Bachelorarbeit

Florian Miess, Jörg Seifert

18. Mai 2007

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Bachelorarbeit -

Planung und Realisierung einer Middleware zur asynchronen Kommunikation zwischen mobilen Anwendungen und Webanwendungen

vorgelegt von

Florian Miess, Jörg Seifert

Referent : Prof. Dr. Michael Massoth
Korreferent : Prof. Dr. Peter Wollenweber

Mai 2007
Fachbereich Informatik der Hochschule Darmstadt

Extern durchgeführt bei
Dymacon Business Solutions GmbH,
Darmstadt

Betreuer : Klaus Peter Klüter

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.
Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 18. Mai 2007 _____

Florian Miess

Darmstadt, den 18. Mai 2007 _____

Jörg Seifert

Kurzfassung

Dieses Dokument stellt die Realisierung einer Middleware vor, die für eine verlustfreie, asynchrone Datenübertragung zwischen einem mobilen Endgerät und Serverdiensten sorgt. Eine Datenübertragung zwischen mobilen Anwendungen und durch das Internet erreichbare Dienste wird häufig durch eine schlechte oder während der Übertragung abreißende Verbindung erschwert. Die Middleware kapselt die Datenübertragung und stellt Anwendungen die Möglichkeit zur Verfügung, Daten asynchron zu senden und zu empfangen. Erreicht wird dies durch eine Komponente auf Client- und Serverseite. Bei der Entwicklung weiterer mobiler Anwendungen kann auf diese Funktionen zurückgegriffen werden, und eine erneute Umsetzung der Übertragungsproblematiken entfällt. Der Service kann durch weitere Module in seinen Funktionen erweitert werden, um zusätzliche Dienste im Internet oder Intranet anzusprechen. Realisiert wurde die Middleware für die BlackBerry-Umgebung des Unternehmens RIM (Reasearch in Motion Limited).

Zunächst werden einige Grundlagen von mobilen Anwendungen behandelt, zum Beispiel aktuelle Übertragungskapazitäten und bereits bestehende Möglichkeiten zur Prozessoptimierung. Anschließend werden verschiedene Lösungswege auf ihre Realisierbarkeit hin untersucht und verglichen. Nach der Entscheidung für eine dieser Optionen wird die Middleware geplant und Entwurfs- und Designentscheidungen erläutert. Danach erfolgt anhand des Entwurfs die eigentliche Realisierung. Im Anschluss werden zwei Anwendungen vorgestellt, die auf der neu entwickelten Middleware aufsetzen. Abschließend werden die Ergebnisse der Arbeit zusammengefasst.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

1 Einleitung

1.1 Vorbemerkung

Diese Arbeit ist ein Gemeinschaftsprojekt der Studenten Florian Miess (Matrikelnummer 702751) und Jörg Seifert (Matrikelnummer 702827). Die Tabelle 1.1 gibt einen Überblick über die Autoren der jeweiligen Kapitel.

Tabelle 1.1: Kapitelautoren

Kapitel	**Autor**
Kapitel 1	Florian Miess und Jörg Seifert
Kapitel 2	Florian Miess und Jörg Seifert
Kapitel 3	Florian Miess und Jörg Seifert
Kapitel 4.1	Florian Miess
Kapitel 4.2	Florian Miess
Kapitel 4.3	Jörg Seifert
Kapitel 4.4	Jörg Seifert
Kapitel 5.1	Florian Miess und Jörg Seifert
Kapitel 5.2	Jörg Seifert
Kapitel 5.3	Florian Miess
Kapitel 6.1	Jörg Seifert
Kapitel 6.2	Florian Miess
Kapitel 6.3	Florian Miess und Jörg Seifert
Kapitel 7	Florian Miess und Jörg Seifert

1.2 Kontext

Diese Bachelorarbeit beschäftigt sich mit der asynchronen Kommunikation zwischen mobilen Anwendungen und Webanwendungen. Das Projekt wurde im Unternehmen Dymacon Business Solutions GmbH absolviert. Dort wurde die hier bearbeitete Fragestellung untersucht und die Lösung durch eine Middleware implementiert. Zusätzlich wurden zwei Anwendungen erstellt, die Aufträge generieren und an das Unternehmensnetzwerk weiterleiten können. Diese Anwendung benutzen einen Barcode-Scanner und ein Unterschriftenfeld, mit deren Hilfe Artikel eingescannt und eine Bestellung generiert wird. Beide Anwendungen samt Middleware wurden auf der CeBIT 2007 vorgestellt, außerdem ist eine Anwendung bereits im Unternehmenseinsatz. Die Middleware und die Anwendungen wurden für die BlackBerry Geräte des Unternehmens Research In Motion Limited (RIM) erstellt.

Die Dymacon Business Solutions GmbH ist eines der führenden Systemhäuser im Bereich der mobilen Datenkommunikation. Der Name Dymacon ist dabei die Abkürzung für Dynamic mobile Application and Consulting. Die Firma ist Mitglied im comTeam Systemhaus-Verbund. Der Geschäftssitz ist in Darmstadt.

