# Entwicklung eines Gatewaysystems für telematikbasiertes Gerätemonitoring

*Ralf Vandenhouten, Thomas Behrens, Bettina Schnor*

## 1 Einleitung

### 1.1 Motivation

In den vergangenen Jahren waren Bestrebungen der marktführenden Hersteller im Bereich der Automatisierungstechnik (z. B. Siemens, ABB, Jetter) zu beobachten, ihre Systeme für einen transparenten, vertikalen Informationstransport von der Feldbus- bis zur Leit-, Planungs- oder Managementebene aufzurüsten und nach außen entsprechende Internetschnittstellen zur Verfügung zu stellen. All diese Bemühungen verfolgen jedoch homogene, proprietäre Lösungen, zugeschnitten auf die spezielle Hardware des jeweiligen Herstellers, mit dem Ziel, die jeweils neueste Generation dieser Hardwarekomponenten auf dem zunehmend auf die Informationstechnik ausgerichteten Markt positionieren zu können. Der Nachteil dieser Lösungen ist, dass sie

a) vorhandene Automatisierungstechnik nicht unterstützen und
b) keine inhomogenen Systeme mit Komponenten verschiedener Hersteller erlauben.

Eine große Zahl laufender Anlagen ließe sich also nur durch den vollständigen Ersatz der Software und Hardware an die moderne Informationstechnik anschließen, was mit enormen Kosten verbunden wäre, die insbesondere viele mittelständische Unternehmen nicht tragen können. Was beim derzeitigen Stand der Technik fehlt, ist eine Lösung, die einerseits Schnittstellen für die vorhandene Automatisierungstechnologie zur Verfügung stellt, andererseits offene Schnittstellen für die Integration zukünftiger Komponenten beliebiger Hersteller bietet und darüber hinaus einen komfortablen, ortsunabhängigen Zugriff über das WWW (World Wide Web) und WAP (Wireless Application Protocol) erlaubt. Die Entwicklung eines solch innovativen Produktes war Gegenstand eines GeWiPlan-Projektes, das gemeinsam mit einem brandenburgischen KMU durchgeführt wurde. Der Prototyp des Systems entstand im Rahmen einer Diplomarbeit am Lehrstuhl Telematik der TFH Wildau in Zusammenarbeit mit dem Institut für Informatik der Universität Potsdam.

### 1.2 Aufgabe

Der Kerngedanke war die Entwicklung eines neuartigen, WWW- und WAP-basierten Telematik-Systems zur Unterstützung der Fernüberwachung und -steuerung von vernetzten Geräten. Das System sollte in der Lage sein, vorhandene IT-Infrastrukturen in Produktionsbetrieben zu integrieren, Betriebsdaten (z. B. Ist-Zustände von Maschinen, Werkstoffen, Mitarbeitern, Patienten) auszuwerten und transparent externen Abfragern ortsunabhängig und zu jedem Zeitpunkt über Internetdienste (WWW) sowie im Falle mobiler Einheiten über das WAP zur Verfügung zu stellen. Dabei ist sowohl die (passive) Überwachung von Ressourcen, Produktionsparametern und Qualitätsmerkmalen möglich, als auch die Fernmanipulation von Einflussgrößen der Planungs-, Leit-, Automatisierungs- und Feldebene.

### 1.3 Schichtenmodell

Zunächst wurde eine Software-Architektur für das System konzipiert, die aus einem Schichtenmodell mit fünf Schichten besteht (Abb. 1), um flexibel unterschiedlichste Endgeräte einbinden zu können.

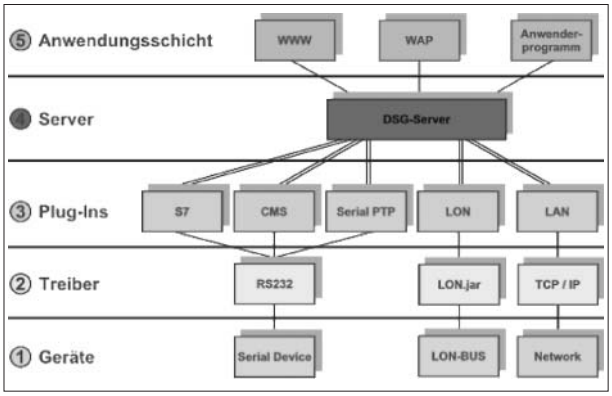


*Abb. 1: Schichtenmodell des Gatewaysystems*

Zu den *Geräten* gehören solche, die eine Verbindung zu einem Computer haben und deren Daten ausgewertet und/oder entfernt visualisiert werden sollen, zum Beispiel Patientenmonitore in der Medizin oder Automatisierungs- und Produktionssysteme.
Die *Treiber* sind für die Anbindung der Geräte an die Software über eine Hardwareschnittstelle verantwortlich, zum Beispiel RS232 oder TCP/IP.
Um Daten aus den Geräten zu lesen, werden (in der Regel proprietäre) Protokolle verwendet. Diese Protokolle sind für jedes Gerät anders, da jedes Gerät andere Daten in anderen Zeitabständen liefert. Die *Plug-Ins* kommunizieren mit den Geräten über deren Protokolle und konvertieren die Daten in allgemeine Datenstrukturen, die der Server verarbeiten kann. Für die unterschiedlichen Geräte werden demzufolge verschiedene Plug-Ins benötigt. Die Plug-In-Schicht macht die spezifischen Eigenheiten einzelner Geräte für den Server transparent.
Der *Server* ist das Kernstück des Systems. Dieser verwaltet die Daten der Geräte und stellt die Benutzerschnittstellen bereit, über die die Daten geschrieben und gelesen werden können.

