Verteilte Dienste mit R-OSGi

Aus der Entfernung

Ralf Vandenhouten, Thomas Kistel

OSGi hat sich in vielen Anwendungsbereichen als flexibles Komponenten-Framework etabliert. Damit entwickelte Module bieten Dienste an und können von anderen Modulen genutzt werden. Mit der Middleware R-OSGi lassen sich OSGi-Komponenten auch über Systemgrenzen hinweg verteilen.

Wenige Softwarekonzepte haben eine solch unauffällige und gleichzeitig nachhaltige Erfolgsgeschichte vorzuweisen wie das Framework der vor 10 Jahren gegründeten OSGi Alliance (früher Open Services Gateway Initiative, www.osgi.org). Während ungezählte Hypes am IT-Horizont erschienen und meist schnell wieder untergingen, verrichtete OSGi unter der Motorhaube von Kraftfahrzeugen, in der Industrie- und Gebäudeautomation oder in Mobilfunkgeräten zuverlässig seinen Dienst. Bei eingebetteten Systemen gilt OSGi heute als einer der führenden Softwarestandards. Als Basis der populären Entwicklungsumgebung Eclipse, die seit Version 3.0 aus OSGi-Komponenten besteht, hat sich die OSGi-Technik inzwischen auch als stabiles Fundament einer Rich Client Platform bewährt und damit nachgewiesen, dass ihr Geltungsanspruch deutlich über Embedded-Systeme hinausgeht.

Sein Erfolg liegt in der Einfachheit und Erweiterbarkeit des Frameworks begründet. Das Komponentenmodell sieht vor, Software in leichtgewichtigen Modulen zu entwickeln. Die OSGi-Plattform steuert den gesamten Lebenszyklus dieser sogenannten Bundles: Sie erlaubt es, Bundles zur Laufzeit zu installieren, zu starten, anzuhalten oder zu entfernen. Das ist insbesondere für Systeme wichtig, die rund um die Uhr laufen müssen.

Minimalinvasiver Eingriff

Bundles können Standarddienste des OSGi-Frameworks oder von anderen Bundles bereitgestellte Dienste nutzen (siehe Abbildung 1). Um einen solchen Service zur Verfügung zu stellen, muss ein Bundle ein entsprechendes Java-Interface beim Framework registrieren. OSGi stellt keine besonderen Anforderungen an das Interface; jedes POJO (Plain Old Java Object) ist dafür geeignet. Möchte ein Bundle einen bestimmten Service nutzen, kann es bei der Service-Registry des Frameworks dessen Verfügbarkeit erfragen und sich Referenzen auf die vorhandenen Instanzen geben lassen.

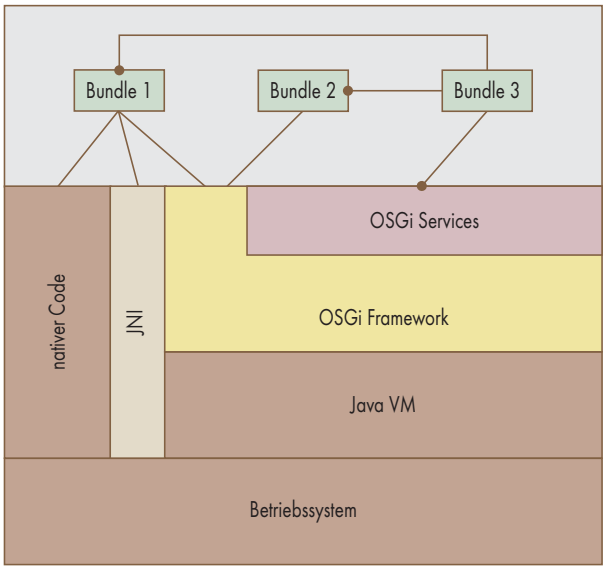
Sein flexibler, serviceorientierter Ansatz macht eine der Stärken von OSGi aus. Es ist jedoch noch keine serviceorientierte Architektur im Sinne des heute oft verwendeten Begriffs SOA, da bei OSGi sämtliche Bundles in derselben virtuellen Maschine und somit auf demselben Host laufen. Zwar gibt es zahlreiche OSGi-Services, die Netzkommunikation unterstützen und über ihre jeweiligen Protokolle – etwa HTTP, FTP oder SOAP (Simple Object Access Protocol) – aus der Ferne ansprechbar sind. Orts- und Zugriffstransparenz von Serviceschnittstellen jedoch, wie man sie bei einer SOA zur Verteilung von Unternehmensanwendungen erwartet, sind damit noch nicht realisiert. Erst die im September 2009 veröffentlichte OSGi-Spezifikation 4.2 sieht mit Distributed OSGi eine transparente Verteilung von Diensten im Netz mit einer geeigneten Middleware vor (siehe Kasten „Distributed OSGi, ECF und R-OSGi“). Das am Institut für Informations- und Kommunikationssysteme der ETH Zürich entwickelte R-OSGi hingegen arbeitet auch mit älteren OSGi-Containern.