1.3 Motivation und Kundennutzen

Bei der Arbeit mit mobilen Anwendungen stößt man häufig auf das Problem, dass der Empfang beziehungsweise die Verbindung, aufgrund von Netzschwankungen, abreißt. Dies ist ärgerlich für den Entwickler und für den Anwender der Applikation. Der Entwickler ist gezwungen, Funktionalitäten, die mit der Kommunikation nach Außen zu tun haben, abzusichern und Fehlerbehandlungen und Routinen zu schreiben, die das Problem umgehen. Dies ist ein erheblicher Mehraufwand, der viel Zeit und Geld kostet. Je nach Implementierung muss der Anwender bei schlechter Verbindung mehrfach versuchen, die gewünschte Aktion auszuführen und es ist nicht sicher, ob die Nachricht gar nicht oder eventuell mehrfach gesendet wurde. Besonders im Geschäftsbereich kann dies kritische Folgen haben. Eine Kommunikationsplattform kann diese Probleme kapseln. Dadurch verringert sich der Zeitaufwand bei der Entwicklung neuer Anwendungen immens, da nur noch die neu benötigten Funktionen der Anwendung erstellt werden

müssen. Die Kosten werden gesenkt und der Entwickler kann sich auf die Aufgaben der neuen Anwendung konzentrieren und so die Qualität verbessern.

Motivation für die Arbeit war es demnach, die Kommunikation für den Entwickler so einfach und komfortabel wie möglich zu gestalten. Er soll sich nur noch um das „Was“, und nicht mehr um das „Wie“ soll etwas transportiert werden kümmern müssen. So sollen Qualität, Kosten und die Akzeptanz der Benutzer für die Anwendungen verbessert werden.

1.4 Aufgabenstellung

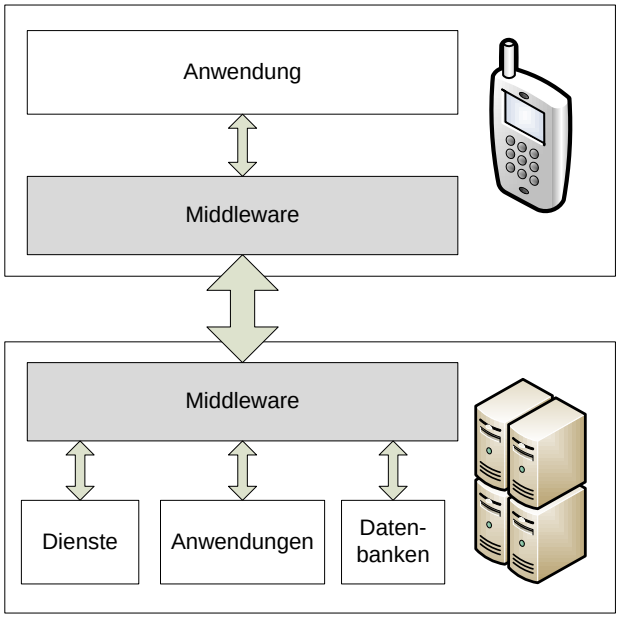


Abbildung 1.1: Kommunikationsplattform

Die Aufgabenstellung im Unternehmen war es, einen allgemein nutzbaren Service für asynchrone Kommunikation für künftige Projekte zu entwickeln. Er soll als Basis für mobile Anwendungen auf BlackBerry Geräten entstehen und Kommunikationsprobleme, die für mobile Geräte existieren, kapseln beziehungsweise lösen. Dieser Service sollte modular erweiterbar und für möglichst viele Anwendungsszenarien nutzbar sein. Dem

Entwickler soll er ermöglichen, weitere, darauf aufbauende Anwendungen zu erstellen, ohne sich um die gekapselten Probleme kümmern zu müssen. Bild 1.1 gibt einen Eindruck dieses Services. Zu ausführlicheren Anforderungen und deren Herleitung, siehe Kapitel 3 und folgende.

1.5 Gliederung der Arbeit

Dieser Abschnitt erläutert kurz den Inhalt der folgenden Kapitel. Es dient dazu, den roten Faden der Arbeit zu verdeutlichen und soll den Umgang beziehungsweise das Arbeiten mit diesem Dokument erleichtern.

- **Kapitel 2: Grundlagen**
  Einige theoretische Grundlagen und Zusammenhänge, die zum besseren Verständnis der Arbeit beitragen, werden hier erläutert.
- **Kapitel 3: Anforderungsanalyse**
  Eine Analyse der Anforderungen und daraus resultierende Aufgaben und Fragestellungen werden diesem Kapitel behandelt.
- **Kapitel 4: Stand der Forschung und Machbarkeitsstudie**
  Hier wird der aktuelle Stand der Forschung beziehungsweise die aktuellen Entwicklungen im Bereich mobiler Entwicklungen beleuchtet. Dann werden in einer Machbarkeitsstudie die verschiedenen Möglichkeiten der Realisierung diskutiert und eine Variante ausgewählt und begründet.
- **Kapitel 5: Entwurf und Design**
  In diesem Kapitel wird der eigentliche Aufbau der Middleware beschrieben. Es werden das Design und Entwurfsentscheidungen dafür genannt.
- **Kapitel 6: Realisierung**
  Im sechsten Kapitel wird die tatsächliche Realisierung eines Prototyps in Java vorgestellt. Danach wird kurz auf zwei Anwendungen eingegangen, die auf der Middleware aufbauen.
- **Kapitel 7: Zusammenfassung und Ausblick**
  Eine Zusammenfassung der Ergebnisse und eine Bewertung dieser Arbeit erfolgt

in diesem Kapitel, außerdem ein abschließender Ausblick auf zukünftige Einsatzmöglichkeiten und Erweiterungen der Middleware.

2 Grundlagen

Dieser Abschnitt beschreibt die notwendigen Grundlagen für das tiefere Verständnis der vorliegenden Arbeit und erklärt die für die kommenden Kapitel wichtigen Begriffe und Konzepte.

2.1 Mobile Anwendungen

Mit der Einführung mobiler Technologien entstanden neue Arten von Anwendungen. Hauptmerkmal dieser Anwendungen ist es, ortsunabhängige Zugriffe auf Informationen und Daten zu erlauben. Im Folgenden werden die dafür nötigen Technologien erläutert und danach gezeigt, wie dadurch im Unternehmen Prozesse optimiert werden können.