In der *Anwendungsschicht* werden die Daten visualisiert. Das kann das Darstellen von Messdaten, Zeitreihen oder mehrdimensionalen Messdaten sein. Die bevorzugten Benutzerschnittstellen sind das WWW und WAP.

### 1.4 Anforderungen

Für die Kommunikation zwischen dem Server und den Plug-Ins wird eine offene Schnittstelle benötigt. Den Herstellern von Automatisierungs- und Produktionssystemen muss die Möglichkeit gegeben werden, eigene Komponenten zu entwickeln und anzubieten, die mit dem Server zusammenarbeiten. Die Common Object Request Broker Architecture (CORBA) bietet für solch eine Schnittstelle eine geeignete und flexible Plattform.
Für den Zugriff über die WWW- und WAP-Schnittstellen können HTML-/JavaScript-Dokumente/Formulare für den statischen Zugriff über HTTP und WML-Dokumente für den Zugriff über WAP entwickelt werden. Darüber hinaus sollten Java-Applets für den interaktiven Zugriff der zur Verfügung gestellten Daten und Geräte entwickelt werden.
Mit einem Konfigurationsprogramm wird der Server konfiguriert. Der Server kann so an die Gegebenheiten eines konkreten Telemetrievorhabens (z. B. in einem Produktionsunternehmen) angepasst werden. Um die Konfiguration komfortabel, robust und effizient zu machen, wird ein grafisch unterstütztes Konfigurationsprogramm benötigt.
Um den Server auf jeder Plattform (z. B. Windows, Linux) ohne weiteren Programmieraufwand einsetzen zu können, sollte der Server in der plattformunabhängigen Programmiersprache Java entwickelt werden.

## 2 Gateway

Der Gatewayserver ist das Kernstück des Projektes. Er hat die Aufgabe, die Daten der Geräte zu verwalten und über die Schnittstellen WWW und WAP zur Verfügung zu stellen. In diesem Abschnitt werden der Aufbau und die Implementierung beschrieben.

### 2.1 Verwendbare Technologien

Das Gatewaysystem sollte auf einem eigenen Rechner/Server installiert werden. Das ist notwendig, um die Sicherheit für die Daten und die Stabilität des Programms gewährleisten zu können. Dieser Rechner/Server muss außerdem auch an das Internet/Intranet angeschlossen sein. Nur dann können die Daten von den Geräten zum Gatewaysystem übertragen und über die Schnittstellen der Anwendungsschicht im Internet angeboten werden.
Für den Aufbau des Gatewaysystems werden verschiedene Technologien eingesetzt. Dazu gehören der Applikationsserver, Servlets, JSP (Java Server Pages) [1] und EJB (Enterprise Java Beans) [5]. Applikationsserver stellen Benutzern in einem Netzwerk Anwendungsprogramme über verschiedene Schnittstellen zur Verfügung. Die Anwendungsprogramme dafür müssen in der Programmiersprache Java als Servlet, JSP und/oder EJB geschrieben werden.

### 2.2 Systemarchitektur

Die wichtigste Anforderung an das System ist, dass die Schnittstellen CORBA und WWW/WAP unterstützt werden. Daraus ergeben sich folgende Möglichkeiten für den Aufbau:

1. Für jede Schnittstelle wird ein eigenständiges System/eigenständiger Server selbst entwickelt (Abb. 2).

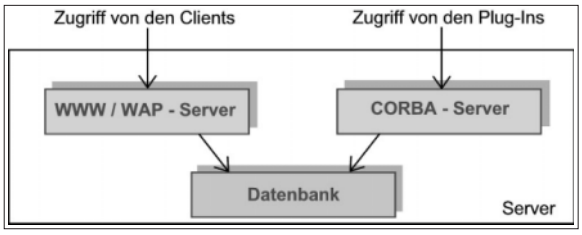


*Abb. 2: Aufbaumöglichkeit 1*

   Diese erste Möglichkeit hat den Vorteil, dass man alles selbst unter Kontrolle hat, Fehler schnell beheben kann, das System sehr kompakt ist und die beiden Schnittstellen unabhängig voneinander funktionieren. Der Nachteil hierbei ist, dass das System fehleranfällig und der Zeitaufwand bei der Entwicklung und Pflege sehr hoch ist.
   Die Fehleranfälligkeit des Systems wird erst nach sehr vielen Tests und Bewährungsproben in der Praxis sinken. Nach der Entwicklung und verschiedenen Tests werden, basierend auf Erfahrungswerten, nicht alle Fehler gefunden, da diese nur unter bestimmten und teilweise komplexen Konstellationen auftreten.

2. Die beiden Schnittstellen werden von einem selbst entwickelten System/Server zur Verfügung gestellt (Abb. 3).

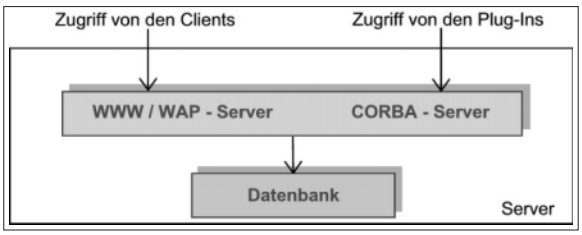


*Abb. 3: Aufbaumöglichkeit 2*

   Dieser Aufbau hat den Vorteil, dass man alles selbst unter Kontrolle hat, die Fehler besser beheben und das System seinen Anforderungen perfekt anpassen kann. Auch dieses System ist sehr kompakt. Nachteile sind hier auch die Fehleranfälligkeit, der Zeitaufwand bei der Entwicklung und Pflege und die Abhängigkeit der Schnittstellen voneinander, da sie alle in einem System integriert sind.

