RFP 124 – Asynchronous Communications

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 communications.

1 Document Information

1.1 Table of Contents

1Document Information....2
1.1Table of Contents....2
1.2Terminology and Document Conventions....2
1.3Revision History....2
2 Introduction....3
3Application Domain....4
3.1Communicate using messages 5
3.2Asynchronous programming models....5
3.3Mixture of programming models....6
4 Problem Description....6
4.1Asynchronous Messaging API (JMS) 6
4.2Message Driven Components....7
4.3Asynchronous Services....7
5Use Cases....7
6Requirements....8
1.1 Asynchronous Messaging API....8
1.2 Message-Driven Components....9
1.3 Asynchronous and Multipoint Invocation....9
7Document Support....9
7.1References....9
7.2Author's Address....9
7.3End of Document....10

1.2 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.`

1.3 Revision History

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

Box

All Page Within This Box

Revision	Date	
Initial	July 21, 2009	Initial draft Eric Newcomer ericn@progress.com David Bosschaert david.bosschaert@progress.com Giovanni Boschi giovanni.boschi@progress.com
0.2	November 5, 2009	Revised after Dublin F2F feedback: factor problem domain into 3 concrete areas – JMS, Message Driven Components, and Asynchronous; added requirements; editorial changes to intro and application domain sections. Giovanni Boschi (gboschi@progress.com)

2 Introduction

The current Remote Services specification defines a set of properties to add to a Java interface for remote communication across address spaces using existing distributed computing software systems, using synchronous request/response semantics. These semantics meet many requirements for distributed computing, but they are not easily mapped to the full capabilities of asynchronous communication protocols. Therefore additional requirements need to be detailed on which to base one or more new designs and specifications for asynchronous communication models and protocols, both to extend the use of Java interfaces to embrace asynchronous semantics, and to explore new APIs and programming models.

The Java interface programming model currently in the OSGi specification provides a significant capability for communications across multiple JMVs and address spaces, and can be used to meet a broad range of enterprise application requirements. However, many enterprise application requirements are better met through the use of asynchronous communication protocols, for example implementing loosely coupled SOA designs, ensuring reliability guarantees, controlling execution priority, and enabling certain load balancing and scalability techniques.

Although it is possible to use asynchronous communications protocols to implement the request/response semantic of a Java interface, the full benefit of asynchronous protocols may be achieved only by using an asynchronous communication programming model, allowing direct programmatic access to APIs and configuration metadata that expose the full range of asynchronous communication features and functions.

Some of the historical tradeoffs in distributed computing technology have occurred between simplifying the programming model (e.g. using RPC) and allowing applications more fine-grained control over the communication among programs in distributed locations by directly handling the communications. Full access to asynchronous communication functions requires the use of programming model for such primitives as SEND (now) and RECEIVE (later) that cannot easily be incorporated into a program-call type interface that uses a blocking (i.e. synchronous) programming model, and assumes that either the communication is successful or an exception is generated. Similarly a request/response interface is difficult to use for pub/sub, broadcast, or send once/multiple reply message exchange patterns.

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

Box

All Page Within This Box

3 Application Domain

This section explores various aspects of adding support for asynchronous communication protocols. Asynchronous communications typically is achieved via the introduction of a queuing mechanism for store and forward style communication, or the use of topics and channels for pub/sub, broadcast, and multicast mechanisms. 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 communication protocols are typically easier to program, but once a client sends a message to a server the client is blocked waiting for the server to execute the request and return a reply. While asynchronous communications may be more complex to program, they offer many benefits and advantages.

For example, synchronous communications protocols 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.

Another problem is the availability of the client and server programs in the first place. One or the other might not be available at the time communication is needed. For example, the server program could be down when the client tries to send a request message to it, or the client program could be down when the server program tries to send a reply. The client is then blocked until the server becomes available again, as is the server until the client becomes available again. A user may have to wait or come back later to submit the request. A good alternative, which is not available for synchronous communication protocols, is to have the system wait and send the request to the server as soon as it becomes available. The user can then continue processing other requests and come back later to receive the reply.

When the client is unavailable to the server and the server is unable to reply, a similar set of problems arises for synchronous communications protocols. The reply may be lost if the server is unable to send it to the client, and the client may never receive a reply despite the fact that the server successfully processed the request. In this case the client doesn’t know whether or not the request actually executed, and whether or not to resubmit it.

