SERVICES IM KLEINEN MAßSTAB: SOA FÜR EINGEBETTETE SYSTEME

Web services and service-oriented architectures are usually associated with large distributed systems. Scenarios using these technologies, however, also exist in the world of small embedded systems. We present a web service solution for embedded systems that is based on the OSGi framework. At the heart of our approach lies an adaptation of the Apache CXF enterprise service bus to embedded Java platforms.The article introduces a number of background technologies and concepts such as Service Oriented Architecture (SOA), Web Services and OSGi. It also outlines the available Java technology in the embedded space. In addition a number of use-cases are shown, which validate the use of SOA and distribution technologies in this context.Technically the article describes a reference project which was undertaken to realize service distribution using Web Services for embedded devices. It utilizes Apache CXF and OSGi to achieve this goal. The reader is presented with a number of problems that had to be addressed in order to achieve this and a walk-through of how this can be realized in a non-intrusive way.
Finally the article examines practical deployment issues and experiences. It outlines the steps needed to run a Java 5-based product like CXF on an embedded Java platform, which is often limited to older versions of Java and therefore lacking key language constructs such as annotations, without making changes to the code.

Einleitung

Wenn es um die Integration großer verteilter Software-Systeme geht, kommen seit einigen Jahren in stetig wachsendem Umfang Web Services und Serviceorientierte Architekturen (SOA) zum Einsatz. Mittlerweile existieren mehrere entsprechende Standardplattformen [7, 10].

Eingebettete Systeme (*embedded systems*) finden sich heutzutage in einer Vielzahl kleiner mobiler Geräte wieder. Für den Einsatz von Web Services in eingebetteten Systemen spricht, dass Web Services es erlauben, Anwendungen genau dann zur Verfügung zu stellen, wenn sie benötigt werden. Dies stellt angesichts der begrenzten Speicherressourcen von eingebetteten Systemen einen großen Vorteil gegenüber dauerhaft installierten Anwendungen dar.

Die begrenzten Speicher- und auch Leistungsressourcen von eingebetteten Systemen erweisen sich jedoch als Problem, wenn es an die Umsetzung von SOA-Standardplattformen auf derartige Systeme geht. Der Speicherbedarf gängiger Plattformen übersteigt häufig die gegebenen Möglichkeiten. Darüber hinaus basieren die meisten SOA-Plattformen auf Java. Für eingebettete Systeme existieren jedoch üblicherweise keine vollständigen Implementierungen der Java Standard Edition (Java SE) oder gar der Java Enterprise Edition (Java EE). Die verfügbaren Java Micro Editions (Java ME) und ähnliche Lösungen bringen weitere Einschränkungen mit sich.

Szenarien für den Einsatz von Web Services und SOA

Um den vorgestellten Ansatz konkreter zu machen, bedienen wir uns zweier Szenarien für den Einsatz von Web Services und SOA in eingebetteten Systemen: einem Fahrzeug-Tracking-Service und einem Anzeigen-Service für Autofahrer. Beide Szenarien nutzen ein hypothetisches eingebettetes Device (im Folgenden „Fahrzeug-Device“ genannt), das in einem Fahrzeug installiert ist und mit der Umgebung kommunizieren kann – z.B. über UMTS. In einer praktischen Implementierung sollten die genannten Lösungen unabhängig von einem bestimmten Fahrzeughersteller sein und gängige Standards wie SOAP berücksichtigen.

Fahrzeug-Tracking-Service

Ein Fahrzeug-Tracking-Service erlaubt Autovermietungen die Positionen ihrer Fahrzeuge zu verfolgen. Dazu stellt das Fahrzeug-Device einen Web Service zur Verfügung, der die aktuellen GPS-Koordinaten des Fahrzeugs und sonstige erforderliche Daten weitergibt. Eine Web-Anwendung der Autovermietung kann den Service nutzen und die Fahrzeugposition über ein Web-Frontend auf einer Landkarte anzeigen (**s. Abb. 1 a**).

die autoren

Roman Roelofsen (E-Mail: roman.roelofsen@googlemail.com)
Roman Roelofsen is currently completing his Master of Science in Computer Science (from FH Hannover) at IONA Technologies. His main interests are software architecture and language design.

