Nach Hause telefonieren

Anbindung mobiler Endgeräte durch jtom

Frank Schlinkheider, Marco A. Stratemann

Es gibt viele Möglichkeiten bestehende Geschäftssysteme um mobile Komponenten zu erweitern. Soll die Integration einer mobilen Lösung aber einfach und kostengünstig erfolgen, so kann die Komplexität verteilter mobiler Anwendungen durch eine Middleware vereinfacht werden. In diesem Artikel wird die mobile Middleware jtom vorgestellt, die über JMS eine Anbindung von J2ME-Geräten an Serversysteme ermöglicht.

Um neue und bestehende Geschäftsanwendungen auch von mobilen Endgeräten aus effektiv nutzen zu können, müssen die spezifischen Eigenschaften mobiler Kommunikation bedacht werden. Ein mobiles Endgerät erlaubt den entfernten Zugriff auf Daten eines Unternehmens von nahezu überall her – jedoch nicht von überall her gleich gut. Das in stationären verteilten Systemen problemlos einzusetzende synchrone Kommunikationsmodell ist in der mobilen Umgebung nicht optimal verwendbar. Der Natur mobiler Kommunikation mit der Unzuverlässigkeit der (Mobilfunk-)Netzverbindung entspricht ein asynchrones Modell viel eher: Könnten Kurznachrichten nur dann empfangen werden, wenn Sender und Empfänger gleichzeitig im Mobilfunknetz eingebucht sind, wäre dem Short Message Service (SMS) kaum ein solcher Erfolg möglich gewesen.

Grundlagen

Im weiteren Verlauf dieses Artikels wird zuerst eine kurze Einführung in die Java 2 Micro Edition (J2ME) und benötigte Technologien gegeben, danach die typische Architektur einer verteilten mobilen Anwendung betrachtet und abschließend die API der mobilen Middleware jtom vorgestellt.

Java 2 Micro Edition

Die J2ME ist eine von Sun Microsystems speziell auf Endgeräte mit begrenzten Ressourcen zugeschnittene Java-Umgebung. Um für diesen Einsatzbereich mit sehr vielen unterschiedlichen Geräten die notwendige Flexibilität zu erreichen, besteht die J2ME aus einer kleinen Anzahl von *Konfigurationen* und *Profilen*. Konfigurationen definieren für eine Klasse von Geräten eine gewisse Basisfunktionalität sowie die virtuelle Maschine (VM). Profile ergänzen diese Funktionalität für spezielle Gerätetypen durch weitere Schnittstellen und Klassenbibliotheken für z. B. Netzwerkkommunikation und Benutzungsschnittstellen. Optionale Pakete erweitern diese Kombination; zusammen bilden diese drei Elemente die Programmierumgebung, die ein Entwickler auf einem bestimmten Gerät verwenden kann, um Anwendungen zu erstellen.

Zur Zeit existieren in der J2ME zwei Konfigurationen, die *Connected Limited Device Configuration (CLDC)* sowie die *Connected Device Configuration (CDC)*. Die CLDC-Spezifikation erfordert eine Java-Laufzeitumgebung von mindestens 192 KB auf dem Endgerät. Die CDC unterstützt die aus der J2SE bekannte VM und erfordert mindestens 4,5 MB für die Laufzeitumgebung. Somit lassen sich CLDC-Geräte eher im Bereich aktueller Mobiltelefone finden; CDC-Geräte entsprechen eher höherwertigen PDAs mit mehr Ressourcen.

Das *Mobile Information Device Profile (MIDP)* definiert in Kombination mit der CLDC die Schnittstellen für die Anwendungsentwicklung auf ressourcenarmen Endgeräten. J2ME-Programme für das weit verbreitete MIDP werden MIDlets genannt, zusammen mit benötigten Ressourcen in einer MIDlet Suite gebündelt und auf dem Endgerät installiert. Dazu beschreibt die J2ME die Möglichkeit, diese Anwendungen drahtlos, also over-the-Air auf das Endgerät zu übertragen. Die Spezifikationen schreiben für die aktuell noch weit verbreitete MIDP Version 1.0 (z. B. Nokia 7650, Siemens S55) HTTP als einzig erforderliches Kommunikationsprotokoll vor, seit der MIDP Version 2.0 (z. B. Nokia 6600, Sony Ericsson K700i) zusätzlich auch HTTPS. Sämtliche Kommunikation zwischen Endgerät und Server muss sich demzufolge auf ein entsprechendes Request/Response-Protokoll abbilden lassen.

