INTEGRATING ASYNCHRONOUS COMMUNICATION INTO THE OSGI SERVICE PLATFORM

Marc Schaaf, Volker Ahlers, Arne Koschel
Faculty IV, Department of Computer Science, University of Applied Sciences and Arts
120 Ricklinger Stadtweg, 30459 Hannover, Germany

Irina Astrova, Ahto Kalja
Institute of Cybernetics, Tallinn University of Technology, 21 Akadeemia tee, 12618 Tallinn, Estonia

David Bosschaert
Progress Software, 158 Shelbourne Road, Ballsbridge 4, Dublin, Ireland

Roman Roelofsen
Weigle Wilczek GmbH, 42-44 Martinstraße, 73728 Esslingen a., Neckar, Germany

Keywords: Asynchronous communication, Remote services, OSGi service platform, Event admin service (EAS), Message-oriented middleware (MoM).

Abstract: OSGi is a popular Java-based platform that was originally intended for embedded systems. But today OSGi is used more and more in enterprise systems. To fit this new application area, OSGi is continuously extended by the OSGi Enterprise Expert Group (EEG). For example, recently, support for remote services has been added to OSGi. But this support implies only synchronous communication of remote services, thus limiting the application of OSGi in the area of enterprise systems, as enterprise systems typically embody both synchronous and asynchronous communication. To fill this gap, we propose a novel approach to integrating asynchronous communication into OSGi.

1 INTRODUCTION

Due to the manifold requirements of real-world examples, enterprise systems typically embody both synchronous and asynchronous communication (Krafzig, Banke, and Slama, 2005). Synchronous communication is characterized by that a service sends a request to another service and keeps waiting until it receives a response. Asynchronous communication is less stringent. A service sends a request to another service but it does not wait for a response. Rather, it continues working. The response can be received at any later time.

Recently, support for distribution has been introduced into OSGi. This support is based on synchronous invocations of methods for accessing remote services. However, OSGi does not specify any methods for asynchronous accessing remote services yet; this is an area of future standardization. Therefore, we propose a novel approach to integrating asynchronous communication into OSGi.

2 OSGI

The OSGi Service Platform (OSGi Alliance, 2009) is a popular Java-based platform that has arisen in the context of embedded systems. OSGi is freely available and continuously extended by the OSGi Enterprise Expert Group (EEG).

There are a number of commercial and open source implementations of OSGi, including Eclipse

Schaaf M., Ahlers V., Koschel A., Astrova I., Kalja A., Bosschaert D. and Roelofsen R..
INTEGRATING ASYNCHRONOUS COMMUNICATION INTO THE OSGI SERVICE PLATFORM .
DOI: 10.5220/0003401601650168
In *Proceedings of the 7th International Conference on Web Information Systems and Technologies* (WEBIST-2011), pages 165-168
ISBN: 978-989-8425-51-5

Equinox, Apache Felix, Knopflerfish and ProSyst's mBedded server. Well-known applications that are based on OSGi include the Eclipse IDE and Apache Service Mix.

The core of OSGi is the OSGi Framework. This framework simplifies the development and deployment of extensible applications also called bundles, by decoupling the bundle's specification from its implementation. This means that a bundle is accessed by the framework through an interface, which is by definition separate from the bundle's implementation. This separation enables changing the bundle's implementation without changing the environment and other bundles.

The OSGi Framework makes it possible to run multiple applications simultaneously within a single Java Virtual Machine (JVM), by dividing applications into bundles that can be loaded at runtime and also removed. For communication within the JVM, the framework provides a service registry to register services, so that services can be found and used by other bundles.

3 OUR APPROACH

Figure 1 provides an overview of our approach. Our approach aims at integrating asynchronous communication into OSGi with little or no impact on OSGi itself. Therefore, our approach is leveraging the OSGi Event Admin Service (EAS), which provides asynchronous communication. This communication is based on the publish-and-subscribe mechanism, where an event sending bundle ("producer") can publish the event to a topic, whereas an event receiving bundle ("consumer") can subscribe to that topic. For subscription, an event receiving bundle registers an event handler service with the appropriate service properties. These properties specify the topic the event handler service is listening for and an optional filter expression. An event handler service can be registered by any bundle that wants to receive the event.