Seine Zielsetzung ist, für beliebige OSGi-Implementierungen einen transparenten, verteilten Zugriff auf die Services zu ermöglichen und dabei folgende Anforderungen zu erfüllen [1]:

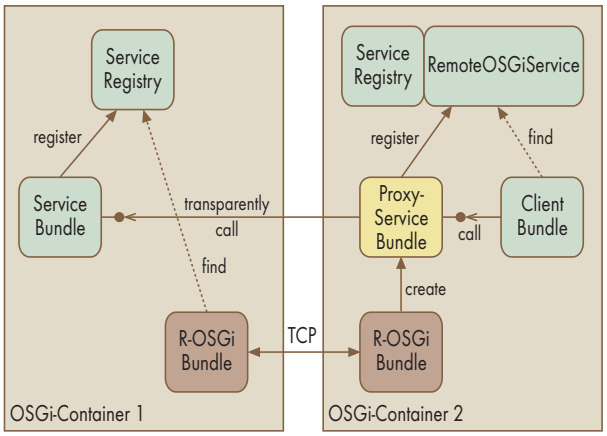
– Nahtlose Einbettung in OSGi: Aus Sicht eines Bundles sollen lokale und entfernte Dienste nicht unterscheidbar sein. Existierende OSGi-Anwendungen sollen ohne Anpassungen verteilbar sein.
– Zuverlässigkeit: Anwendungsentwickler sollen nicht mit neuen Fehlermustern konfrontiert werden, sondern vom Netz verursachte Fehler wie übliche Dienstausfälle behandeln können.
– Allgemeingültigkeit: R-OSGi beschränkt sich nicht auf bestimmte Dienste. Jeder gültige OSGi-Service soll remote verfügbar sein können.
– Portabilität: Die Middleware soll auf kleinen Embedded-Geräten lauffähig sein, etwa auf Mobiltelefonen. Der mit R-OSGi verbundene Ressourcenbedarf muss deshalb niedrig sein.
– Adaptivität: Die Verwendung von R-OSGi ist nicht verbunden mit Rollenzuordnungen wie Client oder Server. Beziehungen zwischen Modulen im verteilten Kontext sind grundsätzlich symmetrisch.
– Performance: R-OSGi soll schnell sein, vergleichbar mit der optimierten

RMI-Implementierung (Remote Method Invocation) von Java 5.

Umgesetzt ist R-OSGi als gewöhnliches OSGi-Bundle, das man auf allen OSGi-Containern starten muss, die am verteilten Szenario beteiligt sind. R-OSGi sucht anschließend jeweils im lokalen Container nach Services, für die die Property *RemoteOSGIService.R_OSGi_REGISTRATION* gesetzt ist, und informiert seine Gegenstücke über das Ergebnis. Auf der entfernten Seite erzeugt und registriert R-OSGi für jeden gefundenen Dienst ein dynamisches Proxy-Bundle, das die Schnittstelle des ursprünglichen Dienstes nachbildet. Es leitet alle Client-Aufrufe über den entsprechenden R-OSGi-Kanal an den „echten" Service und liefert dessen Rückgabewert zurück an den Client (siehe Abbildung 2). Der Client kann nicht feststellen, ob ein Dienst in seinem eigenen Container oder in der Ferne geleistet wird. Mit einer sogenannten *SurrogateRegistration* lassen sich sogar existierende Services verteilen, die ursprünglich nicht dafür vorgesehen waren.