In the case of a network failure, a synchronous communication protocol has no way of letting either the client or the server know the state of a request. All it can do is report the network error, not what happened on either the client or the server. There’s no way to reliably know, in other words, whether the request failed to be executed due to a communications failure or a problem with the server. It may be ok for some applications to simply retry such a failed request, but when an application depends on a request being executed exactly once, there is really no way for synchronous communications to be programmed to always meet this common requirement (i.e. a share of stock should not be bought twice, or a deposit performed twice, or the same ticket sold to two customers).

Box

All Page Within This Box

Another problem area is load balancing. In synchronous communication protocols it's possible to set up a pool of servers and threads to balance a request load, but since the load involves binding the client to the server, the system must provide a mechanism for binding a client to one of the servers in the pool. If the server maintains any state for the client, the system must also provide a way for the client to maintain a session with a particular server instance so that subsequent calls from the client are sent to the same server instance.

Random selection is one technique that can be used to balance the load for synchronous communications, for example, but it can be more complex to reliably balance an overall application load across all servers than using an asynchronous protocol, especially given varying lengths of request processing time.

Synchronous communication protocols operate on a first-come, first-served scheduling mechanism (i.e. the server has to process requests in the order they arrive since there's no way to predict or control request ordering). This means that it's not easy to treat some requests 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).

3.1 Communicate using messages

Instead of using the service interface model to define interactions between bundles (i.e. the export/import mechanism for sharing interface arguments) a given set of bundles could instead exchange information with another set of bundles using data items defined in a message. Data in a message does not have to be structured using an interface method with arguments – it can be structured according to binary, text, or XML, or can be untyped. It can be defined using a message type as well, which could be registered and discovered in a similar way that service interfaces as registered and discovered. Bundles communicating using messages could use a message-oriented API to send and receive messages, potentially using an intermediary such as a queue or other storage resource (persistent or volatile).

3.2 Asynchronous programming models

A variety of asynchronous programming models are successfully used in enterprise applications today, including REST/HTTP, store-and-forward, pub-sub, and broadcast/multicast to name a few. These programming models assume that a message is visible to a program using one or more asynchronous messaging APIs (for example, JMS) and that the program is responsible for explicitly creating or retrieving a message 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.)

A pub-sub system has a receiving program subscribe to a particular communications channel and the sending program publishing a message to the channel (or topic). A broadcast system has a program sending a message to a list of recipients.

In each case, management utilities are required to configure the capabilities of communication protocol being used so that they are available to publishing and consuming programs, and available to identify and resolve any errors that may occur during a given message communication pattern.

Box

All Page Within This Box

3.3 Mixture of programming models

Many enterprise applications require both synchronous and asynchronous communications 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.

4 Problem Description

The current OSGi programming model for communications among services is based on the Java 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 distributed RPC and meet many requirements, but we want to extend these capabilities to support asynchronous invocation, and message-oriented communications

4.1 Asynchronous Messaging API (JMS)

A significant number of messaging middleware systems are available in the enterprise, implementing various types of asynchronous communications, such as store-and-forward queues, pub/sub channels, and broadcast/multicast mechanisms, with various levels of QoS. These should be available to applications developed using OSGi or adapted to the OSGi framework, and available to infrastructure implementing higher-level asynchronous communication models on the OSGi platform. JMS is widely implemented and adopted, with JMS clients available for most legacy MOM middleware, making it a relatively easy choice to leverage for this purpose.

JMS is being used in OSGi frameworks today, but there are several areas of use that we want to ensure are adequately specified:

- In JSE and JEE, applications access JMS providers by looking up a connection factory from JNDI or constructing connection factory objects directly; both of these techniques will continue to be used within OSGi, for various reasons, both by legacy code migrating into OSGi, and by new applications.
- In addition to the mechanisms above, there are likely benefits (consistency of programming model, version-aware and property-aware selection, etc.) to registering and accessing JMS connection factories as, or via, services in the service registry.
- Transactional behavior, and if appropriate connection pooling, consistent at least with JDBC; the existing approach in the Java platform relies on the JCA resource adapter architecture, which couples transactional and connection pooling behavior to some degree, and it's possible that the eeg may want to look at these as related problems and design a similar model. It's worth clarifying that general support for JCA in OSGi is *not* in scope of this problem – although it is within the design and standards discretion for EEG to consider the reuse of some JCA technology in designing the solution.