David Bosschaert (E-Mail: david.bosschaert@iona.com)
is a Principal Engineer at IONA Technologies in Dublin. He has been developing software since 1983 and coding Java since 1997. He spends most of his time developing Enterprise Java products for IONA and Open Source contributions for Eclipse STP. David currently represents IONA in the OSGi Enterprise Expert Group.

Arne Koschel (E-Mail: akoschel@acm.org)
ist Prof. für Verteilte Systeme an der FH Hannover und nebenberuflich Berater. Er hat langjährige Industrieerfahrung als Berater, Projektleiter und Produkt-Manager in Themen wie Java EE, Middleware, SOA, SOA Governance bis hin zu Technologien wie Java ME und JAIN SLEE.

Volker Ahlers (E-Mail: volker.ahlers@fh-hannover.de)
ist Professor für Simulation und Mathematik an der Fakultät Wirtschaft und Informatik der Fachhochschule Hannover. Er besitzt langjährige Erfahrung auf den Gebieten Scientific Computing und Bildverarbeitung.

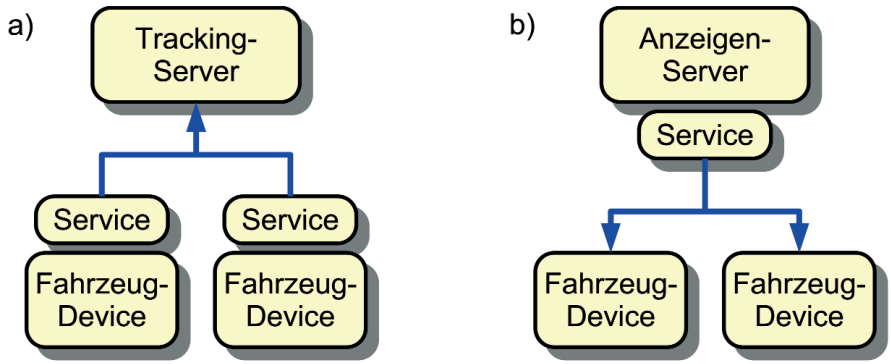

Abbildung 1: Szenarien für den Einsatz von Web Services und SOA in eingebetteten Systemen: Fahrzeug-Tracking-Service (a) und Anzeigen-Service für Autofahrer (b).

Da das Fahrzeug die benötigten Daten selbst in einem Standardformat zur Verfügung stellt, ist kein zentraler Server mit Daten aller Fahrzeuge erforderlich. Stattdessen kann der Fahrzeug-Tracking-Service lokal verteilt genutzt werden. Des Weiteren kann der Web Service auf einfache Weise erweitert werden, um kundenspezifische Zusatzinformationen zu übermitteln.

Anzeigen-Service für Autofahrer

Ein Anzeigen-Service übermittelt einem Autofahrer, der z.B. an einer Ampel wartet, Werbung und lokale Informationen, die auf nahegelegene Tankstellen, Geschäfte, Restaurants etc. hinweisen. Der Fahrer kann über das Fahrzeug-Device steuern, welche Art von Werbung und Informationen er erhalten möchte (s. Abb. 1 b).

Durch den Einsatz von Web Services müssen sich die Service-Anbieter nicht auf ein Standardformat für die Anzeigen einigen. Stattdessen werden auf dem Fahrzeug unterschiedliche Service-Nutzer-Clients installiert, wenn das Fahrzeug in ein lokales Netzwerk eintritt. Die Clients können gelöscht werden, wenn das Fahrzeug das Netzwerk wieder verlässt, oder für spätere Anwendungen gespeichert werden.

Basistechnologien

Web Services, Service-Oriented Architecture (SOA), Enterprise Service Bus (ESB)

Nach [20] ist ein Web Service ein *„software system designed to support interoperable machine to machine interaction over a network“* (s.a. [7, 10]). Die Schnittstelle des Software-Systems ist mittels XML beschrieben (typischerweise WSDL) und kommuniziert wird in der Regel mittels XML-Nachrichten nach dem SOAP-Standard.