Technik Auf technischer Ebene existieren zum einen die verschiedenen mobilen Endgeräte, zum anderen die verschiedenen Übertragungsmöglichkeiten von Daten vom Endgerät zum Unternehmen. Auf diese wird im Folgenden kurz eingegangen.

Auf dem Markt existieren eine Vielzahl von mobilen Geräten. Diese umfassen Mobiltelefone und PDAs, aber auch Mischlösung wie Smartphones. Alle diese Systeme unterliegen einigen Einschränkungen. Obwohl sie immer leistungsfähiger werden sind die Ressourcen, im Gegensatz zu stationären Geräten wie PCs, gering. Schwache Hardware-Plattformen, kleine Displays, geringer Speicher und begrenzte Eingabemöglichkeiten sind nur einige dieser Beschränkungen. Hierzu kommen eingeschränkte Übertragungskapazitäten, da die Kommunikation der Endgeräte zur Zeit noch hauptsächlich durch die GSM-Netze der Mobilfunkanbieter erfolgt. Erst UMTS stellt wesentlich höhere Datenraten zur Verfügung.

In Europa sind zurzeit folgende verbindungslose, paketorientierte Übertragungverfahren im Mobilfunk verbreitet:

- GPRS (General Packet Radio Service): Eine auf dem GSM-Netz aufbauende Technologie. Sie erweitert GSM um ein paketorientiertes Übertragungverfahren.
- EDGE (Enhanced Data Rate for Global Evolution): Ein geändertes Modulationsverfahren für GPRS um die Datenraten zu steigern und die Netzauslastung zu verbessern.
- UMTS (Universal Mobile Telecommunication System): Standard der 3. Mobilfunkgeneration (3G). Die Übertragung basiert auf komplett neuer Netztechnik.
- HSDPA/HSUPA (High Speed Downlink/Uplink Packet Access): Protokollzusätze zu UMTS, die durch ein geändertes Kodierungsverfahren die Übertragungsraten steigern.

Für Übertragungsraten siehe Tabelle 2.1. Die Werte sind grobe Richtwerte und von verschiedenen Einflüssen, zum Beispiel der Netzauslastung, abhängig.

Tabelle 2.1: Übertragungsraten in der Praxis (Richtwerte)

Name	Geschwindigkeit	Round-Trip-Zeiten
GPRS	Downlink: 62,4 kBit/s Uplink: 31,2 kBit/s	500 ms und mehr
EDGE	Downlink: 200 kBit/s Uplink: 100 kBit/s	300 bis 400 ms
UMTS	Downlink: 384 kBit/s Uplink: 64 kBit/s	170 bis 200 ms
HSDPA Stufe 1	Downlink: 1,8 MBit/s Uplink: 384 kBit/s	60 bis 70 ms
HSDPA Stufe 2 (2007)	Downlink: 3,6 MBit/s Uplink: 1,8 MBit/s	60 bis 70 ms
HSDPA Stufe 3 (2008)	Downlink: 7,2 MBit/s Downlink: 3,6 MBit/s	60 bis 70 ms

Neben dem schnellen Fortschritt der technischen Möglichkeiten nimmt auch die Verbreitung mobiler Technologien immer schneller zu. So nutzten Ende 2005 rund 2,3 Millionen

Deutsche[1] UMTS. Anfang 2007 wurde in Deutschland mit einem Anstieg auf neun Millionen UMTS-Nutzer gerechnet. Dieser Anstieg und die Tatsache, dass 2005 bereits 95% der Deutschen ein Mobiltelefon besaßen, zeigen, dass mobile Anwendungen zunehmend an Bedeutung im Markt gewinnen.[2]

Prozesse optimieren Ein durchgängiger Einsatz von GPRS oder UMTS erlaubt eine kontinuierliche Synchronisation der Daten zwischen mobilem Endgerät und Unternehmens-IT, wann immer eine entsprechende Verbindung besteht. Die Chancen einer solchen Unternehmenslösung zeigen sich, wenn die Informationsflüsse traditionell organisierter Prozesse mit denen, die bei der Nutzung von mobilen Lösungen entstehen, verglichen werden.

Ohne mobile Unternehmenslösungen bestehen die Informationsflüsse zwischen mobilen Mitarbeitern und den Backend-Systemen[3] aus mehreren Schritten, die jeweils mit Medienbrüchen verbunden sind. Trotz unterschiedlicher Prozesse ähneln sich deren Abläufe (siehe Abbildung 2.1).

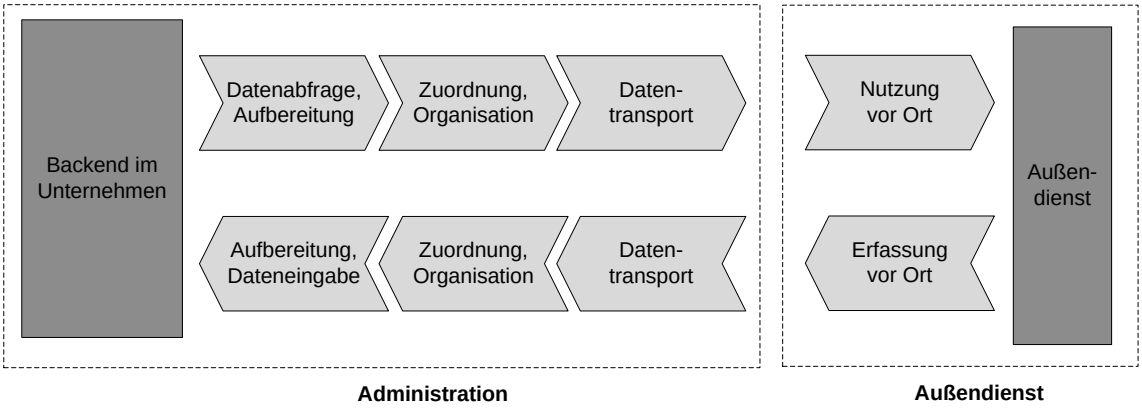


Abbildung 2.1: Informationsfluss ohne mobile Lösung (siehe Wichmann u. Stiehler (2004))