4.2 Message Driven Components

A common pattern in asynchronous messaging architectures is the development of components purposed to processing messages, by processing them directly or by delegating to interface-driven components. There are examples of this in several environments, including for Java the Message-Driven Beans (MDB)

The proposal is to specify within OSGi a component model for message-driven components. There is ample design space available for this, but the main goal of the specification would be to define an architecture well integrated with OSGi-specific facilities like Blueprint and OSGi Services and complementary to Remote Services, so that message-driven components may delegate to other business logic components after deserializing and perhaps partially processing a message. *The migration of JEE MDBs for unmodified deployment into OSGi is not directly a goal of this proposal*

4.3 Asynchronous Services

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

- Asynchronous invocation of components – 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, and component-level variants are provided within EJB3 and JAX-WS. 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.
- Multipoint invocation - how a component may invoke multiple services with equivalent interfaces in a single call, over a publish/subscribe messaging transport from a JMS provider, and returning a multivalued result. An important goal is to support this over arbitrary interfaces exposed by the invoked services, without imposing dependencies on the called components being developed specifically for this - ideally any Remote Service, but design issues may need to introduce interface constraints. One hypothetical approach could be to specify a client binding pattern which defines and registers derived service interfaces with multivalue return types with collection generics, and uses service metadata properties to define policies like completion timeouts, etc.

5 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 communications protocols are 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 persists 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. Data to be shared among multiple applications across different company divisions has to conform to a common schema or type system compatible with Java programs. Messages to be serialized for use with asynchronous protocols can be based on Java data types within interface definitions to simplify the task.

6. Data to be shared among multiple applications across different company divisions has to be abstract from any specific language or set of data types since the different applications use different languages and technologies. Messages to be serialized are based on an independent schema type (such as XML) or even on untyped data that is packed and unpacked by the application programs.

7. 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.

6 Requirements

1.1 Asynchronous Messaging API

The solution MUST compliant with the JMS specification, at least in terms of the core messaging semantics of the API – threading model requirements, reliability semantics, acknowledgement, failure modes, etc.

The solution MUST provide a way to select JMS providers dynamically at runtime, without code dependencies

The solution SHOULD provide a standard way to configure JMS providers on the OSGi platform

The solution MUST provide a configuration and runtime lookup mechanism for administered objects based on the OSGi service registry. It SHOULD provide a declarative mechanism for selecting a JMS provider for a given bundle at deployment.

The solution SHOULD support the configuration and lookup of JMS administered objects from JNDI, alongside any other mechanisms

The solution SHOULD avoid or at least minimize changes required to existing JMS provider JARs

The solution MUST support XA-transactional messaging, consistent with RFC98, in general, but specifically for use of JMS and JDBC XA resources in the same transaction

The solution MUST provide for connection and session pooling

The solution MUST support access to provider-specific methods (those not defined by the JMS interfaces) in JMS provider classes.

1.2 Message-Driven Components

The solution MUST provide the ability to configure bindings between message-driven components and JMS destinations, and selection of QoS policies

The solution MUST provide for both one-way and request-response invocations of message-driven components

The solution MUST use the same set of JMS providers configured in the framework for general JMS use.

The solution MUST support Blueprint components as message-driven components

The solution SHOULD support the handling of arbitrary JMS messages: of any JMS message type, format, and content.

1.3 Asynchronous and Multipoint Invocation

The solution MUST support asynchronous invocation of any Remote Service, without additional requirements imposed on the service executable. It MUST support alternate synchronous and asynchronous invocations of the same deployed service.

The solution SHOULD support multipoint invocation of the broadest possible domain of Remote Service interface types, ideally any Remote Service interface. It MUST support alternate single and multipoint invocations of the same service.

The solution MAY include support for asynchronous invocation of local OSGi services, if desirable for consistency of programming model.

7 Document Support

7.1 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

7.2 Author's Address

Name	Eric Newcomer
Company	Progress Software
Address	14 Oak Park Drive, Bedford MA USA
Voice	+1 781 492 2003
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	david.bosschaert@progress.com

7.3 End of Document