Anbindung an vorhandene Geschäftslogik

Verteilte mobile J2ME-Anwendungen kommunizieren in der Regel über einen Webapplikationsserver (WAS) mit der Geschäftslogikschicht (s. Abb. 1). Der WAS bearbeitet die HTTP/HTTPS-Anfragen mobiler Clients und leitet diese an die zurückliegende Geschäftslogik weiter. Antworten werden häufig ebenfalls über den WAS zurückgeliefert und so bietet es sich an, im Sinne einer einfach erweiterbaren Anwendungsarchitektur auf dem WAS J2EE-Komponenten, wie zum Beispiel Servlets, zu verwenden. Über diese J2EE-Schicht kann dann beliebige serverseitige Geschäftslogik angebunden werden.

Messaging & JMS

Es gibt viele Möglichkeiten, um zwischen zwei oder mehr Teilnehmern in einem Netzwerk Daten auszutauschen. Soll dies zuverlässig zwischen Anwendungen geschehen und sollen diese dabei am besten noch voneinander entkoppelt werden, so führt an einer Lösung mittels Messaging schwerlich ein Weg vorbei. Der *Java Message Service* (JMS) bietet eine Schnittstelle für Java-Anwendungen, um auf Messaging-Systeme zugreifen zu können. JMS definiert dazu für 1:1 Kommunikation Nachrichtenqueues, sowie für 1:N Verteilung ein Publish/Subscribe-Modell. Über eine wählbare *Quality of Service* (QoS) kann sichergestellt werden, dass jede Anwendung Nachrichten mit der gewünschten Zuverlässigkeit senden und empfangen kann.

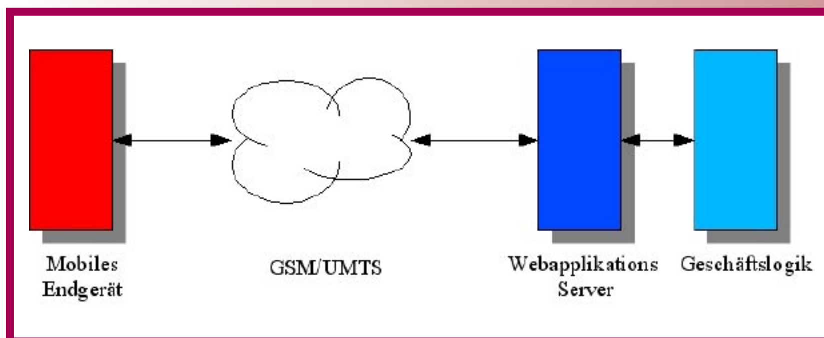

Abb. 1: Typische Architektur einer verteilten Anwendung mit mobilen und stationären Komponenten

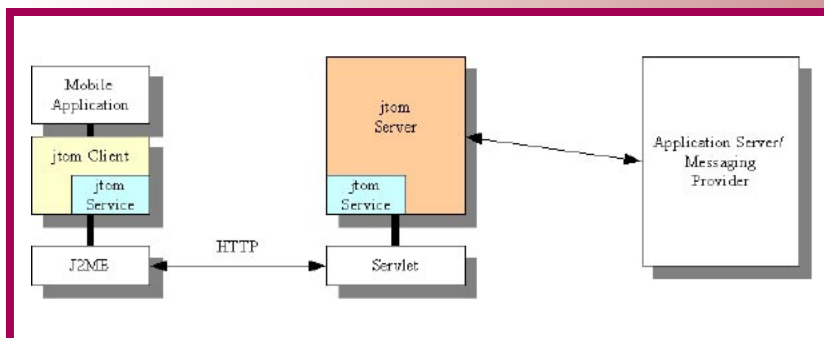

Abb. 2: Schematische Darstellung der Anbindung eines mobilen Endgerätes über die mobile Middleware jtom an J2EE-Applikationsserver

