Distributed OSGi through Apache CXF and Web Services

Irina Astrova
Institute of Cybernetics
Tallinn University of Technology
Tallinn, Estonia
irina@cs.ioc.ee

Arne Koschel
Faculty IV, Department for Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
akoschel@acm.org

***Abstract*—The OSGi Service Platform supports rudimentary distribution through Universal Plug and Play (UPnP) specification, which facilitates interaction with UPnP-enabled consumer devices. The main goal of UPnP is to allow simple and seamless connection between devices and sharing of those devices. Although UPnP can be seen as a distributed system, the range of its use is very limited. Yet the way how OSGi services interact to each other is constrained to a single Java Virtual Machine (JVM) where they run. This prevents to provide OSGi services in a distributed manner. Therefore, the main goal of this paper is to add distribution capability to OSGi, without having to change OSGi itself. The contribution of this paper is twofold: (1) it supplies implementation details to show how OSGi can be extended with distribution; and (2) it implements a flight information system to show how this extension can be applied to business applications.**

Keywords—OSGi Service Platform; Web services; Apache CXF; distribution; flight information system.

I. Introduction

The OSGi Service Platform [1] is an emerging successful Java-based standard for developing component-based software. As its core, OSGi is about bundles and services. Bundles provide modularization and encapsulation for components. A bundle is also a deployable unit, which can be installed and removed at runtime. Bundles can register services, which can be looked up in the service registry and then used by other bundles. OSGi can be deployed on a wide range of devices from sensor nodes, home appliances, vehicles to high-end servers.

One of the weaknesses of OSGi (addressed in this paper) is that it defines how services "talk" to each other from within a single Java Virtual Machine (JVM). Thus, the way how the services interact to each other is constrained to an OSGi container where they run (local communication). In today's IT world, distributed systems have been used rapidly in business applications. Thus, a lack of support of distribution is a severe hindrance for further use of OSGi in business applications because it does not allow external systems to access OSGi services remotely [7]. This holds true especially for enterprise systems, as nowadays OSGi grows more and more from its original roots (viz., embedded systems) into a Java platform for enterprise system. Simple evidence of this fact is that OSGi's Embedded Systems Expert Group was discontinued, while its counterpart – OSGi's Enterprise Systems Expert Group – is still "alive and kicking". Therefore, our main goal was to add distribution capability to OSGi in order for OSGi to be more applicable to enterprise systems. Toward this goal, in our previous paper [13] we proposed an approach, where distribution is enabled by exposing OSGi services as Web services, which is done by creating a "middleware" bundle that adapts OSGi services to become Web services. Figure 1 gives an overview of our approach.

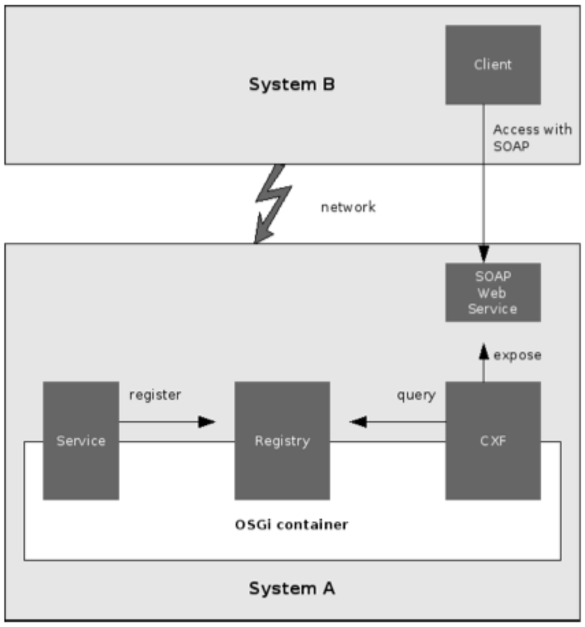


Figure 1. Architecture of our approach [13]. (System B is an external system.)

As middleware, we selected Apache CXF [2]. CXF is an enterprise service bus (ESB) that helps to develop services using different frontend programming APIs on different protocols, which in their turn use Web services defined by WSDL contract with SOAP bindings over HTTP. CXF is divided into multiple units called endpoints. We selected CXF for providing OSGi with distribution capability because being open source, CXF has no license fee. Furthermore, CXF itself can be represented as a set of bundles.