3. Die WWW/WAP-Schnittstelle wird von einem bereits entwickelten Applikationsserver unterstützt. Der Programmcode wird als Servlet oder JSP in den Applikationsserver eingebunden. Die CORBA-Schnittstelle wäre ein selbstentwickeltes Programm und damit eigenständig (Abb. 4).

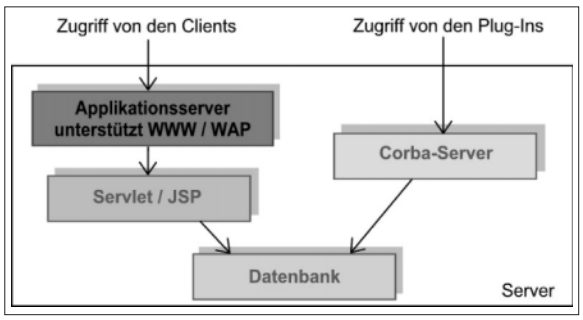


*Abb. 4: Aufbaumöglichkeit 3*

Der Vorteil bei dieser Möglichkeit ist, dass die Kommunikation über das Netzwerk vom Applikationsserver übernommen wird. Außerdem ist ein ausgereifter Applikationsserver weniger fehleranfällig. Die Entwicklungs- und Pflegezeit reduziert sich, da der zu programmierende Code geringer wird und damit übersichtlicher zu gestalten ist. Zusätzlich ist eine Benutzerverwaltung im Applikationsserver integriert, die das Entwickeln des Systems vereinfacht. Die Nachteile liegen in der Planungszeit. Diese vergrößert sich, da der für dieses Projekt richtige Applikationsserver in aufwendigen Testverfahren gefunden werden muss. Hinzu kommt noch die Einarbeitungszeit, die man benötigt, um den Applikationsserver zu administrieren und einzusetzen. Auch benötigt ein Applikationsserver mehr Ressourcen als ein selbstentwickeltes System. Ein weiterer Nachteil sind bei diesem Aufbau die beiden unterschiedlichen Systeme (Applikationsserver und eigenständiger Server).

4. Beide Schnittstellen werden von einem bereits entwickelten Applikationsserver unterstützt. Der Programmcode kann als Servlet, JSP und/oder EJB in den Applikationsserver eingebunden werden (Abb. 5).

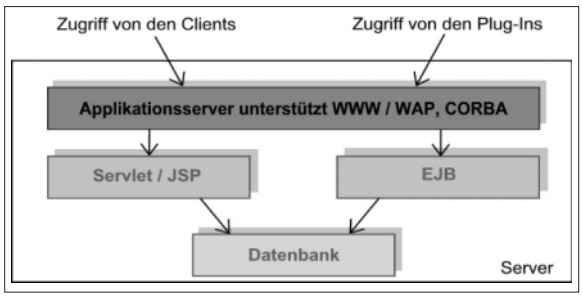


*Abb. 5: Aufbaumöglichkeit 4*

Die Vorteile entsprechen zum großem Teil denen der dritten Möglichkeit. Die Kommunikation über das Netzwerk wird vom Applikationsserver übernommen. Die Fehleranfälligkeit bei einem ausgereiften Applikationsserver ist nicht mehr so groß, da hier die auch nur unter bestimmten Konstellationen auftretenden Fehler zum Großteil gefunden und beseitigt wurden. Wie bei Alternative 3 reduziert sich die Entwicklungs- und Pflegezeit und ist eine Benutzerverwaltung im Applikationsserver integriert. Ein weiterer Vorteil ist, dass alle drei Schnittstellen in einem System integriert sind und kein zweites System zusätzlich entwickelt werden muss. Die Nachteile liegen auch hier wieder in der Planungszeit, die sich durch die Auswahlmöglichkeiten der Applikationsserver vergrößert (einmaliger Aufwand). Auch die Einarbeitungszeit in den Applikationsserver ist nicht zu unterschätzen sowie die zusätzlich benötigten Ressourcen (CPU-Leistung, Speicher).

Das Projekt wurde entsprechend der vierten Aufbauvariante realisiert, da hier die Vorteile (insbesondere Skalierbarkeit und langfristig geringer Entwicklungsaufwand) gegenüber den anderen Alternativen überwiegen.
Als Applikationsserver wurde der Sun ONE Application Server [11] gewählt, da dieser die beiden Schnittstellen unterstützt und in seiner einfachsten Version kostenlos im Internet zur Verfügung steht. Außerdem existiert eine hohe Kompatibilität zwischen dem Applikationsserver und der Programmiersprache Java (Servlet, JSP, EJB), da beide von der Firma Sun entwickelt wurden. Der Sun ONE Application Server ist auch auf verschiedenen Betriebssystemen lauffähig und damit so gut wie plattformunabhängig.

## 3 Anwendungsfälle

Die wichtigsten Aufgaben des zu entwickelnden Servers sind Daten zu speichern und diese Daten im HTML- oder WML-Format zur Verfügung zu stellen. Um die Daten übersichtlich darstellen zu können, müssen diese noch konfiguriert werden können. Für dieses System werden drei Rollen definiert, die verschiedene Rechte haben, um die erforderlichen Aufgaben erledigen zu können:
– Nutzer,
– Nutzer mit Konfigurationsrechten,
– Plug-In.