Ein Anwendungsszenario

Ein mögliches Szenario für den Einsatz einer mobilen Middleware ist im Umfeld eines (Fahrrad-)kurierdienstes zu finden. Die Anbindung aller Kuriere an die Zentrale ist auf diesem Wege einfach und kostengünstig zu realisieren. Die Anwendung besteht hier aus einer Komponente auf dem Client, die für die Schnittstelle zum Benutzer verantwortlich ist, sowie einer Serverkomponente, die die Anbindung an den JMS-Provider und weitere Geschäftslogik ermöglicht.

Die Kuriere können in diesem Fall ihr eigenes Mobiltelefon für den Einsatz verwenden, da die Anwendung nur MIDP 1.0 Standard APIs verwendet. Um eine möglichst einfache und reibungslose Bedienung durch die Kuriere zu gewährleisten, wird die Anwendung vor der Installation personalisiert, dann drahtlos zu dem jeweiligen Endgerät übertragen und installiert.

Sobald ein Auftrag in der Zentrale eingeht, wird über das System ein Start- und Zielpunkt für die Abholung und Anlieferung eingegeben und an alle Fahrer versendet. Durch die konfigurierbare QoS kann sichergestellt werden, dass alle Fahrer diese Nachricht erhalten. Kommt diese Route für einen Fahrer in Betracht, so bestätigt er den Auftrag über den entsprechenden Menüpunkt der Anwendung auf seinem Mobiltelefon. Hierbei spielt wiederum die garantierte Zustellung von JMS bei entsprechend konfigurierter QoS eine wichtige Rolle; der bestätigende Fahrer kann somit nach Erhalt einer Empfangsbestätigung sicher sein, dass seine Auftragsannahme die Zentrale auch wirklich erreicht hat.

jtom

Um das beschriebene Anwendungsszenario umzusetzen, kommt die hier vorzustellende mobile Middleware jtom zum Einsatz. Dazu werden erst ein paar architekturspezifische Punkte erläutert, bevor es zur eigentlichen Realisierung geht. Die jtom-Architektur gliedert sich in drei Module *Client*, *Server* und *Service*.

Der *jtom-Client* ist der Teil des Frameworks, der auf dem mobilen Endgerät den Anwendungen Schnittstellen für die Kommunikation mit J2EE-Komponenten zur Verfügung stellt. Anwendungen implementieren die Benutzungsschnittstelle und verwenden die Clientkomponente für den Zugriff auf die entfernte Geschäftslogik.

Die *jtom-Service*-Komponente beinhaltet die auf Client- und Serverseite gleichermaßen benötigten Funktionen für die Datenübertragung. Sie erlaubt die transparente Umwandlung von Objekten zu Datenströmen und zurück und wird vom jtom-Client und dem jtom-Server verwendet.

Der *jtom-Server* ist für die Verwaltung der Clients und die erforderliche Übersetzung der zu übertragenden Daten verantwortlich. Der hier verwendete Server entspricht von der Funktion her dem WAS in Abbildung 1. Er nimmt die Anfragen der Clients entgegen und ist gleichzeitig Client für den J2EE-Applikationsserver, auf dem die eigentliche Geschäftslogik der gesamten Anwendung residiert. Abbildung 2 zeigt die Komponenten der Architektur im Überblick.

Installation von jtom

Eine für nicht-kommerzielle Zwecke kostenlose jtom-Version kann unter [jtom] bezogen werden. Nach der Registrierung und dem Download wird das Archiv auf dem Entwicklungsrechner entpackt und anhand der beiliegenden detaillierten Anleitung installiert.