The rest of the paper is organized as follows. In Section II we provide an overview of the related work. In Sections III – VII we supply details on our approach. Given the background from the previous sections, in Section VIII we give an example of how to use our approach. In Section IX we present the implementation of our approach. In Section X we make conclusions and outline the future work.

II. Related Work

A number of extensions – e.g., OpenSOA, Redistributable OSGi (R-OSGi) [6], Distributed OSGi (D-OSGi), IBM Lotus Expeditor, Eclipse Communication Framework and Newton Framework – were done to allow services to "talk" with each other across multiple JVMs. The goal of all these extensions was to add distribution capability to OSGi, thus enabling a service running in one "local" OSGi container to invoke a service running in another, potentially remote, OSGi container. While meanwhile distribution has become part of D-OSGi, it lacks features like asynchronous messaging. This is, however, possible with our approach by utilizing both the appropriate CXF features and the extension that we implemented.

There also exists J2ME Web Service Specification [8], which extends OSGi with Web services functionality. This specification is widely used in embedded systems. Embedded systems are losing their original meaning, which referred to small computational isolated (stand-alone) systems that give functional support for devices that do not fit to the definition of a computer [9]. Today we can define an embedded system as a micro-processed device, thus programmable, which uses its computing power for a specific purpose [10]. However, the scope of the specification includes only how to expose remote services. It is not specified how external systems can access the services from embedded systems.

This paper extends our previous paper [13] in the following ways:

- It supplies more details on our approach (see Sections III, IV and VI).
- It shows an example of using our approach (see Section VIII).
- It implements a flight information system to demonstrate how our approach can be used in business applications (see Section IX).

III. Registering OSGi Services

Since a CXF bundle and an OSGi bundle exporting Web services are completely decoupled, we cannot rely on their installation order. In particular, the CXF bundle can be installed both before and after the OSGi bundle. Therefore, there are two cases to consider.

First, when the CXF bundle is installed before the OSGi bundle, the CXF bundle can register a service listener to get notified when the OSGi bundle is installed. This listener can be associated with a filter expression.

Figure 2 shows an example of how to register a service listener. Here `context` is an instance of the `BundleContext` class; it is provided during the CXF bundle activation. As can be seen, the `addServiceListener` method of the `BundleContext` class takes two parameters. The first parameter is an instance of a class implementing the `ServiceListener` interface; this instance will be used by the CXF bundle as a callback to create an endpoint. The second parameter is a filter expression, which specifies that the CXF bundle gets notified about the service registration if and only if OSGi services have the `expose.service` property set to `true`.

```
BundleContext context = . . . ;
context.addServiceListener                     (cxfListener,
"(expose.service=true)");
```

Figure 2. Registering a service listener, which listens to Web services.

The service listener can use a `ServiceReference` instance to have more information about the OSGi services. In both situations, the `ServiceReference` instance can be used to acquire the necessary service information, e.g., the name of the service interface used by the service registry and an instance of the class implementing the specified interface. Figure 3 shows how to fetch this information.

```
ServiceReference ref = . . . ;
Class iface = (Class) ref.getProperty("expose.interface");
String url = (String) ref.getProperty("expose.url");
Object instance = context.getService ( ref );
```

Figure 3. Accessing service properties.

Second, when the CXF bundle is installed after the OSGi bundle, the CXF bundle can query the service registry for OSGi services. Figure 4 shows an example of such a query. As can be seen, the `getServiceReferences` method of the `BundleContext` class takes two parameters. The second parameter is again the filter expression, whereas the first parameter specifies the service interface. However, since the CXF bundle cannot know the service interface in advance, a value `null` is passed to the method to specify that all (registered) OSGi services should be matched against the filter expression.

```
ServiceReference[] refs = context.getServiceReferences(
null , "(expose.service=true)");
```

Figure 4. Querying for Web services.

IV. Exposing OSGi Services as Web Services

As said above, since the CXF bundle can be installed both after or before the OSGi bundle as well as removed after a certain time, we have to enforce a loose coupling between the two bundles but still enable communication between them. There are three approaches to this:

- Extender model
- Listener model
- Whiteboard pattern (which we follow).

A. *Extender Model*

In the extender model, the CXF bundle can register a service using a `BundleContext` instance to get informed when new bundles are installed. The CXF bundle can react on these events and inspect the content of the new bundles. The same approach can be used by the OSGi bundle to inform the CXF bindle about Web services it wants to export. CXF could define the location of a configuration file. This file would contain the Web services and the necessary service properties like a service interface, a port number and a URL.