Web Services sind heutzutage ein vorherrschender Ansatz (keinesfalls der einzige), um Anwendungen an einer SOA teilnehmen zu lassen [7, 10, 11]. In einer SOA kommunizieren Anwendungen über wohl definierte Schnittstellen in oft technisch heterogenen, verteilten Umgebungen. Enterprise Service Bus (ESB) [7, 10] Implementierungen wie das für unsere Untersuchungen genutzte Apache CXF [2], stellen ggf. eine Kommunikationsinfrastruktur für eine SOA bereit. ESBs erlauben meist einen Mix verschiedener Schnittstellen- und Kommunikationstechnologien für eine konkrete SOA einzusetzen. Abbildung 2 zeigt eine Beispiel-SOA technisch, in der ein ESB verschiedene Schnittstellen-, Kommunikations- und Anwendungstechnologien koppelt.

OSGi

OSGi ist eine Java-Container-Technologie, die gleichermaßen für eingebettete Java-ME-Anwendungen wie für Desktop- und Server-Umgebungen genutzt wird. Die Kernelemente von OSGi sind „Bundles“ und „Services“.

Bundles bieten Modularisierung und Kapselung für Komponenten; sie sind ferner eine „Deployment“-Einheit (Software-Bereitstellung) für eine OSGi-Laufzeitumgebung. Bundles können Services registrieren, die von anderen Bundles gesucht und genutzt werden können. De facto war OSGi eine SOA-Grundlage für lokale Maschinen, lange bevor der SOA-Begriff populär wurde. Auch in der aktuellen OSGi-Spezifikation sind Services noch nicht maschinenübergreifend angelegt.

Eine OSGi-Stärke ist die Dynamik der Umgebung. Bundles und Services können zur Laufzeit hinzugefügt oder gelöscht werden, ohne einen Neustart der OSGi-Umgebung zu erfordern. Zudem können problemlos multiple Versionen von Bundles und Services koexistieren; entsprechend mächtig sind die Lebenszyklus- und Versionskonzepte von OSGi.

OSGi existiert seit den späten 90er Jahren. In der heutigen OSGi-Allianz sind ca. 30 Mitglieder; der Stand der Spezifikation ist OSGi 4.1. Es existiert eine Reihe von open source-OSGi-Implementierungen wie Equinox, das als Grundlage der Eclipse IDE dient, Apache Felix und KnopflerFish. Hinzu kommen verschiedene kommerzielle OSGi-Implementierungen.

Eingebettete Systeme

Für unsere Beispielszenarien (Fahrzeug-Tracking, Anzeigen-Service) kommen verschiedene eingebettete Hardware-Plattformen in Frage, z.B. ARM/XScale, PowerPC und Intel x86. Soll eine große Bandbreite von Zielplattformen bedient werden, sind zahlreiche Einschränkungen zu beachten.

Eingebettete Systeme sind gekennzeichnet durch begrenzte Ressourcen, insbesondere im Hinblick auf Performanz und Speicher, an die jede ESB-Lösung angepasst werden muss [3]. Auch wenn die oben

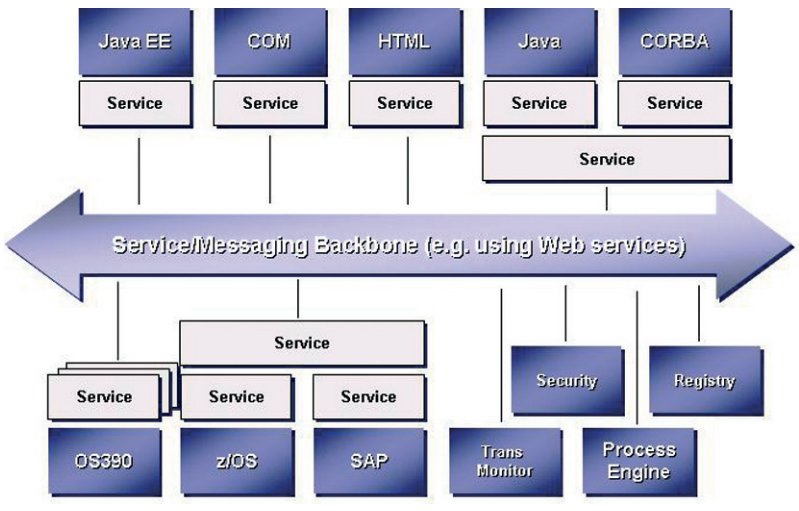

Abbildung 2: Beispiel-SOA technisch

genannten Hardware-Plattformen sich stark voneinander unterscheiden, lassen sich doch einige gemeinsame Randbedingungen für den Einsatz von Java festhalten.