Zusätzlich wird für die Entwicklung mobiler Anwendungen eine entsprechende Entwicklungsumgebung benötigt. Der Entwickler kann sich dabei zwischen einer Vielzahl von Werkzeugen entscheiden (z. B. J2ME WTK [WTK], Eclipse inkl. EclipseME [Sourceforge], IBMs WebSphere Studio Device Developer [WSDD], Borlands JBuilder Mobile Studio [Borland], Sun Java Studio Mobility [JSMobility], NetBeans inkl. Mobility-Pack [NetBeans], etc.). Die Entwicklung mit dem jtom-Framework ist dabei immer identisch: die benötigte Client-Bibliothek muss in den CLASSPATH aufgenommen und beim Erzeugen des lauffähigen MIDlets mit eingebunden werden.

Die mobile Middleware jtom besteht aus einer Client- und einer Serverkomponente. Die Serverkomponente wird in einem Webapplikationsserver wie z. B. Apache Jakarta Tomcat eingebunden. Hier wird der Applikationsserver JBoss 4.x in Kombination mit Tomcat eingesetzt, um den integrierten Messaging Provider nutzen zu können. Das mitgelieferte `jtomServer.war` muss bei der Installation nur in das entsprechende JBoss-Verzeichnis kopiert werden. Mehr ist für die Installation der Serverkomponente nicht notwendig.

Implementierung eines J2ME-Clients

Nach der erfolgreichen Installation kann jetzt ein erster J2ME-Client entwickelt werden. Der Client soll einen Überblick über die aktuelle API liefern und die Möglichkeiten der mobilen Middleware darstellen.

jtom API

Die jtom API ist an die vorhandene Wireless Messaging API (WMA) angelehnt. WMA wird bei der J2ME-Entwicklung für das Versenden und Empfangen von SMS und MMS verwendet. jtom erweitert diese Schnittstelle um JMS-Funktionalität. Das *GenericConnection Framework*, ein Bestandteil der CLDC, wird dazu als Einstieg verwendet. Listing 1 zeigt, wie sich ein J2ME-Programm bei einem JMS-Provider für Nachrichten registriert.

Diese Registrierung bei dem Message-Provider muss erfolgen, damit ein Client neue Nachrichten erhält; übersetzt in unser Beispiel bedeutet es, dass die Zentrale den einzelnen Kurier über die jüngsten Aufträge informieren kann. Im Gegensatz zu dem Versand von SMS-Nachrichten können hierbei komplexe und umfangreiche Datenstrukturen übertragen werden, die dann anschließend von der Anwendung auf dem Client ausgewertet und interpretiert werden können.

```
/**
 * register a MessageListener with a message provider
 */
public void registerReceiver(String url) {
  try {
    // open connection to JMS-server
    MessageConnection mConn = (MessageConnection)Connector.open(url);

    // "this" is registered as MessageListener for JMS-messages
    // When receiving a message the method
    // "this.notifyIncomingMessage()" is called
    mConn.setMessageListener(this);
    mConn.setExceptionListener(this);
  } catch (java.io.IOException ioEx) {
    // handle Exceptions as required
  }
}
```

Listing 1: Anmeldung an einem JMS-Provider

Der auf dem Endgerät installierte Client konfiguriert über eine URL alle Informationen, die die Verbindung betreffen, sowie für JMS relevante Daten (wie der verwendete Messaging-Typ oder die benötigte Fabrik für Verbindungen). Listing 2 stellt eine mögliche URL dar, diese besteht aus obligatorischen Angaben zu Verbindungen und JMS sowie optionalen Daten für die Benutzerauthentifizierung. Die detaillierte Bedeutung der jeweiligen JMS-Parameter kann in der JMS-Dokumentation unter [JMS] nachgelesen werden. Konkret besteht eine solche URL aus Pflicht- und optionalen Angaben. Zunächst werden die Pflichtangaben genauer erläutert:

```
// req: jtom protocol identification string
String url = "wma2jms: //"

  // req: connection url for service
+"http://localhost:8080/jtomServer/MIDPService;"

  // req: key for accessing WMA-to-JMS mapping bean
+"jtom/Connection;"

  // req: JMS connection factory
+"factory=ConnectionFactory;"

  // req: message destination name
+"name=testTopic;"

  // req: used messaging model (topic|queue)
+"type=TOPIC;"

  // opt: JMS user name
+"username=john;"

  // opt: JMS user password
+"password=needle";
```