Bundles nutzen die Dienste des OSGi-Frameworks und solche, die von anderen Bundles registriert wurden (Abb. 1).



Clients erhalten lokalen Zugriff auf einen von R-OSGi erzeugten transparenten Proxy-Service, der Aufrufe an den eigentlichen Dienst weiterleitet (Abb. 2).

Anders als bei Distributed OSGi haben sich die Entwickler von R-OSGi entschieden, die Client-Seite nicht vollkommen agnostisch gegenüber dem Netz zu gestalten. Ein Client „sieht" nur die Services, mit deren Container sich das Client-Bundle zuvor per R-OSGi verbunden hat. Das kann entweder direkt durch explizite Angabe eines URI erfolgen oder mit einem Discovery Service wie SLP (Service Location Protocol), der im Netz nach Anbietern sucht. Die Referenz auf den entfernten Dienst wird auch nicht wie beim lokalen Zugriff vom *BundleContext* geliefert, sondern vom *RemoteOSGiService* der Middleware. Dem größeren Aufwand für die Konfiguration, der mit diesem Konzept verbunden ist, steht eine stärkere Kontrolle der Verdrahtung gegenüber, die sich bei komplexen Szenarien positiv auf die Skalierbarkeit auswirken kann.

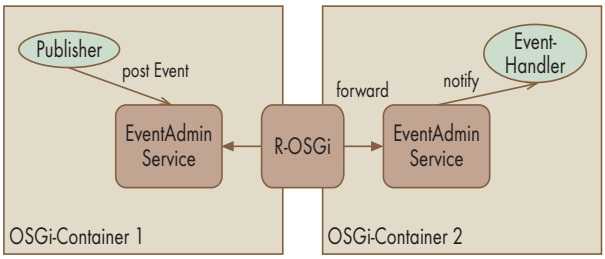
Generell stehen Anwendungen in verteilten Systemen vor größeren Herausforderungen als solche, die in nur einem Adressraum ablaufen, selbst wenn sie einem transparenten Verteilungsmodell folgen. Das liegt an den Schwierigkeiten, die durch miteinander vernetzte Rechnerknoten hinzukommen können. Dazu gehören Unterbrechungen der Netzverbindung oder der Ausfall einzelner Knoten, aber auch höhere Antwortzeiten durch Latenzen, Verlust von Nachrichten oder ein nichtdeterministisches Verhalten des Gesamtsystems. Diese Phänomene tauchen grundsätzlich auf und lassen sich durch R-OSGi nicht beseitigen.

Allerdings dürfen sich OSGi-Bundles ohnehin nicht auf die Verfügbarkeit von Diensten verlassen, da der Nutzer diese jederzeit anhalten oder sogar deinstallieren kann. Clients müssen mit solchen unvorhersehbaren Ausfällen rechnen und umgehen können. Dabei helfen in der Regel *ServiceTracker* oder Komponentenklassen im Fall von Declarative Services. R-OSGi nutzt das aus, indem es Fehler im Zusammenhang mit der Verteilung, etwa eine Netzunterbrechung, als Dienstausfall ansieht und das zugehörige Proxy-Bundle deinstalliert. Das Client-Bundle muss für einen solchen Fall auch bei lokalem Zugriff einen Plan B parat haben. Über die Service-Verteilung im Netz muss es hingegen nichts wissen, sodass OSGi-Entwickler nicht umlernen müssen.

OSGi-Bundles können Java-Pakete aus anderen Bundles verwenden, wenn sie deren Import explizit in ihrem Bundle-Manifest deklarieren. Das hat Konsequenzen für die Generierung von Proxy-Bundles. Es kommt vor, dass die Service-Schnittstelle Klassen als Methodenparameter oder Rückgabewerte verwendet, die nicht zum Java-Standard gehören und auf der Client-Seite nicht bekannt sind. Da der Service-Proxy sie jedoch im Client-Container benötigt, stellt R-OSGi sie dort per Type Injection bereit. Dafür analysiert das R-OSGi-

iX-TRACT

- OSGi gilt als einer der führenden Softwarestandards im Embedded-Umfeld und schickt sich an, auch andere Bereiche der Softwareentwicklung zu erobern.
- Die R-OSGi-Middleware erweitert beliebige OSGi-Implementierungen um einen transparenten, verteilten Zugriff auf Services.
- Entwickler müssen nicht viel Neues lernen, aber beim verteilten Entwurf Disziplin walten lassen.