Figure 5 shows an example of such a configuration file. The CXF bundle would check if the OSGi bundle contains this file and then use the file during the configuration.

```
<service
name="ExampleService"
interface="com.example.ExampleService"
class="com.example.ExampleServiceImpl"/>
<soap/>
<http port="8080" cont ext="/exampleService" />
</service>
```

Figure 5. Configuration file.

B. *Listener Model*

In the listener model, the CXF bundle itself can register a service that provides the necessary functionality for exporting Web services. For example, this service could have a method `exportService`; the service information like a service interface, a port number and a URL would be passed to that method. The OSGi bundle would fetch the CXF service from the service registry and call the method with the corresponding parameters.

C. *Whiteboard Pattern*

With the Whiteboard pattern [3], the OSGi bundle can register itself Web services. In addition to the service registration, the OSGi bundle can provide additional service properties (e.g., `expose.interface`, `expose.port` and `expose.url`), which contain the necessary configuration information. The CXF bundle can use the service registry to fetch the Web services.

D. *Evaluation of Approaches*

The extender model requires that all configuration information to be specified in a configuration file. It is therefore necessary that all this information is available when the OSGi bundle is created. The OSGi bundle is not able to provide additional information or change existing one during the deployment. Moreover, all (exported) Web services have to be listed in the configuration file. Thus, bundles are not able to inform the CXF bundle about new services after the deployment. Therefore, we rejected this approach.

The Whiteboard pattern has been originally developed as an alternative to the listener model. Both can be used for the configuration of the OSGi and CXF bundles. Both ensure that the bundles are completely decoupled from a specific CXF API. No dependencies on CXF classes or packages exist and the bundles do not need to be linked at compile time. However, the extender model analyzes the content of the bundles and uses the bundle events to enable export of new services, whereas the Whiteboard pattern uses the OSGi Service Layer to enable communication between the bundles.

Moreover, due to the dynamic nature of OSGi (i.e., due to the fact that bundles can be removed at runtime), bundles cannot assume the continuous existence of the CXF service. Rather, they need to monitor the service registry in order to check for the CXF service.

When using the Whiteboard pattern, the configuration information is embedded into the service properties. The OSGi bundle can change these properties at runtime. That is, the OSGi bundle can change the configuration at deployment time and afterwards when needed. If the OSGi bundle wants to stop some service from being exported, it can just remove that service from the service registry and thereby inform the CXF bundle. The OSGi bundle is therefore able to control the export at runtime. Furthermore, the service properties can be used as filters when the service registry gets browsed by the CXF bundle for exported services. For example, the CXF bundle could query for all OSGi services that need to be exported with SOAP or at a specified port. This querying is, however, not possible when the configuration information is embedded into a configuration file as it is done in the listener model. Therefore, we also rejected the listener model and selected the Whiteboard pattern instead.

V. Configuring Web Services

As said above, we decided to use the Whiteboard pattern for extending OSGi with distribution capability. When using this pattern, the configuration mechanism relies on embedding the necessary configuration information into the service properties while registering a Web service. Since we did not want to change the code, we decided to provide the configuration information during the service registration. There are three approaches to this:

- Java properties
- Declarative services Specification
- Spring-OSGi (which we follow).

A. *Java Properties*

Figure 6 shows an example of how to register a Web service using Java properties and associate the Web service with the service properties during this registration. The CXF bundle will listen to OSGi services that have a property `expose.service=true`. The service properties are stored in `java.util.Dictionary`, which is then passed as a parameter during the service registration.

```
ExampleService service = new ExampleServiceImpl();
Dictionary dict = new Hashtable();
dict.put ("expose.service", true) ;
dict.put ("expose.interface", ExampleService.class);
dict.put ("expose.url", "http://localhost:8080/exampleService");
```

Figure 6. Registering a Web service using Java properties [13].

B. *Declarative Services Specification*

The intention of Declarative Services Specification (DSS) [11] is to ease the use of the OSGi Service Layer.