Innerhalb des Unternehmens fallen mehrere Arbeitsschritte an:

[1] Weltweit 45,3 Millionen
[2] BITKOM (2006)
[3] Hier: Systeme innerhalb der Unternehmens-Infrastruktur

- **Datenabfrage und Aufbereitung** Daten, die elektronisch vorliegen, müssen für den Außendienst aufbereitet werden, z.B. Informationen abfragen und ausdrucken.
- **Zuordnung und Organisation der Verteilung** Mitarbeiter brauchen zum richtigen Zeitpunkt die richtigen Informationen.
- **Datentransport** Der Außendienstmitarbeiter kann die Daten selbst im Unternehmen abholen oder auch geschickt bekommen (E-Mail, Fax, Telefon). In beiden Fällen werden zeit- und kostenaufwändige Schritte durch Mitarbeiter notwendig.

Allerdings generiert auch der Außendienstler Daten. Um diese im Unternehmen zu nutzen, sind ebenfalls mehrere Schritte nötig:

- **Datenerfassung** Dies geschieht meist handschriftlich auf Formularen oder Notizblättern.
- **Datentransport** Hier fallen dieselben Wege und Probleme an, die beim Transport der Daten zum Außendienstler hin existieren.
- **Organisation des Dateneingangs und Zuordnung der Daten** Die vom Außendienstler erfassten Daten müssen im Unternehmen wieder gesammelt und dem richtigen Auftrag, Kunden oder Vorgang zugeordnet werden.
- **Aufbereitung und Dateneingabe** Die Daten müssen, dem Backend entsprechend, aufgearbeitet und elektronisch erfasst werden.

Beim Einsatz von mobilen Unternehmenslösungen können Informationen und Daten vor Ort aus dem Backend-System abgefragt und auch wieder eingespeist werden. Daraus ergibt sich der in Abbildung 2.2 dargestellte Ablauf.

So entfällt die doppelte Erfassung von Daten - schriftlich vor Ort und elektronisch für die Eingabe in das Backend-System. Prozessschritte, die für das Aufbereiten der Daten für den Außendienst beziehungsweise für das Backend-System nötig waren, entfallen ganz. Dadurch werden Medienbrüche vermieden und Fehlerquellen wie falsche Zuordnung von Daten oder unleserliche Informationen reduziert. Außerdem kann der Datenaustausch zwischen Außendienstlern und der Zentrale orts- und zeitnah gestaltet werden. Daten stehen somit dann zur Verfügung, wenn sie gebraucht werden. Damit erhöht sich die Flexibilität und die Reaktionszeit auf unvorhergesehen Ereignisse und damit auch die Kundenzufriedenheit.

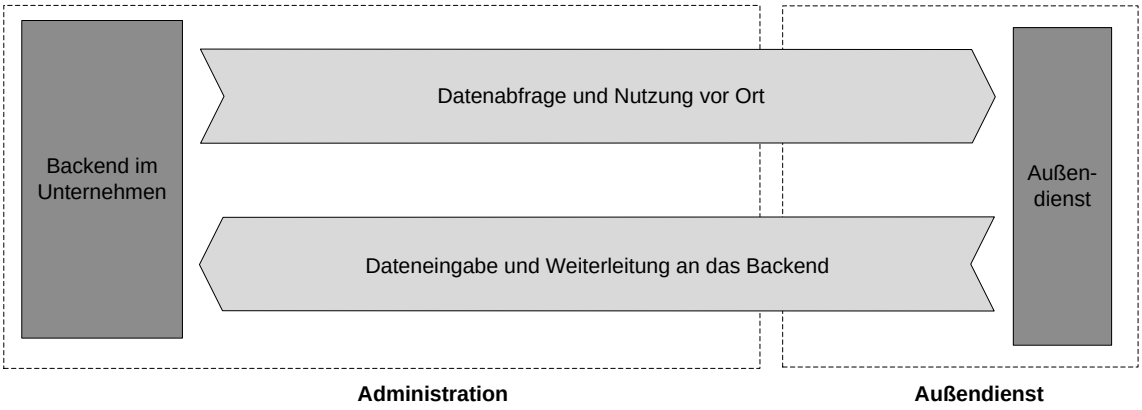


Abbildung 2.2: Optimierte Prozesskette mit mobiler Lösung (siehe Wichmann u. Stiehler (2004))

Die mobile Prozessoptimierung bewirkt also[4]:

- Einsparung von Prozessschritten, da Doppelerfassungen und Schritte zur Transformation der Daten zwischen Mitarbeiter und Backend-Systemen wegfallen.
- Vermeidung von Medienbrüchen, da die Daten direkt elektronisch erfasst und abgefragt werden können.
- Die zeit- und ortsnahe Bereitstellung von Informationen.

und dadurch:

- Der Personaleinsatz wird optimiert.
- Materialkosten und Sachkosten werden gespart.
- Arbeitsabläufe werden beschleunigt.
- Die Datenqualität wird erhöht.
- Eine verbesserte Kundenbeziehung wird erreicht.