- Die neuesten Bestandteile von Java sind üblicherweise nicht verfügbar. Da zurzeit Java 1.4 die Grundlage von Java ME [16] und Java SE Embedded [17] für ARM-Prozessoren bildet (s. folgenden Abschnitt), können Bestandteile von Java 5 wie Annotations oder Generics auf diesen Plattformen nicht genutzt werden. Java 5 selbst bildet die Grundlage von Java SE Embedded [17] für PowerPC- und x86-Prozessoren sowie JamVM [9], so dass auf diesen Plattformen Bestandteile von Java 6 ausscheiden.
- In allen Embedded-Java-Plattformen ist nur ein Teil der Java-API implementiert.
- Aufgrund der genannten Speicherbeschränkungen muss der Speicherbedarf gängiger ESB-Implementierungen wie CXF substantiell verringert werden.

Java ME, Java SE Embedded

Auf den ersten Blick scheint eine Java-basierte Lösung für unsere deutlich eingeschränkte Zielumgebung schwierig erreichbar zu sein, denn eine typische Java SE hat einen wesentlich höheren „Memory Footprint“ (Speicherbedarf) als für unsere Zielgeräte praktikabel. Jedoch sind zwei Lösungen im Java-Umfeld vorhanden, die hier helfen können:

- Java Micro Edition: Java ME [16] ist eine schlanke Java-Version mit stark eingeschränkter Klassenbibliothek, die speziell „low end“-Geräte adressiert. Die Java ME Connected Limited Device Configuration (CLDC) beginnt mit Geräten ab 128KB RAM (ältere Mobiltelefone). Die Connected Device Configuration (CDC) startet für Geräte mit 2 MB Speicher. Derzeit ist Java ME auf Java 1.4 eingeschränkt.
- Java Standard Edition Embedded: Java SE Embedded [17] ist eine recht neue Sun-Entwicklung, die eine volle Java-SE-Umgebung bereitstellen soll; dies allerdings nur mit 30MB Speicherbedarf für Java 5, weniger als die Hälfte einer normalen Java SE. Konfigurationsoptionen erlauben zudem das Verhältnis dynamischer Java-Speicherallozierungen zu physischem Speicher zu kontrollieren.

Ein ESB für OSGi

Um einen ESB für OSGi bereitzustellen, zeigen bereits unsere Beispielszenarien eine Reihe von Fragestellungen, die zunächst zu beantworten sind. So werden z.B. unterschiedlichste Transporttechnologien benötigt, um die Kommunikation zwischen unabhängigen Parteien zu erlauben. LANs bieten ggf. spezielle Services an, die ein Fahrer bzw. sein Fahrzeug nicht vorab kennen. Selbst wenn die Services erkannt wurden, kann nicht davon ausgegangen werden, dass sie auch in künftigen Fällen verfügbar sind, ihre Verfügbarkeit ist also nicht zuverlässig vorhersagbar. Stellt gar ein Fahrzeug selbst einen Service bereit, so müssen potentielle Konsumenten ebenfall beide entdecken und ggf. nutzen können. Eventuell sind gar mehrere Fahrzeuge nötig, um einen spezifischen Service-Aufruf abzudecken.

In traditionellen „Enterprise“- und „Personal Computer“-Umgebungen werden solche Fragestellungen durch standardisierte Technologien wie z.B. Web Services oder nachrichtenorientierte Middleware behandelt. Deren Ziel ist eine hohe Abstraktion von Basistechnologien, um sich auf SOA-Anwendungen konzentrieren zu können; mächtige ESBs können eine Vielzahl von Verbindungsmöglichkeiten für verschiedenste Szenarien anbieten.