Figure 7 shows a code example of how to register a Web service using DSS.

```
<?xml version=“1.0” encoding=”UTF−8”?>
<component name=“ExampleService”>
<implementation
class=“com.example.ExampleServiceImpl”/>
<property name=“expose.service”>true</property>
<property name=“expose.interface”>
com.example.ExampleService
</property>
<property name=“expose.url”>
http://localhost:8080/exampleService
</property>
<service>
<provide interface=“com.example.ExampleService”/>
</service>
<component>
```

Figure 7. Registering a Web service using Declarative services Specification [13].

C. Spring-OSGi

This approach is similar to DSS. The key difference is that Spring-OSGi [12] itself defines a service registry like OSGi does. This registry is managed by `BeanFactory`. In particular, OSGi services can be searched and registered by `BeanFactory`; `BeanFactory` also makes services applicable to dependency injection. Figure 8 shows a code example of how to register a Web service using Spring-OSGi. Here `exampleService` is a normal Spring bean that will act as the instance of the service. An XML element `<osgi:service>` publishes this service in the service registry by referencing it and embeds additional information like the necessary configuration for CXF into the service.

```
<?xml version=“1.0” encoding=“UTF−8”?>
<bean id=“exampleService”
class=“com.example.ExampleServiceImpl”/>
<osgi:service ref=“exampleService”>
<osgi:interfaces>
<value>com.example.ExampleService</value>
</osgi:interfaces>
<osgi:service−properties>
<prop key=“expose.service”>true</prop>
<prop key=“expose.interface”>
com.example.ExampleService
</prop>
<prop key=“expose.url”>
http://localhost:8080/exampleService
</prop>
</osgi:service−properties>
</osgi:service>
```

Figure 8. Registering a Web service using Spring-OSGi [13].

D. Evaluation of Approaches

The use of Java properties is the simplest approach; it embeds the configuration information directly into Java classes. Another advantage of this approach is that it allows for full control over the service properties and thus, it can be used if, e. g., some values need to be calculated at runtime. However, because of the dynamic nature of bundles, the sate of the services have to be tracked all the time. Therefore, we rejected this approach.

Instead of “hard-coding” the necessary logic for the service registration and then the tracking of the service state, DSS allows us to define this declaratively. Bundles that want to publish or use Web services define their intention in a configuration file that is then processed by DSS. However, DSS can be viewed as a hybrid approach that combines the Whiteboard pattern with the extender model. Since the configuration get supplied through the extender model, DSS inherits all the drawbacks of the extender model. Therefore, we also rejected this approach and selected Spring-OSGi instead.

VI. Analyzing Service Interfaces

The information required by CXF is necessary to export a Web service registered in the service registry. This information is either specified as the service properties during the service registration or deduced from the service interface. The frontend that should be used by CXF (either JAX-WS or simple) can be determined by checking if the service interface is annotated. For example, when using the JAX-WS frontend, the service interface is annotated with `WebService`. Figure 9 shows an example of how to determine if this annotation is present.

```
Annotation [] as = iface.getAnnotations ();
for ( Annotation a : as )
{
    if (a.annotationType().equals(WebService.class))
       { // use JAX−WS frontend
    }
}
```

Figure 9. Using annotations.

In addition, CXF supports the use of different data binding frameworks. As with the frontend, the data binding used by CXF (either JAXB or Aegis) can be determined by checking if the classes passed as method parameters are annotated.

VII. Accessing Web Services with Client from External Systems

CXF is divided into multiple units called endpoints. The communication over a network often takes place in a heterogeneous environment where some endpoints may share the same set of technologies whereas others may use a different set. When using CXF, the class libraries to use different technologies are located at different machines. Therefore, when services want to change the used technologies, only the libraries on the corresponding machines have to be updated. Other services are not affected.

To create an endpoint, CXF can fetch the configuration information from the service properties or deduce it from the service interface. Figure 10 shows how this information can be passed to CXF. The `ServerFactoryBean` class is used to configure an endpoint. The service interface, the URL and the service instance are passed to a `ServerFactoryBean` instance. The `getFactory` method of the `ServerFactoryBean` class returns either a `JaxWsServerFactoryBean` or `ServerFactoryBean` instance depending on the frontend (either JAX-WS or simple). Similarly, the `getDataBinding` method returns either a `JAXBDataBinding` or `AegisDatabinding` instance depending on the data binding (either JAXB or Aegis).

```
ServerFactoryBean factory = getFactory(frontend);
factory.setServiceClass (( Class ) iface );
factory.setAddress ( url );
factory.getServiceFactory(). setDataBinding(
getDataBinding(databinding));
factory.setServiceBean (instance);
Server server = factory.create ();
```

Figure 10. Passing configuration information.