Lokale *EventAdmin*-Services leiten Ereignisse an andere Container weiter (Abb. 3).

Listing 1: Schnittstelle des Bibliotheksdienstes

```
public interface ILibraryService {
    public List<Book> getBookList();
    public List<Book> getSortedBookList();
    public List<Book> getBookList(long userId);
    public boolean isAvailable(long bookId);
    public boolean lendBook(long userId, long bookId);
    public void returnBook(long bookId);
}
```

Bundle auf der Serviceseite die Schnittstelle und trägt alle direkt oder indirekt benötigten Klassen in eine Injection-Liste ein, die es mit dem Service registriert. Sobald ein Client den Service anfordert, erzeugt R-OSGi die erforderlichen Klassen im Proxy-Bundle. Für die dynamische Proxy-Generierung bedient sich R-OSGi der ausgereiften Open-Source-Bibliothek ASM des ObjectWeb-Konsortiums [g]. ASM ist ein Framework zur Manipulation, Analyse und Generierung von Java-Bytecode, das in vielen anderen Bibliotheken und Frameworks zum Einsatz kommt.

Übertragungskanal für Ereignisse

Kaum eine Anwendung kommt ohne asynchrone Benachrichtigungen aus. In Stand-alone-Applikationen verwendet man dafür meist das Beobachtermuster. Im OSGi-Kontext ist es nicht zu empfehlen, wenn es über Bundle-Grenzen hinweg zum Einsatz kommen soll, da es den dynamischen Lebenszyklus von Bundles nicht berücksichtigt und deshalb Schwierigkeiten verursachen kann [2]. Noch schlimmer sieht es im Fall von R-OSGi aus: In einer verteilten Umgebung müssen alle Fernaufrufe als Call-by-Value (im Gegensatz zum Java-Standard Call-by-Reference) ausgeführt werden. Mit Referenzen kann die Gegenstelle nichts anfangen, deshalb muss R-OSGi Kopien der Objekte übertragen. Würde sich ein Beobachter bei seinem entfernten Subjekt registrieren, würde dieses nur eine Referenz auf eine Kopie des Beobachters in seinem Adressraum speichern. Eine Benachrichtigung ginge daher nur an die Beobachter-Kopie, nicht an den ursprünglichen Beobachter im anderen Container, da die Objektkopien nicht mit ihren Originalen synchronisiert sind.

Abhilfe schafft der *EventAdmin*-Service. Es handelt sich um einen Standardservice der OSGi-Compendium-Spezifikation, der asynchrone und synchrone Benachrichtigungen zwischen beliebigen Bundles ermöglicht. Ein Sender von Benachrichtigungen legt eine Rubrik dafür fest, das sogenannte Topic. Wer diese Nachrichten bekommen möchte, muss einen Event Handler für das gewünschte Topic registrieren. Dieses Muster funktioniert auch mit R-OSGi, sogar ohne besondere Vorkehrungen. R-OSGi setzt den *EventAdmin*-Service im Netz um, sodass Nachrichten eines Containers die anderen Container erreichen (siehe Abbildung 3). Dafür muss lediglich auf jedem Container eine *EventAdmin*-Implementierung laufen.

Für die Datenübertragung im Netz verwendet R-OSGi ein eigenes Protokoll über sogenannte *NetworkChannel*. Zwar kommt standardmäßig eine TCP-

Distributed OSGi, ECF und R-OSGi

Die jüngst im Service Compendium der OSGi Service Platform von der OSGi Alliance veröffentlichte Spezifikation zu Remote Services beschreibt eine Erweiterung des Standards für verteilte OSGi-Anwendungen (Distributed OSGi). Bei dessen Spezifikation wurde bewusst darauf geachtet, keine zusätzliche Middleware-Lösung zu schaffen, da es dafür bereits eine Reihe ausgereifter Konzepte gibt. Vielmehr definiert Distributed OSGi (D-OSGi) eine Möglichkeit, existierende Middleware-Lösungen wie SOAP, CORBA, JMS oder R-OSGi auf standardisierte Art und Weise in das OSGi-Framework zu integrieren und dabei die bewährten Konzepte der Service-Konsumption beizubehalten.

