Journal of Computing and Information Technology - CIT 20, 2012, 3, 151–157
doi:10.2498/cit.1002091

Asynchronous Messaging for OSGi

Arne Koschel[2], Irina Astrova[1], Marc Schaaf[3], Volker Ahlers[2], Stella Gatziu Grivas[3] and Ahto Kalja[1]

[1]Institute of Cybernetics, Tallinn University of Technology, Estonia
[2]Faculty IV, Department of Computer Science, University of Applied Sciences and Arts, Hannover, Germany
[3]Institute for Information Systems, University of Applied Sciences Northwestern Switzerland, Olten, Switzerland

OSGi is a popular Java-based platform, which has its roots in the area of embedded systems. However, nowadays it is used more and more in enterprise systems. To fit this new application area, OSGi has recently been extended with the Remote Services specification. This specification enables distribution, which OSGi was previously lacking. However, the specification provides means for synchronous communication only and leaves out asynchronous communication. As an attempt to fill a gap in this field, we propose, implement and evaluate an approach for the integration of asynchronous messaging into OSGi.

Keywords: OSGi, Event Admin (EA), asynchronous messaging, reliable message delivery, LightSabre

1. Introduction

Messaging [1] is defined as a middleware technology that enables speedy, asynchronous, application-to-application communication with reliable message delivery. Applications communicate to each other by sending and receiving packets of data called *messages*. Channels are logical pathways that connect the applications and transport the messages.

There are two types of channels: queues and topics. Both are provided by means of message-oriented middleware (MoM). Figure 1 shows that applications are connected over MoM for communication with each other. This communication is asynchronous because an application ("client") sends a request to another application ("server"), but does not wait for a response. Rather, it continues working. The response can be received at any later time. This is in contrast with synchronous communication, where an application ("client") sends a request to another application ("server") and keeps waiting until it receives a response.

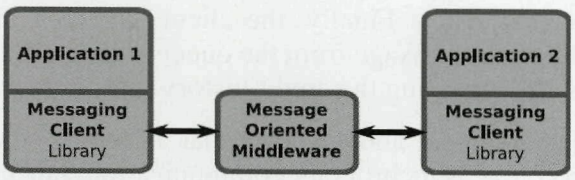

Figure 1. Messaging.

By definition, messaging refers to asynchronous communication. However, it is also possible to implement synchronous communication based on MoM. Therefore, we'll distinguish two types of messaging: synchronous and asynchronous.

2. Motivation

OSGi [3] provides a runtime container for Java-based applications, also known as *bundles*. OSGi has its roots in the area of embedded systems. However, nowadays it is used more and more in enterprise systems. To fit this new application area, OSGi has recently been extended with the Remote Services specification. This specification enables distribution, which OSGi was previously lacking. Still, the specification leaves out asynchronous communication. Rather, it provides means for synchronous communication only. This requires the client and the server to be available at the same time. However, such tight coupling may not be possible or desired in many applications, which are asynchronous by nature.

As an example, let's consider the credit request process submitted to an Automated Underwriting System (AUS) in a home loan processing application [4]. After a borrower submits the loan application, the mortgage company sends a request to the AUS for credit history information. Since this request is for a comprehensive credit report with details such as borrower's current and past credit accounts, late payments and other financial details, it usually takes longer time (hours or sometimes even days) to get a response. It does not make sense for the client to keep a connection to the server open and wait that long for the result. So the communication occurs asynchronously; i.e., once the request is submitted, it is placed in a queue that can be accessed across a network and the client disconnects from the server. Then the AUS service picks up the request from the specified queue, processes it, and puts the result message in another queue. Finally, the client will pick up the result message from the queue and continue with processing the credit history information.

The example above shows that a lack of support for asynchronous communication can be a severe hindrance for further use of OSGi in enterprise systems. As an attempt to fill a gap in this field, in our previous paper [2] we proposed an approach to integrating asynchronous messaging into OSGi. In this paper, we'll also discuss the implementation of our approach and report the results of performance evaluation.

3. Approach

Figure 2 gives an overview of our approach. Since our approach aims at integrating asynchronous messaging into OSGi in a non-invasive way (i.e., without changing OSGi and legacy bundles), it leverages an OSGi's service called *Event Admin* (EA) [3].