After fetching all necessary configuration information (i.e., a service interface, a port number and a URL), CXF becomes responsible for creating an endpoint based on this information using a method `create` of a class `Sever`. The endpoint is published at the specified URL and can be accessed using SOAP [4].

In addition to exposing OSGi services to external systems, these systems require to access endpoints. There are two approaches to accessing OSGi services remotely:

- Creating a remote API
- Creating a proxy (which we follow).

A. *Creating Remote API*

To enable bundles to access (remote) endpoints, CXF could offer a special API that encapsulates the necessary logic. A service that is to be used by the bundles would then be registered in the service registry. Figure 11 shows a code example of how to create a remote API.

```
RemoteEndpoint re=new RemoteEndpoint();
re.setAddress(url);
re.setDataBinding(getDataBinding(databinding));
Object result=re.callMethod(“methodname”, “parameter”);
```

Figure 11. Creating a remote API [13].

B. *Creating Proxies*

CXF provides a class `ClientProxyFactoryBean` to create a proxy [5]. This proxy will implement the service interface. Thus, it can be casted to a variable declared as an instance of the service interface. The proxy forwards method calls to the (remote) endpoint, waits for the result, and passes the result to the caller. This way we have both an elegant and a flexible way to access Web services remotely.

Figure 12 shows a code example of how to create a proxy. The information about the service interface, the URL and the data binding are passed to this proxy.

```
ClientProxyFactoryBean factory=new
ClientProxyFactoryBean();
factory.setServiceClass(iface);
factory.setAddress(url);
factory.getServiceFactory().setDataBinding( getDataBinding(d
atabinding));
Object proxy = factory.create();
Dictionary props = . . .;
context.registerService(serviceClass.getName(), proxy,
props);
```

Figure 12. Creating a proxy [13].

After the proxy has been created, it gets registered in the service registry. During this registration, additional service properties can be attached to the proxy. These properties can be used as a filter expression when bundles are querying the service registry for specific services. Figure 13 shows an example of a query that returns all services that represent endpoints located on a server `www.company.com`.

```
ServiceReference[] refs = context.getServiceReferences(null,
”(&(service.is_remote=true)
(service.host=www.company.com))”);
```

Figure 13. Querying for specific services.

C. *Evaluation of Approaches*

Creating a remote API is simple. However, this approach introduces several drawbacks. After the bundle has been removed, all the information specified in the service interface is lost. This interface defines which methods are available and which parameters the methods require. Hence, the service interface represents the contract between a server and a client. By embedding the method name and parameters into a method call, we have to ensure that the specified values conform to the service interface. Moreover, depending on the used frontend, the service interface may specify additional semantics. For example, some methods may use asynchronous call semantics and require the registration of a callback method. However, the remote API would have to be aware of all these possibilities. Therefore, we rejected this approach and selected to create a proxy instead.

VIII. Example

To demonstrate our approach, let us consider `SimpleService.`

Figure 14 shows an example of the service interface. `SimpleService` will be accessed by a client through the service interface, which is by definition separate from the service implementation. This separation enables changing the service implementation without changing other services.

```
package com.xyz.simple;
public interface SimpleService {
    public String getName();
}
```

Figure 14. SimpleService interface.

Figure 15 shows an example of the service implementation.

```
package com.xyz.simple;
public class SimpleServiceImpl implements SimpleService {
    public String getName() {
            return "My Name is SimpleService";
    }
}
```

Figure 15. SimpleService implementation.

Figure 16 shows an example of an OSGi bundle, which exposes `SimpleService` as a Web service using the Java properties approach (see Section V).

```
package com.xyz.simple;
import org.osgi.framework.BundleActivator;
public class SimpleBundleActivator implements
BundleActivator {
    public void start(BundleContext context) throws Exception
{
            System.out.println("Starting SimpleBundle");
            SimpleService service = new SimpleServiceImpl();
            java.util.Properties props = new Properties();
            props.put("expose.service", true);
            props.put("expose.interface", SimpleService.class);
            props.put("expose.url",
                    "http://localhost:8080/simpleservice");
            context.registerService(
            SimpleService.class.getName(), service, props);
    }
    public void stop(BundleContext context) throws Exception
{
            System.out.println("Stopping SimpleBundle");
    }
}
```

Figure 16. OSGi bundle.