Remote-Fähigkeiten bereitstellen

Um diesen Anforderungen im OSGi-Kontext Herr zu werden, müssen OSGi-Bundles um ein mächtiges „Remoting“-Framework ergänzt werden. In unseren Beispielen nutzen wir Apache CXF und betrachten zwei Möglichkeiten, es in Bundles zu nutzen:

Remote-Fähigkeiten in Bundles einbetten: In OSGi kann jedes Bundle einen eigenen Classpath haben. Das einfachste Verfahren ist also, alle nötigen CXF-Dateien (JARs, Konfiguration) in das Bundle aufzunehmen und dann wie in einer eigenständigen Anwendung zu nutzen. Dies funktioniert für eine einzige Anwendung, die CXF nutzt, gut. Kommen jedoch weitere Anwendungen hinzu, so gibt es Schwierigkeiten, u.a.:

- Speicherverschwendung: Speziell in eingebetteten System ist es nicht akzeptabel, große Bibliotheken mehrfach vorzuhalten. Jedes Bundle würde in dieser Lösung die CXF-Bibliotheken duplizieren und in seinen Classpath aufnehmen.
- Ressourcen-Konflikte: Der Kern von CXF ist der sog. CXFBus. Er kennt alle Erweiterungen wie Transportmechanismen und verantwortet deren Verbindung. Während beispielsweise. ein Web Service erzeugt wird, kreiert der Bus intern eine Jetty-Instanz, die auf einem bestimmten Port lauscht. Hat jedes Bundle einen eigenen CXFBus, so kann dies zu Port-Konflikten führen oder es müssen verschiedene Ports genutzt werden, was die Nutzbarkeit erschwert.
- Klassen-Inkompatibilitäten: Wie erklärt, hat jedes Bundle einen eigenen Classloader; die dadurch geladenen Klassen sind folglich inkompatibel. Ohne explizite zusätzliche Konfiguration kann dann z.B. nicht der CXFBus eines Bundle A einem Bundle B zugewiesen werden, es gäbe Classloader-Exceptions, was wiederum eine lästige Einschränkung darstellt.

CXF als OSGi-Bundle: Naheliegenderweise könnte CXF auch als Menge von OSGi-Bundles angeboten werden. Im Ergebnis gäbe es keine Speicherverschwendung durch Duplikate, keine Ressourcen-Konflikte oder Klasseninkompatibilitäten und nur genau ein CXFBus und ein Classloader sind nötig.

Allerdings müssen dann die verschiedenen Teile zur Laufzeit verbunden werden, z.B. das Bundle, das einen Web Service exportieren will, mit dem CXF-Bundle. Hier bietet OSGi eine Reihe von Optionen sowie ein Programmiermodell, in dem Services und Bundles dynamisch „kommen und gehen“ können. Das CXF-Bundle kann z.B. nach dem Web Services-Bundle installiert oder auch wieder ganz gelöscht werden. Entsprechend muss der einzusetzende Mechanismus sowohl eine lose Kopplung zwischen Bundles zulassen als auch deren Nutzung untereinander.

OSGi-Bundles verbinden

Ein passendes Entwurfsmuster, um Bundles in OSGi zu verbinden, ist das Whiteboard-Entwurfsmuster [8]. Dessen Idee ist wie folgt: Statt einen Service zu suchen und seinen Verfügbarkeitsstatus zu überwachen, registrieren dessen potentielle Anwender sich in der OSGi-Service-Registry. Folglich muss nur der Service all seine Anwender überwachen und nicht umgekehrt.

Ebenfalls nötig ist die Unterscheidung zwischen den Anwendern verschiedener Services. Hier könnte z.B. der Anwender-Service ein bestimmtes Java-Interface im-

plementieren. Sollen Code-Änderungen jedoch vermieden werden, so können passende Properties (Beschreibungsmerkmale) während der Registrierung bereitgestellt werden. Auf Code-Ebene könnten hier java.util.Properties verwendet werden oder auch deklarative Mechanismen für Services und Properties z.B. mittels Spring-OSGi [4]. Der folgende Code-Ausschnitt zeigt die Schritte, die einen Web Service HelloWorldService registrieren und passende Properties setzen:

```
HelloWorldService service = new HelloWorldServiceImpl();
Dictionary<String, Object> dict = new Hashtable<String,
Object>();
dict.put( "cxf.expose", true );
dict.put( "cxf.expose.interface", HelloWorldService.class );
dict.put( "cxf.expose.url", "http://localhost:8080/helloworld" );
context.registerService ( HelloWorldService.class.getName(),
  service, dict );
```

Der context ist eine Instanz von org.osgi.framework.BundleContext und wird während der Aktivierung eines bundle bereitgestellt. Das CXF-Bundle lauscht nun auf Services, deren Property cxf.expose gesetzt ist, die es dann als Web Service exportieren würde. Das Interface HelloWorldService nutzt JAX-WS-Annotationen nach JSR-181 [6], um Informationen zu ergänzen.