Listing 2: Ein mögliches Beispiel für eine Connection-URL

1. `wma2jms://`: Protokoll zum Versenden und Empfangen von JMS-Nachrichten.
2. `http://localhost:8080/jtomServer/MIDPService;`: URL für den Zugriff auf den Webapplikationsserver (und die jtom-Serverkomponente).
3. `jtom/Connection;`: Schlüssel für den Zugriff auf die Bean für das Mapping von WMA- auf JMS-Nachrichten.
4. `factory=ConnectionFactory;`: JMS-Verbindungsfabrik.
5. `name= testTopic;`: Name des Bestimmungsortes für Nachrichten (Destination).
6. `type=TOPIC;`: Unterscheidung zwischen Topic und Queue (Destination-Typ).

Sowie den folgenden optionalen Bestandteilen:

7. `username=john;`: Benutzername des JMS-Users.
8. `password=needle`: Passwort des JMS-Users.

Wenn keine Daten zur Benutzeranmeldung angegeben werden, meldet sich der Client als „Guest“ an. Die Verwaltung der Benutzer wird vom verwendeten Message-Provider übernommen. Wird z. B. JBoss als JMS-Provider eingesetzt, können die benötigten Benutzer, Passwörter und entsprechende Rollen in der `jbossmq-state.xml` gepflegt und konfiguriert werden. jtom besitzt kein eigenes Benutzerverwaltungssystem, sondern nutzt die vorhandene Implementierung des angebundenen JMS-Providers.

Nach dem Aufbau einer Verbindung mittels `Connector.open(url)` trägt sich der `MessageListener` mittels `setMessageListener(MessageListener)` für die asynchrone Zustellung als Empfänger ein. Der `MessageListener` muss die Methode `notifyIncomingMessage` implementieren, um Text- oder auch Binärnachrichten zu empfangen (s. Listing 3).

JMS unterstützt eine Vielzahl unterschiedlicher Nachrichtentypen. Wegen der geringeren Performance und den meist geringen Ressourcen mobiler Endgeräte unterstützt jtom nur zwei Typen: Text- und Binärnachrichten. Dieses reicht aber für sämtliche Anwendungszwecke aus, da mit binären Nachrichten beliebige Datenstrukturen in Form von Byte-Arrays übertragen werden können.

Die oftmals instabile Verbindung zwischen mobilen Clients und entfernten Serversystemen ist beim Entwurf einer Anwendung besonders zu bedenken. Die Anwendung muss die Möglichkeit bekommen, auf den Verlust der Verbindung, evtl. Probleme beim Versand oder Empfang von Nachrichten entsprechend reagieren zu können. Dazu wurde abweichend von der

```
/**
 * This method is called when a message arrives.
 */
public void notifyIncomingMessage(MessageConnection conn) {
  try {
    // fetches message from connection
    Message msg = (Message) conn.receive();

    // type based handling of messages
    if(msg instanceof BinaryMessage) {
      // BinaryMessage
      byte[] bytes = ((BinaryMessage )msg).getPayloadData();

      // do other stuff with the payload here
     } else {
       // TextMessage
       String string = ((TextMessage )msg).getPayloadText();

       // do other stuff with the payload here
     }
   } catch (java.io.IOException ioEx) {
     // handle Exceptions as required
   }
}
```

Listing 3: Eingang einer JMS-Nachricht

WMA-Implementierung ein **ExceptionListener** eingeführt. Sobald die Verbindung abbricht oder ein Fehler innerhalb des JMS-Serversystems auftritt, wird der Client durch den Aufruf der **onException()**-Methode des **ExceptionListeners** informiert. Die Anwendung kann dann entscheiden, inwieweit sie direkt versucht auf den Verbindungsverlust zu reagieren und die Verbindung zum WAS erneut aufbaut oder den Anwender über das Problem informiert und ihm dann die Entscheidung überlässt. Wenn der Anwender sich dazu entschließt, die Verbindung zeitversetzt erneut aufzunehmen, ist die garantierte Zustellung der JMS-Nachrichten notwendig. Dieses gilt beispielsweise, wenn beim Verbindungsverlust für den Client im Middle Tier weitere Messages eingehen. Diese gehen nicht verloren, sondern werden nach erneutem Verbindungsaufbau sicher zugestellt.