Die Rolle *Nutzer* stellt den „einfachen" Nutzer/Mitarbeiter dar. Er kann die Daten über das WWW bzw. WAP lesen. Ein Administrator oder ein Mitarbeiter mit besonderen Rechten wird durch die Rolle *Nutzer mit Konfigurationsrechten* bezeichnet. Ein solcher kann sich, wie die einfachen Nutzer, die Daten über das WWW und WAP anschauen, aber auch die Daten und Geräte für die Ansicht im WWW bzw. WAP konfigurieren. Die dritte Rolle *Plug-In* bezeichnet in diesem Fall keine Person, sondern das System, welches die Daten vom Gerät erhält und zum Server übermittelt. Diese Rolle hat nur die Aufgabe, Daten an den Server zu senden und zu schreiben. Dieser Sachverhalt wird auch im Anwendungsfalldiagramm in Abbildung 6 dargestellt.

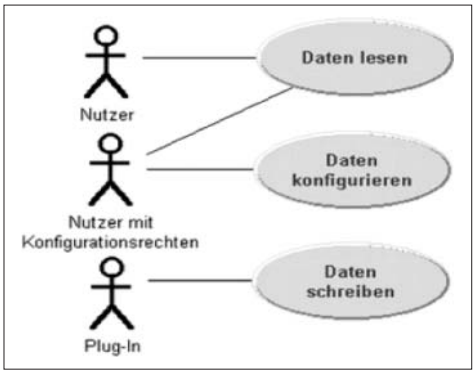


*Abb. 6: Anwendungsfälle*

Der interessanteste Anwendungsfall ist *Daten schreiben*. Dieser wird im nächsten Abschnitt näher beschrieben.

### 3.1 Daten schreiben

Dieser Anwendungsfall beinhaltet das Senden und Speichern der von einem Gerät erhaltenen Daten. Dies wird durch das Plug-In initiiert. Das physische Speichern übernimmt der Server. Aus diesem Grund sind die folgenden Aktivitäten dem Plug-In bzw. dem Server zugeordnet:

***Plug-In:***
Verbindung aufbauen
Gerätename senden
Signale senden
Kanäle senden
Daten senden
Verbindung trennen

***Server:***
Gerätename in der DB speichern
Signale in der DB speichern
Kanäle in der DB speichern
Daten in der DB speichern

Im Aktivitätsdiagramm der Abbildung 7 ist der Arbeitsablauf eines Plug-Ins dargestellt.

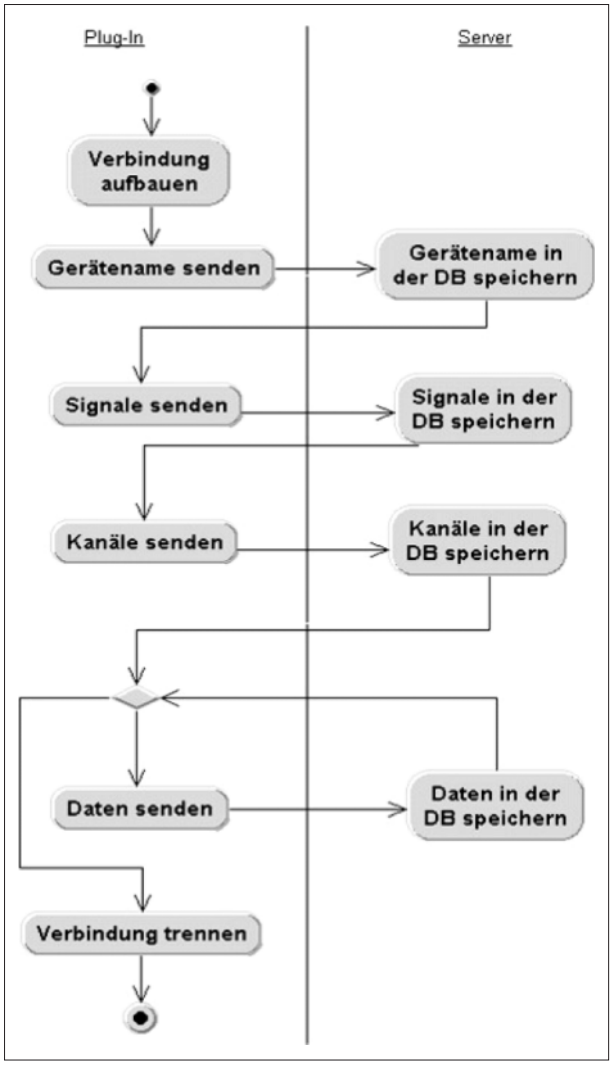


*Abb. 7: Arbeitsabläufe des Anwendungsfalls „Daten schreiben"*

## 4 Klassenarchitektur

In diesem Abschnitt werden die entwickelten Java-Klassen und deren Zusammenhänge vorgestellt. Dabei handelt es sich um das logische Architekturmodell. Hier werden nur die wichtigsten Methoden und Attribute der einzelnen Klassen beschrieben. Das technische Architekturmodell stellt alle Klassen mit ihren Methoden und Attributen dar.

Die Struktur von Paketen, denen Klassen zugeordnet sind, orientiert sich an den Anwendungsfällen, wobei die Namen der Pakete hierbei den technischen Anforderungen und nicht den der Anwendungsfälle entsprechen. Für jeden Fall wurde ein Paket erstellt. Außerdem wurde ein Paket für die Anwendungsschicht erstellt. Der Anwendungsfall *Daten lesen* findet sich in den Paketen *web* und *applet* wieder. *web* stellt dabei die textliche und *applet* die grafische Variante dar. Das Paket *config* repräsentiert den Anwendungsfall *Daten konfigurieren*. Für *Daten schreiben* wird das Paket *plugin* erstellt. Die Anwendungsschicht wird mit Hilfe der Enterprise Java-Beans erstellt. Diese werden dem Paket *ejb* zugeordnet. In den nächsten Abschnitten werden die Pakete *ejb* und *plugin* näher beschrieben, die auch die Schnittstellen für die Kommunikation zwischen Server und Plug-In zur Verfügung stellen.