Für die Verteilung von Services über eine vorhandene Middleware sind bei D-OSGi die Distribution Provider zuständig. Dafür stellen sie einen Endpoint bereit, über den sich Services importieren und exportieren lassen. Der Endpoint ist dabei der Zugriffspunkt der Service-Kommunikation über lokale OSGi-Grenzen hinweg. Ein OSGi-Framework kann mehrere Distribution Provider gleichzeitig nutzen, die unterschiedliche Services importieren und exportieren. Mit diesem Konzept lässt sich beispielsweise eine Anwendung für Buchbestellungen entwickeln, welche für den Abruf der Buchkataloge einen Distribution Provider für Webservices nutzt und für die Bestellabwicklung einen anderen Provider.

Import und Export von Remote Services erfolgen bei D-OSGi über Remote Service Properties, die man beim Registrieren und Referenzieren von Services nutzen kann. Mit sogenannten Intents lassen sich bestimmte Anforderungen definieren, die die entfernten Dienste unterstützen müssen. Das Konzept der Intents stammt von der SCA Policy Framework Specification [a]. Es erlaubt es unter anderem, Angaben zur Verschlüsselung oder Dienstgüte der Übertragung des Distribution Provider zu machen.

Apache CXF [b] und das Eclipse Communication Framework (ECF) [c] sind Implementierungen eines Distribution Provider für D-OSGi. ECF ist ein Framework für die Entwicklung von verteilten Eclipse-Anwendungen, das asynchrone Punkt-zu-Punkt oder Publish-and-Subscribe-Kommunikation unterstützt. Die Installation von ECF in die Eclipse-Entwicklungsumgebung ermöglicht Echtzeitkommunikation und stellt Teamfunktionen wie das gemeinsame Editieren von Quellcode bereit. Grundlage dafür ist eine Reihe von APIs und Frameworks, die auf bestehenden Protokollen wie XMPP, IRC, BitTorrent oder MSN aufbauen. Jedes dieser Protokolle ist eine Implementierung eines Communication Containers, die bei ECF den Zugriff auf einen protokollspezifischen Kontext gestatten. ECF liefert außerdem einen Communication Container für das an der ETH Zürich entwickelte R-OSGi [d].

R-OSGi ist keine direkte Implementierung für D-OSGi, sondern stellt eine Middleware bereit, mit der sich OSGi-Services ebenfalls remote zur Verfügung stellen und abrufen lassen. Bereits existierende Services kann man bei R-OSGi über eine *Surrogate-Registration* exportieren. R-OSGi kann mit älteren OSGi-Frameworks arbeiten, sofern die Anwendung die Ereignisbenachrichtigung durch den *EventAdmin*-Service nicht benötigt. Für OSGi R3-Implementierungen, die *EventAdmin* benötigen, kann der Entwickler auf den Backport zurückgreifen, der Teil der OSGi-Implementierung Concierge [e] ist.

Implementierung (*TCPChannel*) zum Einsatz, man kann sie jedoch leicht austauschen. Auf diese Weise lassen sich Kanäle mit Bluetooth oder dem Mina-Framework realisieren, was beispielsweise die drahtlose R-OSGi-Anbindung von Mobiltelefonen ermöglicht.

Schlanke Implementierung

R-OSGi steht als 130 bis 145 KByte große JAR-Datei zum Download bereit, je nachdem, ob die ASM-Bibliothek enthalten ist oder nicht. Der Nutzer muss es als gewöhnliches OSGi-Bundle namens *ch.ethz.iks.r_osgi.remote* gemeinsam mit den Bundles der Anwendung installieren, und zwar auf Service- und Client-Seite. Wie eine solche Anwendung aussehen kann, demonstriert das Beispiel einer Bücherei. Die vollständigen Projekte und Quelltexte sind auf dem *iX*-Listingserver zu finden.