[4] Wichmann u. Stiehler (2004)

Beispiele Beispiele für die Nutzung mobiler Lösungen sind das Erfassen von Kundenaufträgen, die Überprüfung von Lieferfähigkeiten direkt vor Ort oder Flug- und Zugreservierungen unabhängig vom Standort. Die meisten Einsatzgebiete sind zurzeit im B2C- und im unternehmensintern Bereich zu finden. Aber auch im B2B-Bereich werden immer mehr mobile Systeme eingesetzt. Mögliche Anwendungszenarien sind hier zum Beispiel die Just-in-Time Materiallieferung, in der mit Hilfe von mobilen Geräten der Gütertransport überwacht wird, Verzögerungen automatisiert an Kunden gemeldet werden und Abläufe in der Produktion geändert werden können.[5]

Wenn im weiteren Verlauf dieser Arbeit von mobilen Anwendungen gesprochen wird, werden immer Client-Server Anwendungen gemeint, die Daten über Netzwerke austauschen, also von verteilten Anwendungen (siehe Abschnitt 2.2).

2.2 Verteilte Systeme

Nach der Definition von Tanenbaum[6] ist ein verteiltes System „ein System mit mehreren Rechnern bzw. Prozessoren, auf denen mindestens eine verteilte Anwendung lauffähig installiert ist“. Eine verteilte Anwendung ist eine „Anwendung, die auf mehreren Rechnern bzw. Prozessoren abläuft und unter diesen Informationen austauscht“[6]. Die mobilen Anwendungen sind demnach eine spezielle Form von verteilten Anwendungen. In diesem Kapitel werden deswegen kurz die Eigenschaften dieser verteilten Systeme beschrieben und die Eigenheiten, die durch den Namenszusatz „mobil“ hinzu kommen, erläutert.

Insgesamt besteht ein verteiltes System aus:

- einer Menge von autonom arbeitenden Programmen, die
- auf unterschiedlichen Rechnern ablaufen und
- ohne zentrale Kontrolle
- miteinander kommunizieren, um

[5] Teichmann u. Lehner (2002)
[6] Tanenbaum u. van Steen (2003)

- kooperativ eine Aufgabe zu erfüllen.

Eine zentrale Rolle in verteilten Systemen, die für mobile Anwendungen besonders wichtig ist, ist demnach die Kommunikation der Anwendungsteile untereinander. Die Regeln nach denen dieser Informationsaustausch stattfindet, werden als Protokolle bezeichnet. Tabelle 2.2 zeigt die verschiedenen Kommunikationsformen und die entsprechenden Bezeichnungen in verteilten Systemen.

Tabelle 2.2: Kommunikationsformen

	asynchron	**synchron**
meldungsorientiert	Datagramm	Rendezvous
auftragsorientiert	asynchroner entfernter Dienstaufruf	synchroner entfernter Dienstaufruf

Synchrone Kommunikation bezeichnet den Informationsaustausch zwischen Systemen, bei welchen der Fragesteller wartet, bis eine Antwort eintrifft. Die verteilte Anwendung blockiert die Ausarbeitung und wartet. In dieser Zeit können keine weiteren Aufgaben ausgeführt werden.

Unter **asynchroner Kommunikation** versteht man das Verhalten, bei dem auf eine Anfrage nicht unmittelbar auch eine Antwort erfolgen muss. Die Anwendung blockiert demnach nicht und arbeitet weitere Aufgaben ab.

Meldungsorientiertes Verhalten bedeutet, dass nur Einwegnachrichten versendet werden, der Absender erhält keine Antworten.

Bei **auftragsorientiertem** Verhalten dagegen werden Aufträge versendet, auf welche eine Antwortnachricht erfolgt: Der Absender erhält ein Ergebnis zurück.

Bei Asynchronität ergeben sich Probleme bzw. Punkte, die es zu beachten gilt. Bei asynchronem Verhalten ist es möglich, mehrere Anfragen zu stellen. Die Antworten zu diesen Anfragen treffen aber zu einem späteren Zeitpunkt ein und sind möglicherweise in falscher Reihenfolge. Es existieren verschiedene Fehlersemantiken, um diese Szenarien zu behandeln (Tabelle 2.3). In diesen Bereich fällt auch die Datensynchronität. Die Synchronisation ist ein Problem, da Datensätze gleichzeitig an getrennten Orten geändert werden können. Es existieren verschiedene Verfahren, um Unterschiede in den Daten

zu beseitigen, zum Beispiel das Bevorzugen der aktuelleren Daten oder das Bevorzugen eines bestimmten Gerätes.[7]

Tabelle 2.3: Fehlersemantiken

Bezeichnung	Merkmal
Maybe	Keine Fehlerbehandlung, Operationen werden einmal oder gar nicht ausgeführt
At-Least-Once	Die Anfrage wird mindestens einmal und bis zum Erfolg ausgeführt
At-Most-Once	Aufträge werden höchstens einmal ausgeführt, indem Duplikate verworfen werden
Exactly-Once	Anfragen werden genau einmal ausgeführt

2.3 Schichtenmodell

Zum Verständnis der Arbeit ist es wichtig, die Begriffe Schichten, Protokoll und Schnittstelle zu kennen (Abbildung 2.3).[8]

- **Schichten**
  Software wird zur Reduzierung der Komplexität in Schichten unterteilt. Eine Schicht ist eine modellhafte Vorstellung von verschiedenen Ebenen eines Systems, die jeweils unterschiedliche Aufgaben erfüllen. Oder anders: Eine Funktionseinheit komplexer Modelle.