However, leveraging the EAS is more difficult than it originally appears. This is because the EAS provides facilities only for receiving events from bundles and delivering them to all (registered) event handler services. Concepts like distribution and guaranteed delivery are not part of the EAS itself. Next we'll show how our approach addresses these issues.

3.1 Distribution

The EAS provides asynchronous communication. But this communication is limited to a (local) OSGi container. We solved this problem by using message-oriented middleware (MoM) for distribution (see Figure 1).

But herein lies another problem. The communication mechanism of the MoM is based on messages, whereas the communication mechanism of the EAS is based on events. We solved this problem by introducing a mediation component called Event Distribution System (EDS) (see Figure 1). The EDS receives events from the EAS and forwards them as messages to the MoM. In the other direction, the EDS receives messages from the MoM and forwards them as events to the EAS. Thereby the EDS is used to enable the communication between the MoM and the EAS.

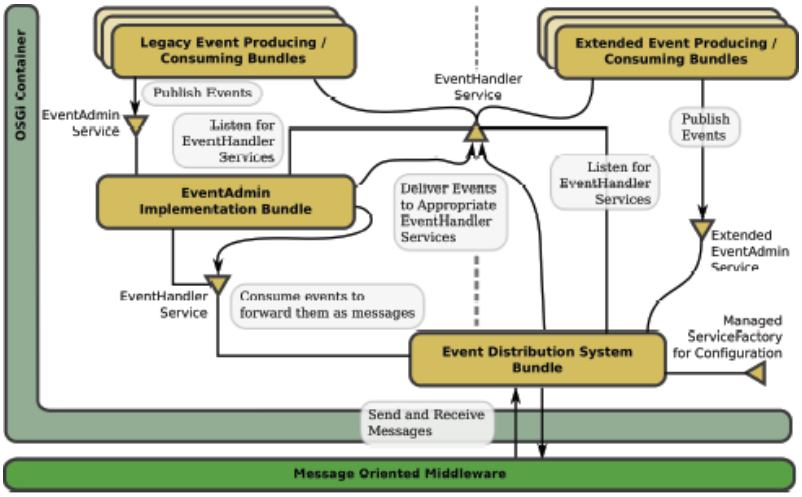

Figure 1: Our approach is leveraging the OSGi Event Admin Service, while leaving the core OSGi relatively untouched.

Figure 2 illustrates this communication when a bundle sends an event to the MoM. At first, this event is sent to the EAS. The EAS fulfils its normal duty by delivering the event to all (registered) event handler services. One of those services is registered by the EDS, which thereby receives the event. The EDS creates a message from the contents of the event and forwards it to the MoM. Thus, the event sending bundle communicates with the EAS only; it has no direct knowledge of the EDS and the MoM.

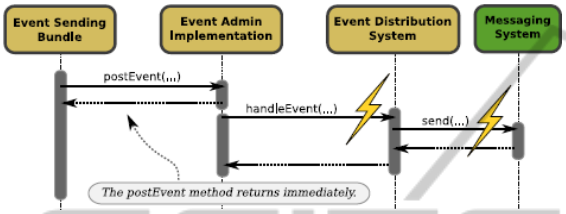

Figure 2: Asynchronous event delivery. The event sending bundle has no possibility to determine if the event was received by the EDS and delivered to the MoM as a message.

Figure 3 illustrates the communication between the MoM, the EDS and the EAS when a bundle receives a message from the MoM. At first, this message is sent to the EDS. The EDS creates an event from the contents of the message and forwards it to the EAS. The EAS delivers the event to all (registered) event handler services that are listening for this event. One of those services is registered by the bundle, which wants to receive the event. Thus, the event receiving bundle also communicates with the EAS only; it has no direct knowledge of the EDS and the MoM.

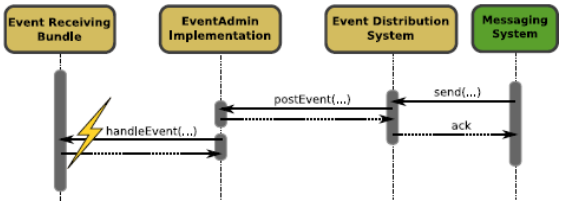

Figure 3: Asynchronous event reception. The EDS has no information if the delivery of the event to the destination was successful. But it needs to acknowledge the successful reception and processing of the message.