Listing 1 zeigt das Interface des Bibliotheksdienstes, das Methoden für unsortierte und sortierte Bücherlisten sowie für den Leihvorgang vorsieht. Es greift dabei auf die einfache Klasse *Book* zurück, die lediglich die Buchattribute wie ID, Titel und Autor sowie die zugehörigen Getter-Methoden enthält. Sie ist gemeinsam mit der Schnittstelle *ILibraryService* in einem separaten Bundle namens *de.thwildau.tm.library.api* untergebracht, das die API definiert und sowohl auf der Client- als auch auf der Serviceseite zum Einsatz kommt. Das Service-Bundle *de.thwildau.tm.library.service* implementiert die Schnittstelle *ILibraryService* in der Klasse *LibraryServiceImpl*. Ausschnitte sind in Listing 2 zu sehen.

Listing 2: Einfache Implementierung des Büchereidienstes

```
public class LibraryServiceImpl
implements ILibraryService {
    private List<Book> bookList = new Vector<Book>();
    private HashMap<Long,Long> lendMap
          = new HashMap<Long,Long>();

    public LibraryServiceImpl() {
        // Fuege einige Buecher in die bookList ein
        ...
    }

    public List<Book> getBookList() {
        return bookList;
    }
    ...

    public boolean isAvailable(long bookId)
    {
        return lendMap.containsKey(bookId)
               && lendMap.get(bookId)==null;
    }
    ... usw.
}
```

Listing 4: Deklarative Aktivierung des Dienstes

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
     immediate="true" name="de.thwildau.tm.library.service">
   <implementation class="de.thwildau.tm.library.service.LibraryServiceImpl"/>
   <property name="service.remote.registration" type="Boolean" value="true"/>
   <property name="service.remote.smartproxy" type="String"
             value="de.thwildau.tm.library.service.SmartLibraryService"/>
   <service>
      <provide interface="de.thwildau.tm.library.api.ILibraryService"/>
   </service>
   <reference bind="bindEvent" cardinality="1..1"
              interface="org.osgi.service.event.EventAdmin"
              name="EventAdmin" policy="static" unbind="unbindEvent"/>
</scr:component>
```

Eine Besonderheit von R-OSGi ist die Möglichkeit, sogenannte Smart-Services zu definieren. Ein Smart-Service ist eine abstrakte Klasse, die nur einen Teil der Schnittstellenmethoden implementiert und die übrigen abstrakt lässt. Sobald der Client den Dienst anfordert, überträgt R-OSGi die Smart-Klasse zum Client-Container, wo sie als Proxy für den Dienst arbeitet. Implementierte Methoden führt die Smart-Klasse auf dem Client aus, Aufrufe abstrakter Methoden leitet R-OSGi an den echten Serverdienst weiter. Auf diese Weise kann das Service-Bundle einen Teil der Last auf die Clients abwälzen – im Beispielprogramm die Bereitstellung der sortierten Bücherliste. Die Klasse *SmartLibraryService* greift dabei sowohl mit *getBookList()* auf den Remote-Service zurück als auch auf die Utility-Klasse *BookSorter*, die im Bundle *de.thwildau.tm.library.util* zu finden ist (siehe Listing 3). Letzteres muss nicht auf dem Client installiert und nicht einmal dort bekannt sein. R-OSGi überträgt es per Type Injection zur Laufzeit automatisch vom Server- zum Client-Container.

Der Beispieldienst stellt nicht nur die API bereit, sondern publiziert darüber hinaus mithilfe des *EventAdmin*-Service alle 60 Sekunden das „Buch der Minute“, indem es in einem separaten Thread ein Event-Objekt mit der Property *bookId* versieht und es per *eventAdmin.postEvent(event)* an Event Handler verschickt, die dem Topic *service/library/BOOKOFTHEMINUTE* lauschen. Aktivieren lässt sich der Server am einfachsten als Declarative Service mit der XML-Datei in Listing 4. Die Property *service.remote.registration* teilt R-OSGi mit, dass es sich um einen verteilbaren Dienst handelt, während *service.remote.smartproxy* die Smart-Klasse festlegt. Der Client lässt sich deklarativ aktivieren, indem man mit

Listing 3: Sortierung im Smart-Service

```
public abstract class SmartLibraryService
implements ILibraryService
{
    public List<Book> getSortedBookList() {
        BookSorter sorter =
            new BookSorter(getBookList());
        return sorter.getSortedList();
    }
}
```

Listing 5: Zugriff auf den entfernten Dienst