Ein *automatischer* Wiederaufbau der Verbindung nach Verlust kann ebenfalls konfiguriert werden. Oftmals soll aber die Anwendung oder der Anwender entscheiden können, ob die Verbindung erneut aufgebaut wird.

Das Senden einer JMS-Nachricht funktioniert in ähnlich einfacher Weise (s. Listing 4); auch hier werden Text- und Binärnachrichten unterstützt.

```
/**
 * Send JMS-messages.
 */
public void sendMessage(String message, String url) {
  try {

    // open connection to JMS-server
    MessageConnection mConn = (MessageConnection)Connector.open(url);

    // create new TextMessage
    TextMessage msg = (TextMessage)mConn.
      newMessage(MessageConnection.TEXT_MESSAGE);

    // set text to send
    msg.setPayloadText(message);

    // send message
    mConnection.send(msg);
  } catch (java.io.IOException ioEx) {
    // handle Exceptions as required
  }
}
```

Listing 4: Senden von JMS-Nachrichten

Die J2ME-Fahrradkurieranwendung besitzt somit alle notwendigen Funktionen. Sie kann Aufträge empfangen und diese im Display anzeigen. Der Kurier hat anschließend die Möglichkeit, einen Auftrag anzunehmen und dieses mittels einer generierten JMS-Nachricht der Zentrale mitzuteilen. Jetzt fehlt in diesem Beispiel noch ein Serversystem, welches die Kuriere über neue Aufträge informiert und nach Eingang einer Auftragsannahme durch den Kurier zusätzliche Informationen über den Auftrag zum Fahrradkurier überstellt. Dieses könnte z. B. über eine Webanwendung realisiert werden, deren Beschreibung den Rahmen dieses Artikels sprengen würde.

Push Registry

War es MIDP 1.0-Endgeräten noch nicht möglich, eingehende Verbindungsanfragen zu akzeptieren und dadurch Serverdienste anzubieten, so ist die in MIDP 2.0 neu hinzugekommene Push Registry ein Schritt in diese Richtung. Die Push Registry bietet die Möglichkeit, eine „schlafende“ Anwendung auf dem Endgerät über eine neue Verbindungsanfrage von der Serverseite (oder einem anderen Endgerät) aus „aufzuwecken“. Anwendungen müssen sich dazu bei dem *Java Application Manager* (JAM) auf dem Endgerät registrieren. Dieser weckt die Anwendung bei Aktivität auf dem gewünschten Port auf und übergibt die bisher eingegangenen Daten.

Auf dem ersten Blick scheint diese Funktionalität ähnlich der jtom-Lösung, allerdings liegen die Unterschiede im Detail: Die Push Registry arbeitet nur lokal auf dem Endgerät und kommuniziert nicht mit Anwendungen auf der Serverseite. Um eine (Client-)Anwendung von der Serverseite aus ansprechen zu können, muss das Gerät adressiert werden können. Da liegt es nahe, mangels statischer IP-Adressen auf Mobiltelefonen ein Endgerät über SMS oder MMS (über WMA) zu kontaktieren und so die Verbindung herzustellen. Die Push Registry funktioniert somit nur zuverlässig über SMS oder MMS; jtom kommt hingegen mit einer HTTP-Verbindung aus.

Es wird zwar immer wieder davon gesprochen, dass die Push Registry auch mittels einer Internetverbindung über TCP/IP funktionieren kann, dieses scheitert aber in der Regel daran, dass dann auch das mobile Endgerät Serversockets unterstützen muss. Dieses ist aber eine optionale Funktion in MIDP 2.0 und wird nur von wenigen Geräten angeboten. Zusätzlich ist über die Push Registry keine direkte Anbindung von J2ME-Geräten an JMS-Provider möglich. Um wirklich beliebige J2ME-Endgeräte über JMS ansprechen zu können, bleibt daher nur der Einsatz einer mobilen Middleware, die auf allgemein unterstützten Protokollen wie HTTP realisiert wurde.