Der Prozess um einen Service aus der OSGi registry zu exportieren wird in Abbildung 3 dargestellt. Abhängig vom benutzten Protokoll mit dem der Service exportiert wird, kann der Service mit z.B. einem SOAP Client aufgerufen werden. Abbildung 4 zeigt die schematische Sicht über den Client, OSGi container, CXF und den Service. Je nach Szenario kann das embedded system sowohl das „System A“ oder das „System B“ darstellen. Auf diese Weise sind CXF und der Web Service gänzlich entkoppelt. Nicht einmal die Aktivierungsreihenfolge der Bundles spielt eine Rolle. CXF kann nicht nur nach neuen Services lauschen, sondern auch bereits registrierte Services überprüfen. Zudem hat der Web Service keine Abhängigkeit zum CXF-Bundle. Durch diese Architektur wäre sogar ein Ersatz des CXF-Bundle denkbar.

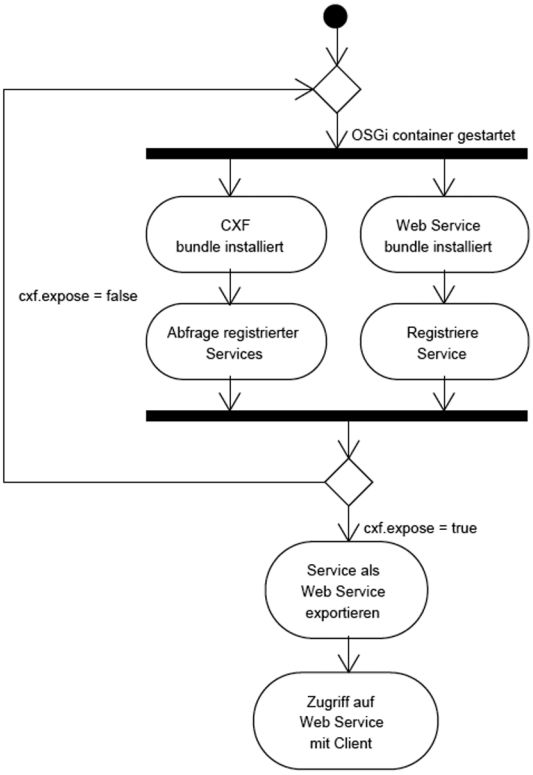

Abbildung 3: OSGi Service als Web Service zur Verfügung stellen

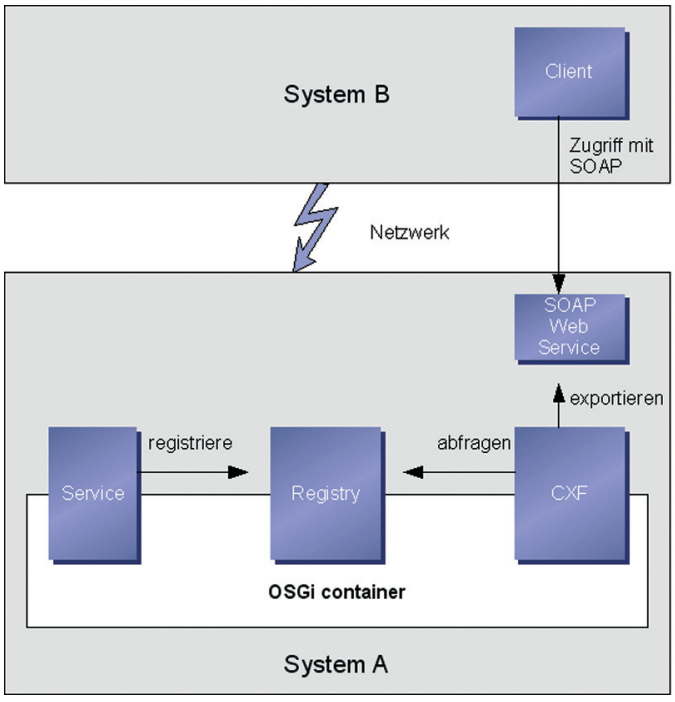

Abbildung 4: Schematische Übersicht der Web Service-Exportierung

Deployment

In der bisher beschriebenen Architektur wurde das Deployment noch nicht berücksichtigt. Während CXF auf Java 5 basiert, kann es je nach Hardware-Plattform erforderlich sein, Java ME CDC zu nutzen, das Java 1.4 als Grundlage hat. Wie oben beschrieben konzentrieren wir uns auf drei unterschiedliche Java-Plattformen:

- Java SE Embedded
- JamVM
- Java ME CDC

Java SE Embedded erfordert keine weiteren Änderungen. Zwar fehlen z.B. Bestandteile wie AWT und Swing, die für CXF aber nicht benötigt werden.

Nach zahlreichen Änderungen an GNU Classpath und CXF waren wir in der Lage, eine eingeschränkte CXF-Version unter

JamVM auszuführen. Diese Änderungen betrafen im Wesentlichen die Pakete java.io und javax.xml.

Sowohl Java SE Embedded als auch JamVM erlauben es, Java-5-Bytecode auszuführen. Die Hauptunterschiede liegen in der Klassenbibliothek. Im Gegensatz dazu basiert Java ME CDC auf Java 1.4 und erlaubt es nicht, Java-5-Bytecode auszuführen. Andererseits hat sich die Bytecode-Spezifikation der virtuellen Maschine zwischen Java 1.4 und Java 5 