- **Protokoll**
  Ein Protokoll ist eine Kommunikationsvereinbarung zwischen Schichten auf gleicher Ebene. Es definiert das Format und die Reihenfolge von Nachrichten, die zwischen zwei oder mehr kommunizierenden Einheiten ausgetauscht werden, sowie die Aktionen, die bei der Übertragung und/oder beim Empfang einer Nachricht oder eines anderen Ereignisses unternommen werden. Nur so kann ein reibungsloser und kontrollierter Datenaustausch in Netzwerken erfolgen.

- **Schnittstelle**
  Schnittstellen bestimmen Signale oder Nachrichten zwischen den angrenzenden

[7] Schütte (2006b)
[8] Massoth (2006)

Schichten eines eines Systems. Sie definieren also die Kommunikationsregeln zwischen den Schichten.

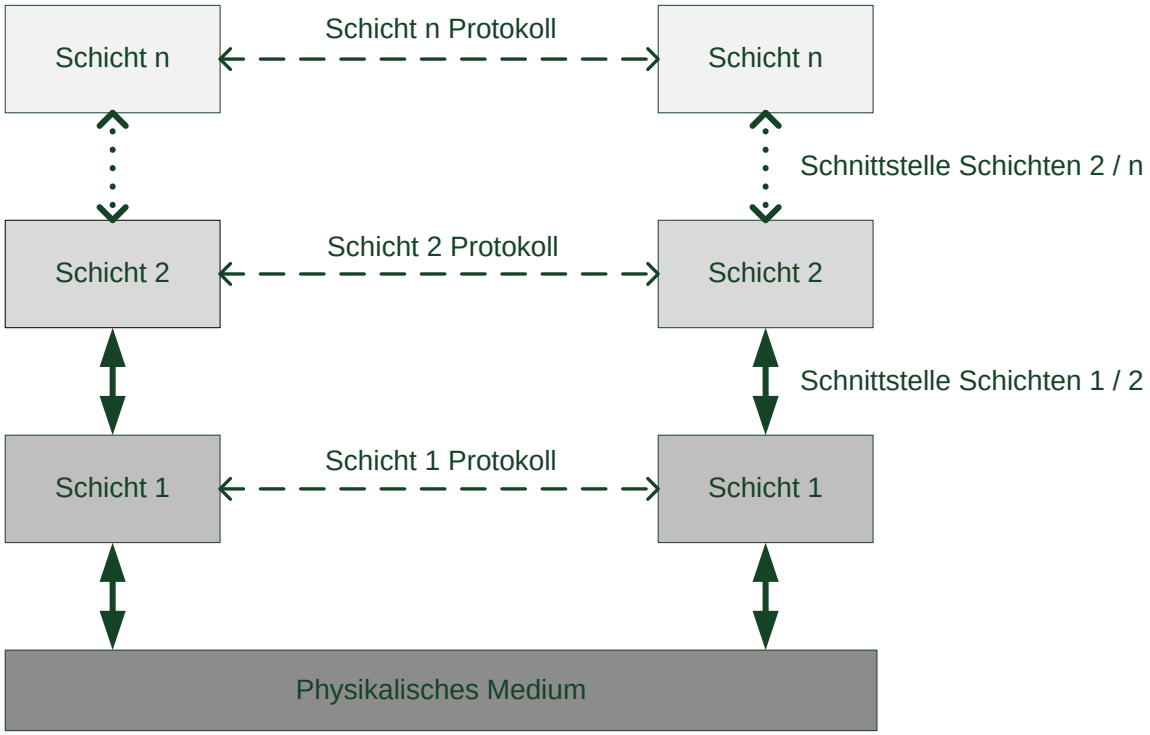


Abbildung 2.3: Schichtenmodell

2.4 BlackBerry

BlackBerry ist der Name einer von dem kanadischen Unternehmen Research In Motion (RIM) entwickelten Lösung für eine drahtlose Kommunikationsplattform, primär für E-Mail und Personal Information Manager (PIM)-Daten. Diese Lösung umfasst eine Client-Server-Architektur, das proprietäre BlackBerry-Protokoll zwischen Clients und Servern und eine Serie von BlackBerry-fähigen Endgeräten[9] (Abbildung 2.4). Die von RIM selbst hergestellten Geräte werden ebenfalls als BlackBerrys bezeichnet. Daneben

[9] Wikipedia Foundation (2007a)

existieren auch Geräte anderer Hersteller, die sich in die BlackBerry Infrastruktur integrieren lassen. Ihnen gemeinsam ist die Tatsache, dass es sich um Smartphones[10] handelt. Die BlackBerry Infrastruktur nutzt einen Push-Dienst, mit dem E-Mails und PIM-Daten drahtlos abgeglichen werden können. Diese Synchronisation geschieht automatisch, sobald das Gerät sich im Empfangsbereich des entsprechenden Mobilfunkanbieters befindet. So entfällt das Problem verschiedener und nicht aktueller Daten.

Ein weiterer Vorteil besteht darin, dass die Geräte sich fernwarten beziehungsweise fernsteuern lassen. Die BlackBerry Infrastruktur erlaubt das Erstellen und Anwenden von IT-Richtlinien. So ist es zum Beispiel möglich, Geräte durch Fernzugriff zu sperren oder den Speicher zu löschen, um den Schutz von Unternehmensdaten bei Verlust des BlackBerry zu gewährleisten.