Figure 17 shows an example of the client (i.e., a CXF bundle, which accesses `SimpleService`). The client listens to `SimpleService` at the endpoint URI: "http://localhost:8080/simpleservice".

```
public class SimpleBundleActivator implements
BundleActivator {
   public void start(BundleContext context) throws Exception {
      CxfOsgiUtils.proxyRemoteEndpoint(context,
                                          SimpleService.class,
"http://localhost:8080/simpleservice");
      ServiceReference ref = context.getServiceReference(
            SimpleService.class.getName());
      SimpleService service =
                    (SimpleService) context.getService(ref);
   }
}
```

Figure 17. CXF bundle.

IX. Flight Information System

To prove the feasibility of our approach, we implemented a flight information system (FIS) using our approach. The purpose of the FIS is to inform passengers waiting at the airport about flight changes made by airlines. Figure 18 gives an overview of the FIS.

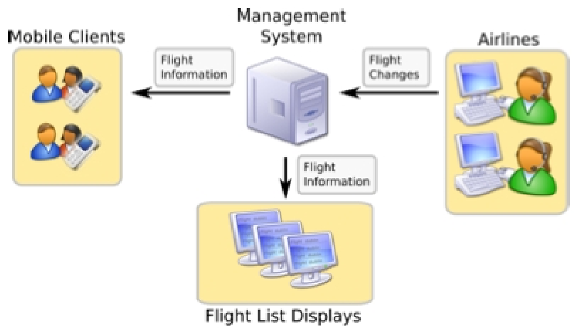
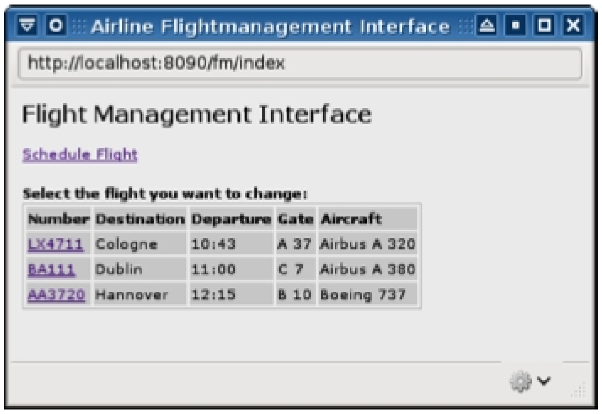


Figure 18. Architecture of flight information system.

The FIS consists of the following components:

- Management system
- Airline interface
- Flight list display system
- Mobile client.

The airline interface lets airlines to change information about scheduled flights. For example, an airline can change the aircraft type to a smaller one if the currently scheduled plane cannot be filled or the airline can change the departure time if the currently scheduled plane is delayed due to weather conditions. All these changes are broadcasted to the flight list display system and the mobile client by the management system. Figure 19 shows GUI of the airline interface.



http://localhost:8090/fm/index

Flight Management Interface

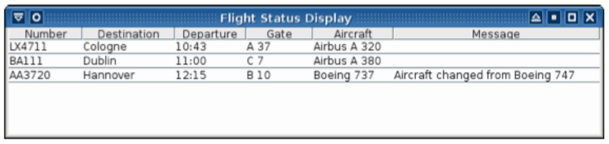
Schedule Flight

Select the flight you want to change:

Number	Destination	Departure	Gate	Aircraft
LX4711	Cologne	10:43	A 37	Airbus A 320
BA111	Dublin	11:00	C 7	Airbus A 380
AA3720	Hannover	12:15	B 10	Boeing 737

Figure 19. GUI of airline interface.

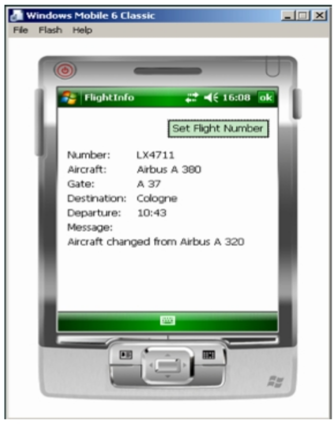
The flight list display system sends updated flight information (including current departure time, gate, aircraft type and flight status: on-time, delayed or cancelled) to flight list displays, which are spread over the airport. Figure 20 shows GUI of the flight list display system.