nicht geändert: Grundsätzlich muss es also doch möglich sein, Java-5-Bytecode auf einer Java-1.4-basierten virtuellen Maschine auszuführen. Dennoch sind einige Änderungen am Bytecode erforderlich, um z.B. Annotations durch Reflections zu simulieren. Für diesen Prozess, der „Bytecode-Weaving" genannt wird, existiert eine Reihe von Werkzeugen [13].

Bytecode-Weaving löst allerdings nur die Probleme, die durch neue Sprachbestandteile hervorgerufen werden. In Java 5 wurden darüber hinaus aber auch neue Klassen eingeführt und bestehende Klassen geändert. Wir nutzen die Klassenbibliothek des Apache-Harmony-Projektes [1], um eine Java-ME-CDC-Laufzeitumgebung mit diesen Klassen auszustatten.

Erfahrungen

Mit Änderungen am Bytecode von CXF und den benötigten Teilen der Apache-Harmony-Bibliothek ist es uns gelungen, eine CXF-basierte Anwendung in einer Java-1.4-Laufzeitumgebung auszuführen. Die betreffende Anwendung definiert Zusatzinformationen über JAX-WS-Annotationen. Dies stellt einen wichtigen Schritt auf dem Weg dar, ein Serverorientiertes Framework wie CXF in einem eingebetteten System zu nutzen: Auf diese Weise vermeiden wir es, CXF in zwei Entwicklungszweige aufzuteilen, und können so von zukünftigen Entwicklungen profitieren.

Unter den verschiedenen Werkzeugen zum Bytecode-Weaving erfüllt *Retrotranslator* alle unsere Anforderungen und erlaubt uns, sämtliche Bestandteile von Java 5 in CXF und unserer Anwendung zu nutzen.

Der Einsatz von JamVM erfordert eine Vielzahl von Änderungen. Die meisten Probleme bereitet dabei die StAX-Implementierung [5], die vom GNU-Classpath-Projekt genutzt wird. Das Deployment unter Java SE Embedded ist dagegen problemlos möglich und erfordert keine Änderungen an CXF.

Verwandte Arbeiten

In der Vergangenheit wurden bereits verschiedene Anstrengungen unternommen, OSGi-Services zu verteilen oder entfernten Zugriff auf OSGi-Komponenten zu ermöglichen. Die OSGi-Spezifikation selbst unterstützt HTTP Service und UPnP(tm). Jini bildete ursprünglich ebenfalls einen Teil der Spezifikation, wurde aber ab Version 4 entfernt.