The EA already supports synchronous and asynchronous communication, by providing two methods: sendEvent and postEvent. As shown in Figure 3, both methods have the same signature, but different semantic. The sendEvent method sends an event synchronously. Therefore, it will wait until the event is handled by event handler services. (The event handler services can be registered by any bundle that is interested in receiving this event.) By contrast, the postEvent method sends an event asynchronously. Therefore, it returns immediately.

```
public interface EventAdmin{

void sendEvent(Event e);

void postEvent(Event e);

}
```

Figure 3. Event Admin API.

However, leveraging the EA is more challenging than it can appear at first sight. This is because distribution and reliable message delivery are not part of the EA itself. Next we'll show how our approach addresses these issues.

3.1. Distribution

The EA already provides asynchronous communication. But this communication is local (i.e., from within an OSGi container). We solved this problem, by introducing MoM, also known as a messaging system (see Figure 2).

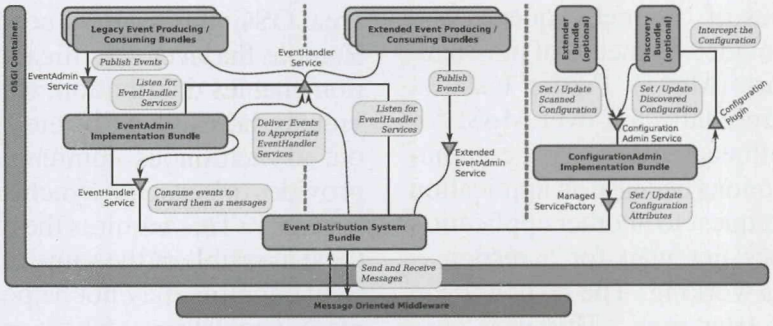

Figure 2. Approach to asynchronous messaging.

But this implies another problem. The communication mechanism of the MoM is based on messages, whereas the communication mechanism of the EA is based on events. We solved this problem by introducing a mediation component called *Event Distribution System* (EDS) (see Figure 2). The EDS receives events from the EA and forwards them as messages to the MoM. In the opposite direction, the EDS receives messages from the MoM and forwards them as events to the EA. Thereby the communication between the MoM and the EA is provided by means of the EDS.

Figure 4 illustrates this communication when a bundle sends an event to the MoM. At first, this event is sent to the EA. The EA delivers the event to the event handler services. One of these services is registered as a listener for the event 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 ("producer") communicates with the EA only; it has no direct knowledge of the EDS and the MoM.

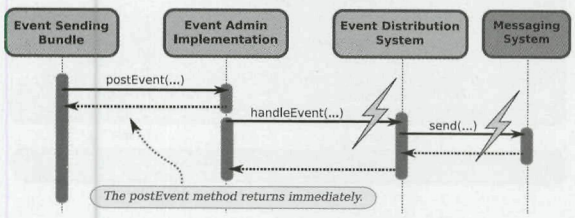

Figure 4. Asynchronous event delivery.

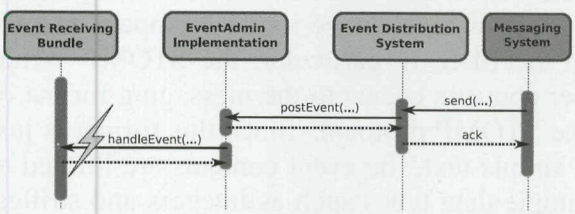

Figure 5. Asynchronous event reception.

Figure 5 illustrates the communication 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 EA. The EA delivers the event to the event handler services. One of these services is registered by the bundle, which is interested in receiving the event. Thus, the event receiving bundle ("consumer") also communicates with the EA only; it has no direct knowledge of the EDS and the MoM.

3.2. Reliable message delivery

As shown in Figure 5, the EDS receives a message from the MoM and forwards it as an event to the EA. The EA delivers this event to the event handler services that are listening for this event. However, once the event has been forwarded to the EA, the EDS cannot know if the event reaches its destination. Therefore, the EDS cannot know if a message can be acknowledged or if an exception needs to be thrown in order to inform the MoM of a failed delivery. 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 2). Thereby the EDS has full control over the delivery process and thus, can acknowledge the MoM when the delivery was successful. The EDS can also store events in a persistent storage and keep them there until they can be successfully delivered to the event handler services. This behavior guarantees reliable message delivery; i.e., the MoM can send a message and be sure that this message will reach its destination, even in the case of a failure.