Zusammenfassung & Ausblick

In diesem Artikel wurde gezeigt, wie auf einfache Weise eine mobile Anwendung für einen Fahrradkurierdienst entwickelt werden kann. Es wurden die Probleme bei der Anbindung mobiler Anwendungen an Serversysteme dargestellt und mögliche Lösungen erörtert.

Sobald ein Zugriff auf entfernte Server- oder Messaging-Systeme benötigt wird, erleichtert und beschleunigt eine mobile Middleware wie jtom das Entwickeln von verteilten J2ME-Anwendungen. Die Vorteile der hier vorgestellten Lösung gegenüber anderen möglichen Lösungsansätzen lassen sich in einigen Punkten zusammenfassen:

- läuft auf jedem J2ME-Endgerät (CLDC und CDC),
- kostengünstiger als SMS und MMS,

- Unterstützung aller JVMs,
- Unterstützung aller zu J2EE konformen JMS-Provider,
- fehlertolerant,
- skalierbar (durch Einsatz in einem Verbund von Applikationsservern),
- gesicherte Zustellung von Nachrichten.

Der Einsatz einer mobilen Middleware hilft bei der Entwicklung solcher Systeme und beschleunigt durch die Verwendung bekannter APIs die Realisierungsphase. Manche Funktionen lassen sich für einzelne Endgeräte auch mit den vorhandenen J2ME-Klassen abbilden. Sobald aber eine Vielzahl von unterschiedlichen Geräten eingebunden werden soll und die eigene Entwicklung einer Middleware aus Zeit- oder Kostengründen nicht wünschenswert ist, kann eine vorhandene mobile Middleware wie jtom mit Unterstützung für Messaging Vorteile bringen.

Die Implementierung einer JNDI-Schnittstelle für J2ME bildet die Basis für diese Messaging API. jtom verwendet diese eigene Schnittstelle zwar intern, sie ist aber bisher noch nicht offen gelegt. Diese API soll jedoch bis Mitte des Jahres den Entwicklern zur Verfügung gestellt werden. Aber wir wünschen bereits jetzt schon viel Spaß bei der Entwicklung mit jtom!

Links

[Borland] Borlands JBuilder Mobile Studio, `http://www.borland.com/mobile/studio/index.html`
[JMS] Dokumentation des Java Message Service, `http://java.sun.com/products/jms/docs.html`
[JSMobility] Sun Java Studio Mobility, `http://www.sun.com/software/products/jsmobility/index.xml`
[jtom] mobile Middleware jtom, `http://www.jtom.de`
[NetBeans] NetBeans inkl. Mobility-Pack, `http://www.netbeans.org/`
[Sourceforge] EclipseME, `http://www.eclipse.org/`, `http://eclipseme.sourceforge.net/`
[WSDD] IBMs WebSphere Studio Device Developer, `http://www-306.ibm.com/software/wireless/wsdd/`
[WTK] J2ME WTK, `http://java.sun.com/products/j2mewtoolkit/`

Frank Schlinkheider ist Systemarchitekt und Projektleiter bei der Firma ITSD Consulting GmbH und dort seit mehreren Jahren verantwortlich für die Umsetzung mobiler Anwendungen. Schwerpunkt ist die Anbindung von J2ME-Anwendungen an J2EE-Serversysteme und die Integration mobiler Lösungen in vorhandene Strukturen und Geschäftsabläufe. E-Mail: fs@itsd-consulting.de.

Marco A. Stratemann ist bei der Firma ITSD Consulting GmbH in den Bereichen Enterprise Web & Mobile Solutions als Anwendungsberater und Entwickler tätig. Er beschäftigt sich seit einigen Jahren mit der Architektur und der Entwicklung verteilter und mobiler Anwendungen. E-Mail: mas@itsd-consulting.de.