processes it.

2. A bank receives an electronic funds transfer request and immediately puts the information into a file, and notifies the sender that the transfer was received. Later, probably seconds or minutes later, the banking system deposits (or withdraws) the transfer amount into (or from) the appropriate account.

3. A telecommunications company's network switch detects an error and publishes an event to report it. Later, probably seconds later, the network management system receives the event and displays a message on the console to alert the operator.

4. An electronics company places an order for parts to build PCs, potentially from different suppliers for CPUs, disks, displays, memory chips, etc. Each supplier acknowledges receipt of the order and notifies the electronics company of the likely shipment date and cost. Later, each supplier processes and ships the order, and notifies the electronics company the order has shipped. Even later, the electronics company receives and confirms all the shipments, ending the transaction with multiple parties, each of whom receives the confirmation separately.

5. A photography website accepts uploads from users, then performs facial recognition to automatically "tag" known users in the photos. When uploads complete they need to trigger the facial recognition, but should not block the sites ability to accept new upload requests.

6. In using an ATM or other point of sale device, transaction requests need to be captured whether the back office server is online or not. Transactions also need to be processed by the server once and only once.

5 Requirements

AS01 – The solution MUST provide a standard client-side API for making asynchronous invocations on existing, synchronous, OSGi services,where the invocation returns quickly and a return value can be obtained later.

AS02 – The solution MUST allow transparent delegation to services that are already implemented in an asynchronous fashion, therefore servicing the asynchronous requests through their own implementations.

AS03 – The solution MAY provide a synchronous client-side API to services which are implemented in an asynchronous fashion.

AS04 – The solution MUST allow for one-way asynchronous services.

AS05 – The solution MUST support Promises, where invocations can be made that later return a value

AS06 – The solution SHOULD support callbacks when asynchronous executions complete, both successfully and unsuccessfully

AS07 – The solution MUST be applicable to both local OSGi Services as well as Remote OSGi Services.

AS08 – The solution MUST be fully backwards compatible with existing OSGi Service and Service Registry usage.

AS09 – The solution SHOULD be sympathetic to Java 8's lambda support, meaning callbacks should follow the Single Abstract Method principle where possible.

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

6.2 Author's Address

Name	Eric Newcomer
Company	Progress Software
Address	14 Oak Park Drive, Bedford MA USA
Voice	+1 781 280 4000
e-mail	ericn@progress.com

Name	Giovanni Boschi
Company	Progress Software
Address	14 Oak Park Drive, Bedford MA USA
Voice	+1 781 280 4000
e-mail	giovanni.boschi@progress.com

Name	David Bosschaert
Company	Progress Software
Address	158 Shelbourne Road BallsBridge Dublin 4 Ireland
Voice	+353 1 637 2000
e-mail	dbosscha@progress.com

Name	Tim Ward
Company	Paremus Ltd
Address	
Voice	
e-mail	Tim.ward@paremus.com

6.3 End of Document