### 4.1 Paket ejb

Dieses Paket repräsentiert die Anwendungsschicht. Ihm sind alle Enterprise Java-Beans-Klassen zugeordnet. In Abbildung 8 ist das Klassenmodell dieses Paketes dargestellt.

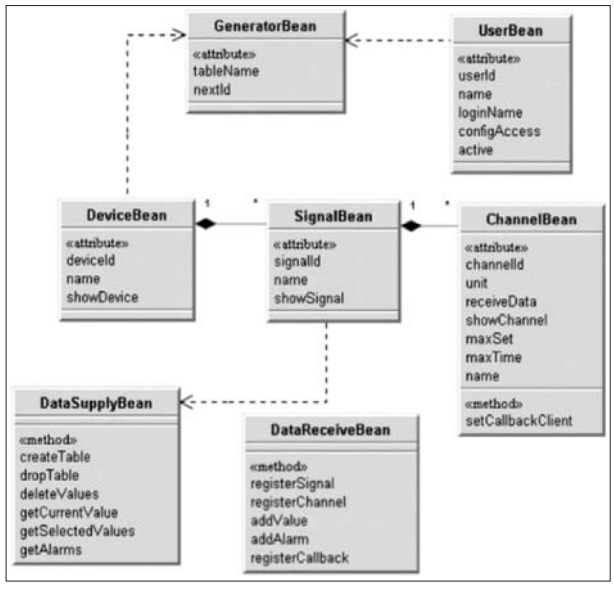


*Abb. 8: Klassendiagramm des EJB-Paketes*

Hauptbestandteil dieses Paketes sind die Klassen *DeviceBean*, *SignalBean* und *ChannelBean*. Sie repräsentieren die entsprechenden Tabellen in der Datenbank. Mittels dieser Klassen werden die Geräte-, Signal- und Kanaldaten in die Datenbank gespeichert. Die *GeneratorBean* und die *UserBean* repräsentieren ebenfalls die entsprechenden Tabellen in der Datenbank. Über die Klasse *Generator-*

*Bean* lassen sich die Klassen *DeviceBean*, *SignalBean*, *ChannelBean* und *UserBean* beim Erstellen eines neuen Datensatzes die nächste Identifikationsnummer geben. Der Aufwand des Erstellens der nächsthöheren Identifikationsnummer ist so geringer, als wenn die Klassen selbst die letzte eingefügte ID über komplexe Datenbankabfragen ermitteln müssten. Die Klasse *UserBean* speichert die Nutzerdaten in der Datenbank. *DataReceiveBean* und *DataSupplyBean* stellen Schnittstellen zum Speichern und Lesen der Wertedaten zur Verfügung. Die *DataReceiveBean* kann dabei über das Remote-Interface von den Plug-Ins und die *DataSupplyBean* nur über das Local-Interface angesprochen werden. Für jedes Plug-In wird dabei eine Instanz der Klasse *DataReceiveBean* erstellt, die erst durch den Garbage Collector vernichtet wird, wenn die Verbindung zu dem Plug-In nicht mehr existiert. Die Klasse *DataReceiveBean* wird von der *SignalBean* auch verwendet, um die Datentabellen zu erstellen oder zu löschen.

### 4.2 Paket plugin

Das Paket *plugin* stellt für die Entwicklung von Plug-Ins eine allgemeine Schnittstelle zur Verfügung. Das Paket kann dabei als Bibliothek in die Plug-Ins eingebunden werden. Das Paket *plugin* übernimmt auch die Kommunikation mit dem Server und das Umwandeln der eindeutigen Identifikationsnummer der Signale und Kanäle des Gerätes und der ID der Signale und Kanäle der Datenbank, mit denen sie auch über mehrere Geräte identifiziert werden können. In Abbildung 9 ist das Klassendiagramm dargestellt.

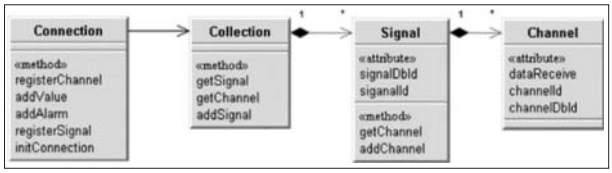


*Abb. 9: Klassendiagramm des Plug-In-Paketes*

Die Klasse *Connection* stellt die allgemeine Schnittstelle für die Plug-Ins zur Verfügung. Weiterhin stellt sie die Verbindung zum Server her und kommuniziert mit diesem. Die Klasse *Collection* verwaltet alle Signale, die das Gerät liefert. In der Klasse *Signal* werden die Daten für das Mapping zwischen Gerät und Server für ein Signal gehalten. Außerdem enthält diese auch eine Referenz auf alle zu dem Signal gehörenden Kanäle. Die Klasse *Channel* verwaltet Daten zum Mapping zwischen Gerät und Server sowie verschiedene Eigenschaften zum Senden von Wertedaten dieses Kanals (z. B. ob Werte gesendet werden sollen).

## 5 Realisierung

Auf der Seite des Applikationsservers werden die entsprechenden EJB-Interfaces öffentlich gemacht, auf die die Plug-Ins zugreifen können. Dafür werden zwei Deployment-Deskriptor-Dateien benötigt. Die Datei ejb-jar.xml enthält Informationen (Name, Typ usw.) zu den EJB. Mit diesen Informationen kann der Applikationsserver die Bean instanziieren und verwalten. Damit die Plug-Ins auf dieses Interface bzw. diese Bean zugreifen können, muss diese bei dem Namens- und Verzeichnisdienst von Java (JNDI = Java Name and Directory Interface) registriert werden. Dafür wird ein Name (JNDI-Name) benötigt. Über diesen Namen kann das Plug-In auf die Bean zugreifen und die Methoden auf dem Server benutzen. Der JNDI-Name wird beim Sun ONE Application Server in der Datei sun-ejb-jar.xml der Bean zugewiesen.