Für die Funktionalität des Push-Dienstes stellt das Unternehmen RIM weltweit Server bereit, die den Kontakt zwischen den lokalen Servern und den Endgeräten in den Mobilfunknetzen herstellen. Sie leiten die eingehenden Nachrichten an die Zielgeräte weiter und nehmen Nachrichten der Mobilgeräte durch die Mobilfunkanbieter an. Hierfür existieren in den Unternehmen Server, sogenannte BE Server (BlackBerry Enterprise Server), die die Unternehmensdaten verschlüsseln und dann zu den Servern von RIM weiterleiten. Auf diesem BES läuft ein Service - Mobile Data System (MDS), der Entwicklern eine Schnittstelle für Push-Nachrichten zur Verfügung stellt.

Die normale Kommunikation der Geräte erfolgt über das GSM-Netz oder, bei neueren Geräten, über UMTS. Die Datenübertragung erfolgt dementsprechend durch GPRS oder EDGE, beziehungsweise über HSDPA, bei älteren BlackBerrys über spezielle Pager-Netze.

Daten, die zwischen dem BlackBerry Enterprise Server (BES) und dem BlackBerry Gerät ausgetauscht werden, werden standardmässig verschlüsselt. Dies geschieht entweder nach dem Advanced Encryption Standard (AES) oder dem Triple Data Encryption Standard (Triple DES). Die implementierte kryptographische Softwarekomponente besitzt die FIPS 140-2-Validierung[11]. [12][13][14]

[10] Kombination aus Mobiltelefon und PDA.
[11] Sicherheitsstandard der USA für Kryptographiemodule.
[12] Research in Motion (2007)
[13] Wikipedia Foundation (2007c)
[14] CMVP (2007)

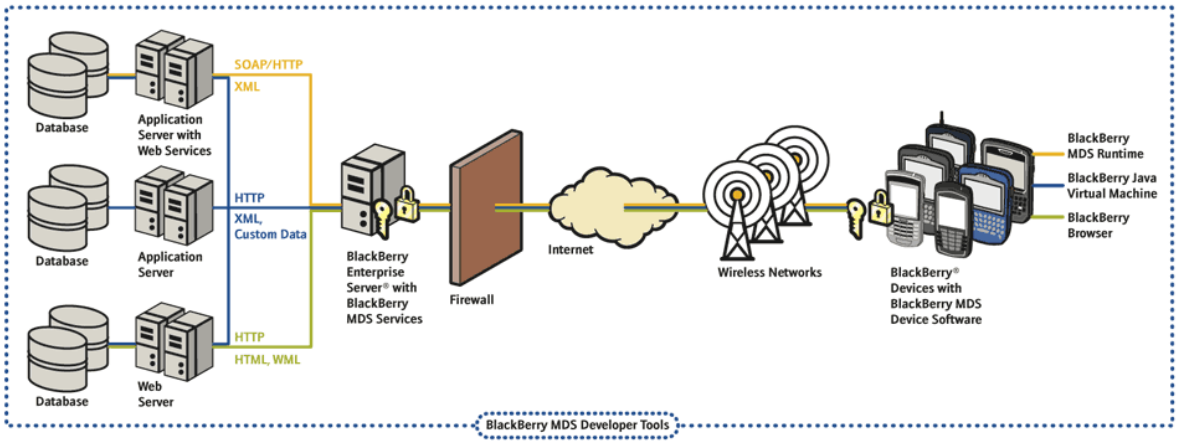


Abbildung 2.4: BlackBerry Infrastruktur

Sowohl das Betriebssystem als auch das Push-Protokoll sind proprietär. Allerdings implementieren alle BlackBerrys die Java-Laufzeitumgebung für mobile Geräte von Sun Microsystems (Sun), erweitert durch RIM-eigene Bibliotheken.

3 Anforderungsanalyse

Dieses Kapitel behandelt die Aufgabenstellung der vorliegenden Arbeit: Die Entwicklung einer Kommunikationsplattform, die für künftige mobile Anwendungen genutzt werden soll. Hierzu werden zuerst die Probleme genannt, die zu der Entwicklung einer solchen Lösung motivieren. Danach werden die Anforderungen spezifiziert. Die Anforderungen werden später mit den geschaffenen Funktionalitäten verglichen, um den Erfolg des Projektes abschließend bewerten zu können.

3.1 Analyse der Probleme

Einige Gründe und Vorteile zum Einsatz mobiler Anwendungen wurden in Kapitel 2.1: *Mobile Anwendungen* bereits genannt. Der Einsatz mobiler Anwendungen bringt einige Problem mit sich, die bei stationären Anwendungen nicht vorhanden sind:

- **Verbindungsabriss**
  Der plötzlich Verlust der Verbindung zum Netz des Mobilfunkbetreibers. Bei stationären Anwendungen ist der Verlust der Verbindung eher selten und dann meist das Resultat eines technischen Fehlers. Bei mobilen Anwendungen dagegen tritt dies ständig auf. Ursachen dafür können zum Beispiel ein Funkloch durch einen Tunnel oder ein massives Gebäude sein. Für den Anwender wäre es komfortabel wenn er sich um Verbindungsprobleme nicht kümmern müsste, sondern sich darauf verlassen kann, dass die Anwendung dies für ihn erledigt. Nachrichten werden dann einfach gesendet, wenn das Gerät wieder im Bereich eines Mobilfunkanbieters ist. Wird dies dem Anwender überlassen, entstehen potenzielle Fehlerquellen, zum Beispiel, wenn der Nutzer davon ausgeht, dass die Sendung erfolgreich war, weil er nicht merkt, dass keine Verbindung besteht. Oder wenn er fälschlicherweise

einen Fehler annimmt und mehrmals dieselbe Nachricht oder denselben Auftrag versendet. Was für Privatanwender, zum Beispiel beim Verlust oder dem doppelten Senden einer SMS, verschmerzbar ist, kann im Geschäftsbereich schwerwiegende Folgen haben.