As shown in Figure 4, the EA receives an event from the event sending bundle and forwards it to the EDS. But the EA can deliver the event only if the EDS is available at the very moment of this delivery. Therefore, the event sending bundle cannot know if the event was successfully delivered to the EDS. We solved this problem by connecting the event sending bundle to the EDS (see Figure 2).

The event sending bundle now 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. We solved this problem by extending the Event Admin API. As shown in Figure 6, the extended Event Admin API provides a method postEventReliable, which has the same semantic as postEvent, but throws an exception in the case of a failure. Thereby, the event sending bundle gets informed about the failure.

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

Figure 6. Extended Event Admin API.

But herein lies another problem. Our previous solution breaks the compatibility with legacy bundles, which are not aware of the extended Event Admin API. We solved this problem as follows. The right side of Figure 2 shows that newly created bundles will use the extended Event Admin API to guarantee reliable message delivery. The left side of Figure 2 shows that legacy bundles will continue to use the "standard" Event Admin API. The centre of Figure 2 shows an event handler service; it is a registered service that is listening for events to be distributed to both legacy and newly created bundles.

4. Implementation

To prove the feasibility of our approach, we implemented an open-source distributed middleware system called *LightSabre* (`http://fusesource.com/forge/projects/LIGHTSABRE`) in cooperation with two industry partners. These partners established the following requirements for LightSabre:

- It should use the ActiveMQ Broker as MoM.
- It should run on Java Micro Edition (Java ME) to demonstrate its usability for embedded systems where OSGi has its roots.
- It should run on Java Enterprise Edition (Java EE) to demonstrate its usability for enterprise systems where OSGi is increasingly used nowadays.

Figure 7 gives an overview of LightSabre. Since the EDS should be able to communicate with ActiveMQ Broker, we might use the ActiveMQ client libraries to implement the EDS. However, these libraries require Java 1.5 and thus, they cannot be used in Java ME, which is based on Java 1.4. Although the translation of Java byte code from 1.5 to 1.4 is possible (e.g., using a RetroTranslator tool), the libraries are still too big for loading into a typical Java ME runtime, which has small memory footprint compared to Java EE. We solved this problem, by introducing two additional mediation components. One is a *STOMP Wrapper*, which allows the EDS to communicate with the broker from within Java ME; this communication is done over a STOMP protocol. Another mediation component is an *ActiveMQ Wrapper*, which allows the EDS to communicate with the broker from within Java EE; this communication is done over an OpenWire protocol, the native protocol of the broker. Thereby LightSabre can run on both Java ME and Java EE. The wrappers also helped us to make the EDS independent of a protocol-specific API.

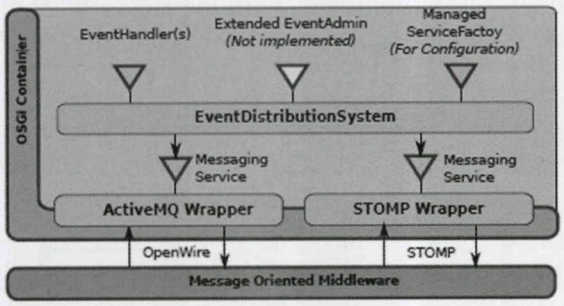

Figure 7. LightSabre.

The conversion from events to messages and back now takes place in the wrappers and not in the EDS. In particular, the STOMP Wrapper converts events to the messaging format of the STOMP protocol. Since this format is just a simple text, the event contents are limited to simple data types such as integers and strings. By contrast, the ActiveMQ Wrapper converts events to JMS Map Messages, the messaging format of the OpenWire protocol. Thus, all serializable Java objects can constitute the event contents.

LightSabre was successfully tested in Eclipse Equinox, Apache Felix and Sprint Titan. However, LightSabre can be used with any OSGi implementation as it leaves OSGi unchanged.

5. Performance Evaluation

To evaluate the performance of LightSabre in Java EE, we compared it to *Java Message Service* (JMS) [5] because JMS already provides both synchronous and asynchronous messaging. We made a conjecture that LightSabre will be slower than JMS due to the overhead of having the EDS and the ActiveMQ Wrapper sitting between the EA and the broker.