Beispiel ejb-jar.xml:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <display-name>DataReceiveEJB</display-name>
      <ejb-name>DataReceiveBean</ejb-name>
      <home>dsg.server.ejb.data.DataReceiveRemoteHome</home>
      <remote>dsg.server.ejb.data.DataReceiveRemote</remote>
      <ejb-class>dsg.server.ejb.data.DataReceiveBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Beispiel sun-ejb-jar.xml:

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>DataReceiveBean</ejb-name>
      <jndi-name>data_receive</jndi-name>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

Um auf die vom Applikationsserver veröffentlichten Schnittstellen zugreifen zu können, muss das Plug-In in der Programmiersprache Java folgenden Code implementieren:

```
Properties env = new Properties();
env.put(„java.naming.factory.initial“,
      „com.sun.jndi.cosnaming.CNCtxFactory“);
env.put(„java.naming.provider.url“,
      „iiop://192.168.111.1:3700“);

Context initial = new InitialContext(env);
Object objref = initial.lookup(„data_receive“);

DataReceiveRemoteHome home =
      (DataReceiveRemoteHome)PortableRemoteObject.narrow(
      objref, DataReceiveRemoteHome.class);

DataReceiveRemote dataBean = home.create();
```

## 6 Performancetests

Für das Projekt wurden Performancetests für die Kommunikation zwischen einem Plug-In und dem Server durchgeführt. Diese zeigen, wie viele Werte bzw. Datensätze pro Sekunde vom Plug-In zum Server übertragen werden können. Es wurden verschiedene Schnittstellen und Aufgaben auf dem Server sowie die Verwendung von mehreren Geräten bzw. Plug-Ins, die Daten zum Server senden, getestet. Die Testrechner hatten folgende Konfiguration:

***Hardware:***

| | |
|---|---|
| Prozessor | Intel Pentium III mit 1 GHz<br>PCI Bus 64 bit/66 MHz |
| Speicher | 1 GB, 133 MHz SDRAM |
| Netzwerkkarte | EtherExpress PRO/100+ |
| Festplatte | U160-SCSI-Harddisc 17 GB |

***Software:***

| | |
|---|---|
| Betriebssystem | Server: RedHat 7.3,<br>Clients: SuSE Linux 8.1 |
| Programmiersprache | Java JDK 1.4.1 |
| Datenbank | MySql 4.0.14-64 |

Zum Testen der Performance wurde ein Test-Plug-In entwickelt, welches 10000 Daten über eine vorher gewählte Schnittstelle zum Server sendet und die Daten dort je nach Test speichert. Vor und nach dem Senden aller Daten wurde der Zeitstempel des Systems ermittelt, um die gesamte Kommunikationszeit zu berechnen. Die Tests wurden mit den gleichen Schnittstellen und Anforderungen mehrmals wiederholt, um die Stabilität der Ergebnisse zu überprüfen. Im Folgenden sind die einzelnen Tests und deren Testergebnisse beschrieben.

Der erste Test zeigt, mit welchen Datentypen bzw. Klassen einzelne Werte über das IIOP (Internet Inter-ORB Protocol) am schnellsten übermittelt werden können. Das Diagramm in Abbildung 10 stellt die Ergebnisse dieses Tests dar.

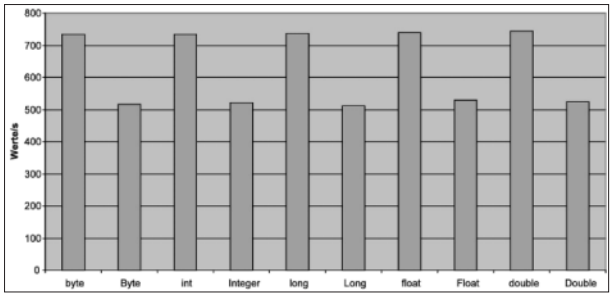


*Abb. 10: Übertragungsgeschwindigkeit eines Wertes*

Die Typen mit den großen Anfangsbuchstaben sind Klassen und die kleingeschriebenen Typen sind die primitiven Datentypen von Java. In dem Diagramm sieht man, dass alle Schnittstellen, bei denen primitive Datentypen übermittelt wurden, deutlich schneller sind und damit auch deutlich mehr Daten in einer Sekunde übertragen können. Dabei ist auffallend, dass alle Klassen bei ca. 525 Werten pro Sekunde und alle primitiven Datentypen bei 725 Werten pro Sekunde liegen. Für eine schnelle Übermittlung der Daten vom Plug-In zum Server sollten primitive Datentypen eingesetzt werden.

Da beim Übertragen von Daten nicht nur ein Wert, sondern ein Datensatz bestehend aus Wert und Kanalnummer oder aus Wert, Kanalnummer und Zeit übertragen werden muss, sollte der zweite Test Aufschluss darüber geben, welche Schnittstelle für die Übertragung am günstigsten ist. Es wurden die Schnittstellen „(channelId long, value double, timestamp long)“, „(channelId long, value double)“ und „(channelId long, value double, timestampDiff byte)“ getestet. Das Ergebnis zeigt die Abbildung 11.