3.2 Guaranteed Delivery

The EDS mediates between the MoM and the EAS. It receives messages from the MoM and forwards them as events to the EAS for further delivery to the (registered) event handler services (see Figure 3). As soon as, the EDS forwards an event to the EAS, it cannot know if the event reaches its destination and if the event is processed successfully. We solved this problem by connecting the EDS to the event handler services so that the EDS can forward the event to them directly (see Figure 1). This way the EDS can inform the MoM about a successful or failed delivery.

The EAS completely decouples the event sending bundle from the EDS and the MoM (see Figure 2). It only guarantees the delivery of an event to all (registered) event handler services that are available at the very moment of this delivery. Since the event sending bundle cannot know which of the event handler services are available, it has no information if the event was successfully delivered to the EDS. We solved this problem by connecting the event sending bundle to the EDS (see Figure 1).

Now the event sending bundle knows that the event was successfully delivered to the EDS as it sends the event to the EDS directly. However, the EDS still has no way to inform the event sending bundle about a failed delivery of the event. This is because the EAS API does not allow the EDS to throw an exception from the call of a `postEvent` method.

```
public interface EventAdmin{
   void postEvent(Event event);
   ...
}
```

As a result, the event sending bundle has to send the event blindly without any chance to get information on the success or failure of the delivery. We solved this problem by extending the EAS API. In particular, we defined a new method `postEventReliable`, which waits (blocks) until the event is successfully delivered. In the case of a failure, it throws an exception. Thereby the event sending bundle is informed about the failure and can react appropriately.

```
public interface ExtendedEventAdmin{
   void postEventReliable(Event e) \
   throws MessagingException;
   ...
}
```

But herein lies another problem. Our previous solution breaks the compatibility with legacy bundles. We solved this problem as follows. The right side of Figure 1 shows that bundles will use the extended EAS API to guarantee the delivery. The left side of Figure 1 shows that legacy bundles will not be aware of the extended EAS API. Rather, they

will continue to use the EAS API. The centre of Figure 1 shows an event handler service. It is a registered service that is listening for events to be distributed to both legacy bundles and bundles that are aware of the extended EAS API.

4 RELATED WORK

The work that most closely comes to ours is the ECF Distributed EAS (Eclipse Foundation, 2009). The Distributed EAS is implemented based on the Eclipse Communication Framework (ECF). Thus, this implementation can be used with a variety of protocols like ActiveMQ and XMMP. The implementation replaces the "standard" EAS with the custom-built EAS, which is capable of sending events between other remote EASs. The Distributed EAS also uses the MoM (viz. ActiveMQ Broker) for distribution. However, by contrast to our approach, guaranteed delivery is not part of the Distributed EAS.

5 CONCLUSIONS

Currently, OSGi does not support asynchronous communication of remote services yet. This is a severe hindrance for OSGi to be used widely, especially in distributed environments. As an attempt to fill this gap, we have proposed a novel approach to integrating asynchronous communication into OSGi. Our approach is leveraging the OSGi EAS, while leaving the core OSGi relatively untouched. The EAS provides asynchronous communication. But this communication is between local services only. We have solved this problem by using the MoM for distribution (i.e. remote services). Another problem with leveraging the EAS was that the EAS does not guarantee the delivery of events. We have solved this problem by extending the EAS API.

ACKNOWLEDGEMENTS

Irina Astrova's and Ahto Kalja's work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF).

REFERENCES

Eclipse Foundation, 2009. Distributed Event Admin Service. http://wiki.eclipse.org/Distributed_Event Admin_Service.

Krafzig, D., Banke, K., Slama, D., 2005. *Enterprise SOA: service-oriented architecture best practices*. Pearson Education, Inc.

OSGi Alliance, 2009. OSGi Service Platform Service. Compendium – Release 4, version 4.2. http://www.osgi.org/Release4/.

Rellermeyer, J., Alonso, G., Roscoe, T., 2007. R-OSGi: Distributed applications through software modularization. In *Middleware'07*, vol. 4834. Springer.