Jüngere Arbeiten auf diesem Gebiet bilden das R-OSGi-Projekt [14, 19] und das Siemens-OpenSOA-Projekt [15]. Derzeit definiert die *OSGi Enterprise Expert Group* (EEG) [12] Anforderungen und Spezifikationen von Enterprise-Szenarien, die das Thema OSGi in verteilten Systemen einschließen.

Zusammenfassung und Ausblick

Szenarien für den Einsatz von Web Services existieren auch in eingebetteten Systemen. Wir haben einen Ansatz vorgestellt, der auf dem OSGi-Framework basiert. Kern dieses Ansatzes war eine Adaption des CXF Enterprise Service Bus für eingebettete Systeme.

Als Teil dieser Arbeit haben wir CXF in einen Satz von Bundles aufgeteilt. OSGi unterstützt dabei die Modularisierung des ESB, um eine Skalierbarkeit zur Laufzeit zu erreichen. Da nur die tatsächlich benötigten Bundles zur Verfügung gestellt werden müssen, wird kein Speicher verschwendet: Bundles werden erst geladen, wenn ihre Funktionalität benötigt wird, und nach Gebrauch wieder gelöscht.

Das auf Java 5 basierende Java SE Embedded stellt eine zuverlässige Plattform für das Deployment dar. Da die verfügbaren Ressourcen eingebetteter Systeme zunehmen werden, ist zu erwarten, dass Java SE Embedded eine wertvolle Plattform werden wird.

JamVM erlaubt ebenfalls, Bestandteile von Java 5 zu nutzen, ohne den Bytecode zu ändern. Allerdings erforderte bereits eine eingeschränkte CXF-Version Änderungen an der Klassenbibliothek, insbesondere am GNU-Classpath-Projekt. Andererseits erwarten wir, dass dies mit der Veröffentlichung von OpenJDK [18], einer Open-source-Java-Version, zukünftig zu einem geringeren Problem werden wird.

Das Deployment unter Java ME CDC stellt eine größere Herausforderung dar, da hier Java 1.4 die Grundlage bildet. Bytecode-Weaving in Verbindung mit dem Apache-Harmony-Projekt hat sich als geeigneter Weg erwiesen. Dabei werden jedoch existierende Klassen überschrieben, so dass ein intensives Testen unumgänglich ist.

Referenzen

1. Apache Software Foundation: Apache Harmony pages. http://harmony.apache.org/
2. Apache Software Foundation: CXF pages. http://incubator.apache.org/cxf/
3. Barr, M., Massa, A.: Programming Embedded Systems, 2nd edn. O'Reilly, CA, 2007
4. Interface21 Inc.: Spring-OSGi. http://www.springframework.org/osgi/
5. Java Community Process: JSR 173: Streaming api for xml. http://jcp.org/en/jsr/detail?id=173
6. Java Community Process: JSR 181:Web services metadata for the Java TM platform. http://jcp.org/en/jsr/detail?id=181
7. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA. Prentice Hall, NJ, 2005
8. Kriens, P., Hargrave, B.J.: Whiteboard pattern, 2004. http://www.osgi.org/documents/osgi_technology/whiteboard.pdf
9. Lougher, R.: JamVM pages. http://jamvm.sourceforge.net/
10. Newcomer, E., Lomow, G.: Understanding SOA with Web Services. Addison-Wesley, 2005
11. OASIS: SOA pages. http://www.oasis-open.org/committees/tc_cat.php?cat=soa
12. OSGi Alliance: Enterprise expert group. http://www2.osgi.org/EEG/HomePage
13. Puchko, T.: Retrotranslator. http://retrotranslator.sourceforge.net/
14. Rellermeyer, J.S., Alonso, G.: Services everywhere: OSGi in distributed environments. In: EclipseCon 2007, Santa Clara, CA, March 5–8, 2007
15. Siemens: OpenSOA press release, May 15, 2007
16. Sun Microsystems: Java ME pages. http://java.sun.com/javame/overview/techpapers/
17. Sun Microsystems: Java SE Embedded pages. http://java.sun.com/javase/embedded/
18. Sun Microsystems: OpenJDK pages. http://openjdk.java.net/
19. ETH Zürich: R-OSGi pages. http://r-osgi.sourceforge.net/
20. World Wide Web Consortium: Web services pages. http://www.w3.org/2002/ws/