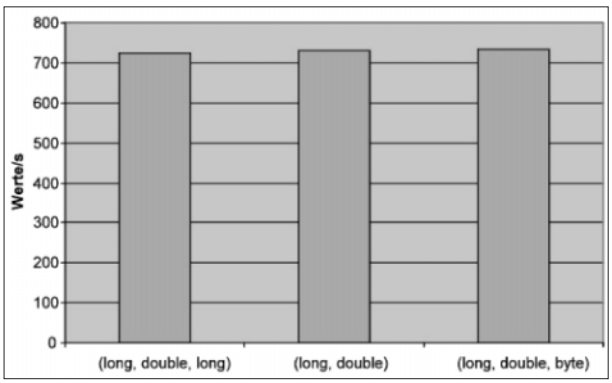


*Abb. 11: Übertragungsgeschwindigkeit von verschiedenen Schnittstellen*

Der Test ergibt, dass bei allen drei Schnittstellen ca. 725 Werte pro Sekunde übertragen werden können. Somit ist es nicht relevant, welche Schnittstelle beim Übertragen von Daten vom Plug-In zum Server benutzt wird.
In beiden vorherigen Tests wurde ermittelt, wie viele Daten zum Server gesendet werden können. Dabei hat der Server die Daten nur empfangen. Er hat sie weder gespeichert, noch hat er irgendwelche anderen Aufgaben erledigt. Der dritte Test soll zeigen, wie viel Daten übermittelt werden können, wenn der Server die Daten speichert und andere Aufgaben bezüglich der Daten erledigt. Für das Senden der Daten wird die Schnittstelle „(channelId long, value double, timestamp long)“ verwendet, da diese auch für die Kommunikation zwischen Plug-In und Server benutzt wird. Das Diagramm der Abbildung 12 zeigt das Ergebnis.

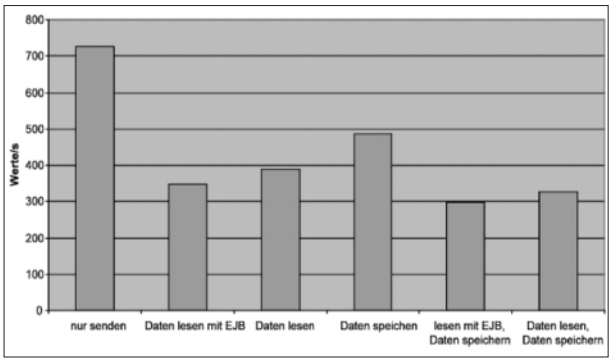


*Abb. 12: Übertragungsgeschwindigkeit mit zusätzlichen Arbeitsaufgaben*

Der erste Balken „nur senden“ wurde aus den Ergebnissen der vorherigen Tests zum Vergleich übernommen. Der zweite Balken „Daten lesen mit EJB“ zeigt das Ergebnis, wenn nach dem Übermitteln der Daten der Server über die EJB-Daten aus der Datenbank lädt, um z. B. zu überprüfen, ob Daten des Kanals gespeichert werden dürfen. Bei Balken Nummer drei („Daten lesen“) wurde das Gleiche getestet wie beim zweiten Balken, mit dem Unterschied, dass keine zusätzlichen EJB zum Lesen der Daten aus der Datenbank benutzt wurden, sondern der Zugriff auf die Datenbank direkt ausgeführt wurde. Das Schreiben in die Datenbank zeigt der Balken Nummer vier „Daten speichern“. Die Balken fünf und sechs geben die Ergebnisse der Kombinationen der beiden Möglichkeiten des Lesens und des Schreiben an.
Das Ergebnis ist nicht überraschend. Das Schreiben der gesendeten Daten in die Datenbank verlängert die Kommunikationsdauer, so dass nur knapp 500 Werte pro

Sekunde übermittelt werden können. Werden zusätzlich noch Daten aus der Datenbank geladen, um z. B. zu überprüfen, ob die übermittelten Daten gespeichert werden können, können nur noch 330 Werte pro Sekunde übermittelt werden, wobei die Benutzung weiterer EJB die Kommunikationsgeschwindigkeit auf nur noch knapp 300 Werte pro Sekunde weiter senkt.

Der letzte Test simuliert eine variierende Anzahl von Geräten bzw. Plug-Ins. Es wurde auf bis zu 14 Rechnern ein Plug-In gestartet. Mittels einer Broadcastnachricht fingen alle zur gleichen Zeit an, Daten über die Schnittstelle „(channelId long, value double, timestamp long)“ dem Server zu senden und in einer Datenbank zu speichern. Das entspricht dem Balken „Daten speichern“ aus dem vorherigen Test. Das Ergebnis dieses Tests stellt das Diagramm der Abbildung 13 dar.

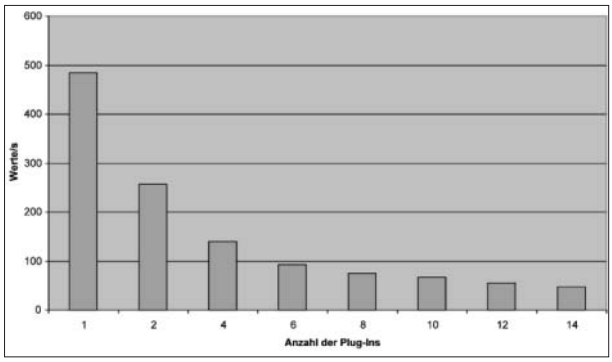


*Abb. 13: Mittlere Übertragungsgeschwindigkeit der Plug-Ins*

Der Test zeigt, dass je mehr Rechner dem Server Daten übermitteln, desto weniger Werte können pro Plug-In übermittelt werden. Bei genauer Betrachtungsweise der Ergebnisse ist erkennbar, dass es sich um eine umgekehrte Proportionalität handelt. Der Server kann immer nur 500 Werte pro Sekunde empfangen und speichern, egal wie viele Plug-Ins Daten übermitteln.
Bei allen Tests war der Prozessor des Servers, während die Plug-Ins Daten zum Server sendeten, ausgelastet. Daraus folgt, dass der Applikationsserver beim Empfang von Daten bzw. speziell wenn die Plug-Ins Methoden auf dem Server aufrufen, so viel Systemleistung in Anspruch nimmt, dass die Kommunikationsgeschwindigkeit darunter leidet.