Number	Destination	Departure	Gate	Aircraft	Message
LX4711	Cologne	10:43	A 37	Airbus A 320	
BA111	Dublin	11:00	C 7	Airbus A 380	
AA3720	Hannover	12:15	B 10	Boeing 737	Aircraft changed from Boeing 747

Figure 20. GUI of flight list display system.

In addition, the mobile client lets passengers receive updated flight information from the comfort of their mobile phones. Figure 21 shows GUI of the mobile client.



Number: LX4711
Aircraft: Airbus A 380
Gate: A 37
Destination: Cologne
Departure: 10:43
Message:
Aircraft changed from Airbus A 320

Figure 21. GUI of mobile client.

As can be seen, the central component of the FIS is the management system. This component dispatches flight changes it receives from the airline interface to all interested parties – i.e., the flight list display system and the mobile client. The management system was implemented as a Java Business Integration (JBI) application. All other components around the management system – i.e., the airline interface, the fight list display system and the mobile client – were implemented as OSGi bundles. Therefore, the FIS is served as an example of how OSGi containers can communicate with an external (non-OSGi) system in a distributed manner.

X. Conclusion and Future Work

We have proposed an approach to extending OSGi with distribution. Our approach does not require making changes to OSGi. Rather, it encourages using the concept of bundles.

OSGi was originally targeted towards embedded systems. Therefore, the main benefit of adding distribution capability to OSGi is to make OSGi more applicable to enterprise systems, which usually require remote communication.

We have added distribution capability to OSGi using CXF. CXF enables external systems to invoke (registered) OSGi services. Bundles can define which of the services can be accessed by external systems. CXF is used to create (remote) endpoints and helps to utilize different technologies when needed. Our approach ensures a loose coupling between CXF and OSGi bundles. OSGi services do not need to be aware of the distribution or how they are exposed to external systems. The only needed interface is the OSGi Service Layer. Furthermore, bundles can access (remote) endpoints. This access is fully transparent to the bundles. A proxy delegates to the endpoints and gets registered in the service registry in order to be used by the bundles. In addition, we have used our approach to implement the flight information system.

In the future, we are going to make performance evaluation of our approach.

Acknowledgment

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

References

[1] OSGi – The Dynamic Module System for Java, http://www.osgi.org, last access: September 21, 2012.

[2] Apache Software Foundation: CXF pages, http://cxf.apache.org, last access: September 21, 2012.

[3] Kriens, P., Hargrave, B.J.: Whiteboard pattern, http://www.osgi.org/documents/osgi_technology/whiteboard.pdf, last access: September 21, 2012.

[4] World Wide Web Consortium. SOAP Version 1.2. http://www.w3.org/TR/soap12, last access: September 21, 2012.

[5] Gamma et al., Design Patterns, Addison-Wesley, 1995.

[6] Swiss Federal Institute of Technology (ETH) Zurich: R-OSGi pages, http://r-osgi.sourceforge.net, last access: September 21, 2012.

[7] Astrova, I., Koschel, A., Roelofsen, R., Kalja, A.: Evaluation of the Applicability of the OSGi Service Platform to Future In-Vehicle Embedded Systems. In Proceedings of the 2nd International Conferences on Advanced Service Computing (SERVICE COMPUTATION), IARIA, pp. 202–207, 2010.

[8] Sun Microsystems. J2ME Web Service Specification. http://jcp.org/en/jsr/ detail?id=172, last access: September 21, 2012.

[9] Janecek, J.: Efficient soap processing in embedded systems. In Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS), pp. 128–135, 2004.

[10] Wolf, W.: Computer as Components: principles of embedded computing system design. Morgan Kaufmann, 2001.

[11] Cervantes, H., Hall, R.: Automating Service Dependency Management in a Service-Oriented Component Model, http://www.osgi.org/wiki/uploads/Links/AutoServDependencyMgmt_byHall_Cervantes.pdf, last access: September 21, 2012.

[12] Interface21 Inc: Spring-OSGi, http://www.springframework.org/osgi, last access: September 21, 2012.

[13] Roelofsen, R., Bosschaert, D., Ahlers, V., Koschel, A., Astrova, I.: Think large, act small: An approach to Web Services for embedded systems based on the OSGi framework. In Proceedings of the 1st International Conference (IESS), pp. 239–253, 2010.