To check our conjecture, we wrote two types of applications: synchronous and asynchronous. *Synchronous applications* sent a number of messages to the specified queue from which they retrieved the messages again. However, the applications always waited for the reception of the last message they had sent, before sending a new one. Thereby they sent and received the messages synchronously. *Asynchronous applications* sent a number of messages as fast as possible to the specified queue. However, while doing so, the applications also waited for messages on the same destination in a different thread and therefore without blocking the send operations. Thereby they sent and received the messages asynchronously.

Each of those two types of applications was implemented on a base of both LightSabre and JMS, resulting in four applications in total (two synchronous and two asynchronous). For all the four applications, we measured how long it took from sending the first message until the reception of the last message. Each of these measurements was repeated 15 times. In each application run, 100,000 messages were sent and received. Furthermore, each of the measurements was done with the broker on a remote machine as well as with the local broker that was running on the same machine as the client. The remote machine was an Ubuntu Linux system with 4GB memory. The machine, which hosted both the broker and the client, was also an Ubuntu Linux system but with 8GB memory. The two machines were interconnected with a gigabit Ethernet.

Figure 8 shows the average results of the different application runs. The asynchronous applications that were based on LightSabre needed 5.63 and 6.46 seconds with the local and remote brokers, respectively. However, the asynchronous applications that were based on JMS needed 4.47 and 5.15 seconds with the local and remote brokers, respectively. So LightSabre was 25% slower than JMS, regardless of whether the broker was local or remote.

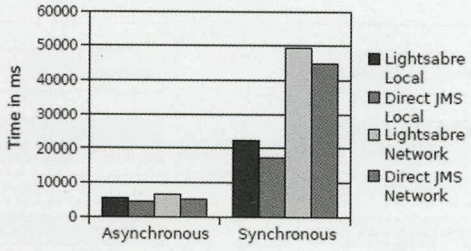

Figure 8. Results of performance evaluation.

The synchronous applications that were based on LightSabre needed 22.28 and 49.32 seconds with the local and remote brokers, respectively. However, the synchronous applications that were based on JMS needed 17.45 and 44.88 seconds with the local and remote brokers, respectively. So in the case of local synchronous communication, LightSabre was 27% slower than JMS, whereas it was only 10% slower when the client communicated with the broker over a network.

Thus, our experiments did show some decrease in the performance. However, taking into account that the EDS and the ActiveMQ Wrapper were not optimized at all, we expect to get better performance after their optimization. Our experiments also showed that the decrease became less significant in the case of remote synchronous communication.

6. Related Work

The work that most closely comes to ours is ECF (Eclipse Communication Framework) [6], where asynchronous messaging is implemented on top of JMS by Distributed Event Admin (DEA). The DEA replaces the "standard" EA and thereby becomes responsible for distributing events to remote services.

Figure 9 gives an overview of the DEA, which uses ActiveMQ Broker as MoM and supports many protocols, including ActiveMQ. The biggest disadvantage of this approach is that it is invasive; the DEA requires that a new API should be used. Thereby the approach breaks the compatibility with legacy bundles, which are not

aware of the new API. Another big disadvantage is that reliable message delivery is not part of the DEA itself.

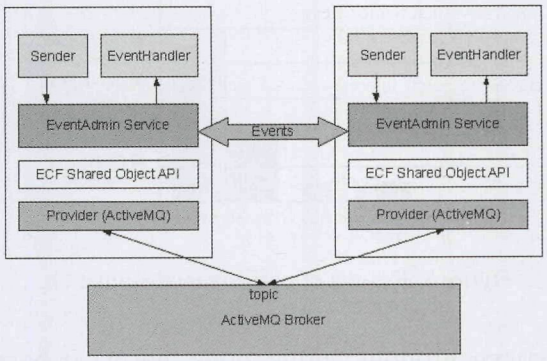

Figure 9. Distributed Event Admin.

7. Conclusion

OSGi is a popular Java-based platform, which has its roots in the area of embedded systems. However, nowadays it is used more and more in enterprise systems. To fit this new application area, OSGi has recently been extended with the Remote Services specification. This specification enables distribution, which OSGi was previously lacking. However, the specification supports synchronous communication only and leaves out asynchronous communication. As an attempt to fill a gap in this field, we proposed, implemented and evaluated an approach for the integration of asynchronous messaging into OSGi.

Our approach has a number of advantages. First of all, it provides seamless integration of asynchronous messaging into OSGi. Besides being relatively non-invasive, our approach guarantees reliable message delivery and is a natural fit for OSGi. Furthermore, our approach allows OSGi still to be used in embedded systems. Originally, OSGi was targeted towards embedded systems, which have limited memory and processing power compared to enterprise systems. Therefore, it is important for OSGi to remain applicable to embedded systems after asynchronous messaging (needed for enterprise systems) has been integrated into it.