## 7 Zusammenfassung und Ausblick

Es wurde ein Telematik-System entwickelt, das die Fernüberwachung und -steuerung von vernetzten Geräten unterstützt. Dabei wurden bewährte (z. B. CORBA) sowie jüngere Technologien (z. B. Enterprise Java-Beans) eingesetzt.
Vor der Entwicklung musste zuerst eine Architektur für das System gefunden werden. Es wurden vier mögliche Systemarchitekturen untersucht und verglichen, von denen die vierte gewählt wurde, die einen Applikationsservers benutzt, der WWW/WAP und CORBA unterstützt. Die vielen Vorteile und die wenigen Nachteile bei der Verwendung dieser Architektur haben den Ausschlag gegeben. Als Applikationsserver wird der Sun ONE Application Server in der einfachsten Version, der *Plattform Edition*, benutzt. Diese unterliegt zwar einigen Beschränkungen, die aber für dieses System nicht relevant sind.
Für die Entwicklung wurden die Anwendungsfälle „Daten lesen“, „Daten schreiben“ und „Daten konfigurieren“ definiert.
Anhand der Anwendungsfälle wurden Pakete definiert. Für den Anwendungsfall „Daten lesen“ wurden die Pakete *web* und *applet* konzipiert. *web* ist für die textliche, *applet* für die grafische Darstellung verantwortlich. „Daten schreiben“ übernimmt das Paket *plugin* und „Daten konfigurieren“ das Paket *config*. Zusätzlich werden Klassen dem Paket *ejb* zugeordnet. Dieses Paket stellt die Anwendungsschicht dar.
In einem durchgeführten Performancetest für den Server wurden verschiedene Schnittstellen auf die Übertragungsgeschwindigkeit, sowie verschiedene Arbeitsaufgaben auf dem Server getestet. Dabei wurde festgestellt, dass der Applikationsserver bei den angegebenen Rechnerkonfigurationen ca. 500 Werte/s von Plug-Ins empfangen und in der Datenbank speichern kann. Die Integration der Komponenten verlief ebenso erfolgreich wie das Testen dieser im Einzelnen und im Gesamten.

Für eine schnellere Übertragungsgeschwindigkeit, wie sie z. B. für hochabgetastete Wave-Daten benötigt wird, ist eine Variante des Systems ohne Applikationsserver in Betracht zu ziehen, z. B. nach dem Modell 2 (siehe Abb. 3). Die Übertragungsgeschwindigkeit kann dadurch, das zeigen einige Testversuche, mehr als verdoppelt werden und ist dann nicht so stark von der Anzahl der Plug-Ins abhängig.
In der Zukunft können ohne großen Aufwand noch allgemeine Funktionen für die Interaktion mit den Geräten implementiert werden. So kann das System beispielsweise auch für Fernbedienung und -steuerung in industriellen und privaten Anwendungsszenarien eingesetzt werden.

## Literatur

[1] Christian Ullenboom: Java ist auch eine Insel. Galileo Press, 2003.

[2] Mark Wutka: J2EE Developer’s Guide. Markt + Technik, 2002.

[3] Cay S. Horstmann, Gary Cornell: Core Java 2; Band 2 Expertenwissen. Markt + Technik, 2002.

[4] Martin Fowler, Kendall Scott: UML Konzentriert. Addison-Wesley, 2000.

[5] Stefan Denninger, Ingo Peters: Enterprise JavaBeans 2.0; 2. Auflage Addison- Wesley, 2002.

[6] Andreas Eberhart, Stefan Fischer: Java-Bausteine für E-Commerce-Anwendungen; 2., aktualisierte und erweiterte Auflage, Hanser, 2001.

[7] Sun Microsystems. The Java Tutorial: RMI. http://java.sun.com/docs/books/tutorial/rmi/index.html.

[8] Sun Microsystems. The Java API: RMI. http://java.sun.com/j2se/1.4/docs/api

[9] Sun Microsystems. The Java Tutorial: IDL. http://java.sun.com/docs/books/tutorial/idl/index.html

[10] Sun Microsystems. The Java API: IDL. http://java.sun.com/j2se/1.4/docs/api

[11] Sun Microsystems. Sun ONE Application Server. http://wwws.sun.com/software/products/appsrvr/home appsrvr.html

[12] Thomas Behrens: Entwicklung eines Gatewaysystems für WWW- und WAP-basiertes Gerätemonitoring. Diplomarbeit TFH Wildau/Universität Potsdam, 2003.

## Autoren

**Prof. Dr. rer. nat. Ralf Vandenhouten**
Technische Fachhochschule Wildau
Fachbereich Ingenieurwesen/Wirtschaftsingenieurwesen
Lehrstuhl für Telematik
Tel. +49 3375 508-359
E-Mail: rvandenh@igw.tfh-wildau.de

**Dipl.-Inform. Thomas Behrens**
Technische Fachhochschule Wildau
Fachbereich Ingenieurwesen/Wirtschaftsingenieurwesen
Lehrstuhl für Telematik
Tel. +49 3375 508-616
E-Mail: tbehrens@igw.tfh-wildau.de

**Prof. Dr. Bettina Schnor**
Universität Potsdam
Institut für Informatik
August-Bebel-Straße 89, 14482 Potsdam
Tel. +49 331 977-3120
E-Mail: schnor@cs.uni-potsdam.de