```
public void lendSelectedBook() {
  ...
  long bookId = selectedLibraryBook.getId();

  boolean lendSuccess =
      libraryService.lendBook(userId, bookId);

  if (lendSuccess) {
      borrowedBooks.add(selectedLibraryBook);
  } else {
      statusField.setText("Book not available!");
  }
  ...
}
```

```
<reference bind="bind" cardinality="0..1"
 interface="ch.ethz.iks.r_osgi.RemoteOSGiService"
 name="RemoteOSGiService" policy="static"
 unbind="unbind"/>
```

den *RemoteOSGiService* referenziert und ihn in der Methode *bind()* an die Variable *remote* bindet. Die Referenz auf den entfernten Bibliotheksdienst erhält die Client-Komponente dann mit

```
RemoteServiceReference[] refs =
   remote.getRemoteServiceReferences(uri,
      ILibraryService.class.getName(), null);
ILibraryService libraryService =
(ILibraryService)remote.getRemoteService(refs[0]);
```

Anschließend lässt sich der Dienst wie ein lokales Objekt verwenden. Listing 5 illustriert das am Beispiel der Methode, die ein Buch an einen Nutzer verleiht. Für das asynchron gesendete Buch der Minute muss das Bundle noch den Event Handler registrieren:

```
Hashtable props = new Hashtable();
props.put(EventConstants.EVENT_TOPIC,
   new String[] { "service/library/ 7
                          BOOKOFTHEMINUTE" });
eventReg = componentContext.
   getBundleContext().registerService(
      EventHandler.class.getName(), this, props);
```

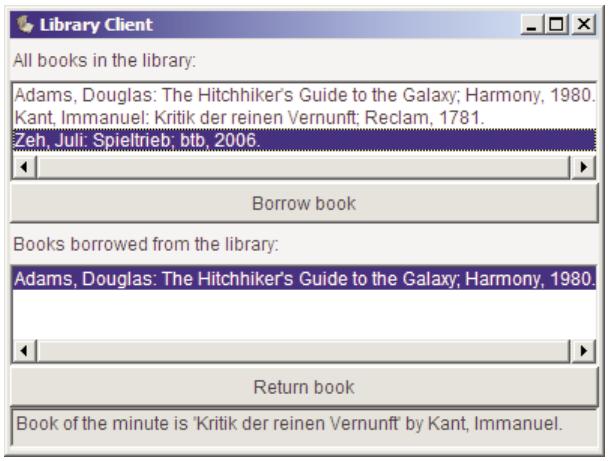
Implementiert ist der Event Handler durch die Methode *handleEvent*:

```
public void handleEvent(Event event) {
   long bookId = (Long)event.getProperty("bookId");
   frame.displayBookOfTheMinute(bookId);
}
```

Das Beispielprojekt demonstriert die Client-Anwendung in einem grafischen Fenster, mit dem man interaktiv auf

Onlinequellen

[a] SCA
www.oasis-open.org/committees/sca-policy
[b] Apache CXF
cxf.apache.org/distributed-osgi.html
[c] ECF
www.eclipse.org/ecf
[d] R-OSGi
r-osgi.sourceforge.net
[e] Concierge
concierge.sourceforge.net
[f] OSGi
www.osgi.org
[g] ASM
asm.ow2.org
[h] BUG
www.buglabs.net



Mit dem Client kann der Nutzer Bücher ausleihen oder zurückgeben und sieht den Bestand der Bibliothek. Die Statuszeile zeigt das aktuelle „Buch der Minute" an (Abb. 4).

den entfernten Bücherdienst zugreifen kann (siehe Abbildung 4).

Inzwischen setzen verschiedene kommerzielle und nichtkommerzielle Anwendungen R-OSGi ein, etwa die Collaborative Middleware flowSGI für mobile Endgeräte oder das Baukastensystem BUG zum Entwickeln von Embedded Devices [h]. Die ixellence GmbH verwendet R-OSGi ebenfalls bei der Entwicklung ihrer verteilten Softwareprodukte. Im medizinischen Bereich überträgt das Telemonitoring-System ixTrend hochabgetastete medizinische Echtzeitdaten via R-OSGi. Auch umfangreiche Konfigurations- und Steuerungsprozesse der verteilten medizinischen Netzkomponenten sind als Remote-Services mit R-OSGi realisiert. In der Videoüberwachungssoftware ixCam übernimmt R-OSGi die Alarmierung und überträgt die Echtzeitbilddaten von den Kameras zu den Client-Stationen. Die Autoren haben den aktuellen Release Candidate 4 der R-OSGi-Version 1.0.0 in den genannten Projekten bereits als zuverlässig, leistungsfähig – vergleichbar mit RMI – und durchaus tauglich für den Produktiveinsatz erlebt. Leider werden jedoch nicht alle Fragen und Fehlerberichte im zugehörigen Forum kurzfristig beantwortet.

Fazit

Wer bereits mit OSGi vertraut ist, erhält mit R-OSGi eine schlanke und leistungsfähige Middleware, mit der er die verteilte Softwarewelt erobern kann, ohne umdenken zu müssen. Eine Einarbeitung ist dennoch erforderlich, nicht zuletzt wegen der Java-unüblichen Call-by-Value-Semantik. R-OSGi beeinflusst letztlich auch die Architektur der Anwendung, denn in verteilten Anwendungen wird der Grundsatz, Schnittstellen einfach und klein zu halten, zur leistungsentscheidenden Notwendigkeit.

OSGi-Puristen mögen bemängeln, dass die Transparenz der Verteilung bei R-OSGi nicht vollkommen ist, weil man entfernte Dienste über den speziellen *RemoteOSGiService* anfordern muss. Es muss sich noch zeigen, wie sich die ersten Implementierungen der OSGi-Spezifikation 4.2 in Bezug auf Distributed OSGi bewähren und welchen Einfluss das auf die Konsolidierung der konkurrierenden Ansätze haben wird. (mr)

Installation des Beispielprojekts

Wer das Bibliotheksprojekt ausprobieren möchte, sollte zunächst Eclipse 3.5 (Galileo Release) installieren. Weiterhin benötigt man das Eclipse Communication Framework (nachinstallierbar über die Galileo Update Site unter Help->Install New Software) sowie das Bundle *org.eclipse.equinox.event*, das unter anderem im Equinox Project SDK enthalten ist. Letzteres ist ebenfalls auf der Galileo Update Site zu finden. Entwickler, die mit SLP experimentieren möchten, müssen außerdem noch das Bundle *service_discovery.slp* von der R-OSGi-Website ins *dropins*-Verzeichnis der Eclipse-Installation kopieren.

Das Beispiel besteht aus vier Eclipse-Projekten, die man auspacken und in die Eclipse-Workbench importieren muss. Nach dem Öffnen der Dateien *library-service.product* im Projekt *de.thwildau.tm.library.service* für den Server sowie *library-client.product* im Projekt *de.thwildau.m.library.client* für den Client lassen sich beide durch Klick auf den grünen Startpfeil in der jeweiligen Produktdatei starten. Sollen Server und Client nicht auf demselben Rechner laufen, muss der Nutzer vorher die Serveradresse in der Property *ch.ethz.iks.r_osgi.service.uri* anpassen. Wer ein Multiuser-Szenario simulieren möchte, kann den Client auch mehrfach starten und dabei jeweils die Property *de.thwildau.tm.library.userid* in der Datei *library-client.product* ändern.

RALF VANDENHOUTEN

leitet den Studiengang Telematik an der Technischen Hochschule Wildau und beschäftigt sich mit komponentenbasierter Softwareentwicklung in verteilten Systemen, Mobile Computing und Bildverarbeitung.

THOMAS KISTEL

ist Geschäftsführer der ixellence GmbH, hat zahlreiche OSGi-Projekte geleitet und interessiert sich für modellgetriebene Softwareentwicklung.

Literatur

[1] Jan S. Rellermeyer, Gustavo Alonso, Timothy Roscoe; R-OSGi: Distributed Applications through Software Modularization; Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, 2007

[2] Peter Kriens, B.J. Hargrave; Listeners Considered Harmful: The Whiteboard Pattern; OSGi Alliance 2004; www.osgi.org/wiki/uploads/Links/whiteboard.pdf

www.ix.de/ix0912142