8. Acknowledgments

Irina Astrova 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). Irina Astrova and Ahto Kalja's work was also supported by the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12.

References

[1] G. HOHPE, B. WOOLF, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] M. SCHAAF, V. AHLERS, A. KOSCHEL, I. ASTROVA, A. KALJA, D. BOSSCHAERT, R. ROELOFSEN, Integrating Asynchronous Communication into the OSGI Service Platform. *Proc. of 7th Intl. Conf. on Web Information Systems and Technologies*, (2011 May 6-9); Noordwijkerhout, Netherlands. pp. 165–168.

[3] OSGI ALLIANCE, 2009. OSGi Service Platform Service Compendium – Release 4, version 4.2. `http://www.osgi.org/Release4/` `[14/02/2012]`

[4] ASYNCHRONOUS MESSAGING WITH SPRING JMS. `http://onjava.com/onjava/2006/02/22/asynchronous-messaging-with-spring-jms.html [14/02/2012]`

[5] R. MONSON-HAEFEL, D. CHAPPELL, *Java Message Service*. O'Reilly & Associates, Inc., 2001.

[6] ECLIPSE FOUNDATION, 2009. Distributed Event Admin Service. `http://wiki.eclipse.org/Distributed_EventAdmin_Service` `[14/02/2012]`

Received: June, 2012
Accepted: August, 2012

Contact addresses:

Arne Koschel
Faculty IV
Department of Computer Science
University of Applied Sciences and Arts
120 Ricklinger Stadtweg
30459 Hannover
Germany
e-mail: arne.koschel@fh-hannover.de

Irina Astrova
Institute of Cybernetics
Tallinn University of Technology
21 Akadeemia tee
12618 Tallinn
Estonia
e-mail: irina@cs.ioc.ee

Marc Schaaf
Institute for Information Systems
University of Applied Sciences Northwestern Switzerland
Olten
Switzerland
e-mail: marc.schaaf@fhnw.ch

Volker Ahlers
Faculty IV
Department of Computer Science
University of Applied Sciences and Arts
120 Ricklinger Stadtweg
30459 Hannover
Germany
e-mail: volker.ahlers@fh-hannover.de

Stella Gatziu Grivas
Institute for Information Systems
University of Applied Sciences Northwestern Switzerland
Olten
Switzerland
e-mail: stella.gatziugrivas@fhnw.ch

Ahto Kalja
Institute of Cybernetics
Tallinn University of Technology
21 Akadeemia tee
12618 Tallinn
Estonia
e-mail: ahto@cs.ioc.ee

ARNE KOSCHEL is a professor of distributed systems at the Faculty of Business and Computer Science of the University of Applied Sciences and Arts, Hannover. He has a long industry and research experience in distributed, large scale information systems and middleware in general. His current research interests include cloud computing, SOA, event processing and messaging, middleware and distributed applications.

IRINA ASTROVA is a Senior Researcher at Software Department of the Institute of Cybernetics at Tallinn University of Technology, Tallinn, Estonia. She has a long research experience in ontologies and databases. Her current research interests include cloud computing and semantic web.

MARC SCHAAF is a scientifc staff member at the Institute for Information Systems, University of Applied Sciences Northwestern Switzerland, Olten, Switzerland. He holds an MsSc in applied computer science and is working towards a PhD. His current research interests include event processing, messaging, and cloud computing.

VOLKER AHLERS is a professor of simulation and mathematics at the Faculty of Business and Computer Science of University of Applied Sciences and Arts, Hannover. He has a long research experience in dynamic systems, computer aided geometry, and image analysis. His current research interests include computer graphics, visualization, embedded systems, and bioinformatics.

STELLA GATZIU GRIVAS is a professor of database and information systems at the Institute for Information Systems, University of Applied Sciences Northwestern Switzerland, Olten, Switzerland. Her current research interests include databases, distributed information systems, event processing and cloud computing.

AHTO KALJA is a Professor of systems programming at Software Department of Institute of Cybernetics at Tallinn University of Technology, Tallinn, Estonia. He has a long research experience in CAD, artificial intelligence and databases. His current research interests include e-Government solutions and e-Services.