- **Pushen von Nachrichten zu mobilen Geräten**
  Ebenfalls ein Problem bei mobilen Anwendungen ist das Übertragen von Daten von einem stationären zu einem mobilen Gerät. Während die Kommunikation, die vom mobilen Gerät aus angestoßen wird, von den Mobilfunkanbietern unterstützt wird, ist dies in die andere Richtung nicht so einfach möglich. Im Moment bietet nur RIM mit ihrer BlackBerry-Plattform in Kooperation mit den Mobilfunkanbietern diese Möglichkeit. Andere Unternehmen simulieren lediglich einen solchen Daten-Push. Microsoft zum Beispiel sendet ständig einen Http-Request an den Server und hält somit die Verbindung offen. Das Senden von Nachrichten an ein mobiles Gerät zu ermöglichen ist also, wie die Verbindungsprobleme auch, aufwändig zu handhaben.

Diese Probleme sind allgemeiner Natur und beziehen sich auf alle Arten von Anwendungen im mobilen Bereich. Außerdem sind mobile Anwendungen eine spezielle Form verteilter Systeme, so dass Problematiken aus diesem Bereich auch hier beachtet werden müssen (siehe Unterkapitel 2.2: *Verteilte Systeme*). Unternehmen, die Anwendungen für mobile Geräte entwickeln, müssen die oben genannten Punkte beachten. Dies macht die Entwicklung, unabhängig von der eigentlich zu realisierenden Logik des Programms, aufwändig und damit teuer. Durch Einsparungen in diesem Bereich leidet häufig die Qualität der Anwendung. Um daher diese Problematiken nicht für jede Anwendung erneut zu lösen, ist es sinnvoll diese Funktionalitäten zu kapseln und so zu entwickeln, dass sie von möglichst vielen Anwendungen genutzt werden können.

Diese Erkenntnis hatte auch die Dymacon Business Solutions GmbH. Um schneller und günstiger Anwendungen entwickeln zu können, um sich so einen Vorsprung vor Mitbewerbern zu sichern, sollte eine solche Kommunikationsplattform für die BlackBerry Umgebung geschaffen werden. Die konkreten Anforderungen dafür werden in nächsten Abschnitt genauer definiert.

3.2 Anforderungen

Die Anforderungen der Dymacon Business Solutions GmbH, sind nur Konzepte und nicht sehr detailliert:

- **Zugriff auf Daten** Das schließt Daten aus dem Unternehmen, aber auch aus dem Internet mit ein, zum Beispiel der Zugriff auf Datenbanken, das Abfragen von Webservices und das Herunterladen von Dateien. E-Commerce ist hier das Stichwort, also Ankauf, Verkauf und Handel durch den Zugriff auf Dienstleistungen im Internet wie Auktionshäuser oder Versandhäuser[1]. Darüber hinaus sollen die Anwendungen auf dem BlackBerry in Geschäftsprozesse des Unternehmens eingebunden werden können.
- **Nachrichten zum BlackBerry schicken** Das System soll die Möglichkeit bieten, Nachrichten an die Geräte zu schicken. Anstatt immer einen Synchronisationsprozess vom Gerät aus zu starten sollen Änderungen sofort an die Geräte gepusht werden können.
- **Offlinefähigkeit** Der Anwender soll Nachrichten verschicken können, unabhängig davon, ob er im Empfangsbereich eines Mobilfunkanbieters ist. Die Anwendung übernimmt dann das Senden von Nachrichten, wenn dies wieder möglich ist. Für die Antworten beziehungsweise Nachrichten, die an die Geräte gesendet werden, gilt das Gleiche. Diese Anforderungen sind nur durch **Asynchronität** zu erreichen.
- **Modularer Aufbau** Das System soll modular aufgebaut sein. Damit ist einerseits die Austauschbarkeit von einzelnen Bereichen des Systems, zum Beispiel nach einem Update, gemeint. Andererseits das Hinzufügen von neuen Modulen, um die Funktionalität zu erweitern, zum Beispiel, wie in Punkt „Zugriff auf Daten“ aufgeführt, die Integration in spezielle Geschäftsprozesse.
- **Sichere Datenübertragung** Die Kommunikation zwischen dem Gerät und einem Service soll verschlüsselt sein. Zusätzlich soll eine sichere Anbindung an das Unternehmensnetzwerk realisiert werden.

[1] Kollmann (2007)

- **OTA Konfigurierbarkeit** Mögliche Anpassungen an der Plattform sollen Over-The-Air (OTA) möglich sein. Zum Beispiel sind Vertreter mit BlackBerry Endgeräten häufig unterwegs und sollen nicht jedes mal in das Unternehmen kommen müssen, weil an der Kommunikationsplattform etwas umgestellt werden muss. Den Anwender selbst Einstellungen vornehmen zu lassen ist keine Alternative. Dies wäre für den Anwender erstens nicht komfortabel und würde zweitens eine Fehler- und Risikoquelle sein. Die Konfiguration soll also vom Unternehmen aus möglich sein.
- **Geringes Datenvolumen** Um die Kosten gering zu halten, müssen die Übertragungsdauer und das Datenvolumen möglichst klein sein.

Wie diese Anforderungen technisch umgesetzt werden sollten war unklar. Deswegen wird, im folgenden Kapitel, der aktuelle Forschungsstand beleuchtet und eine Machbarkeitsstudie durchgeführt. So sollen die Möglichkeiten analysiert werden, die zur Erfüllung der Anforderungen existieren. In Abbildung 3.1 sind die Anforderungen nochmals grafisch dargestellt.

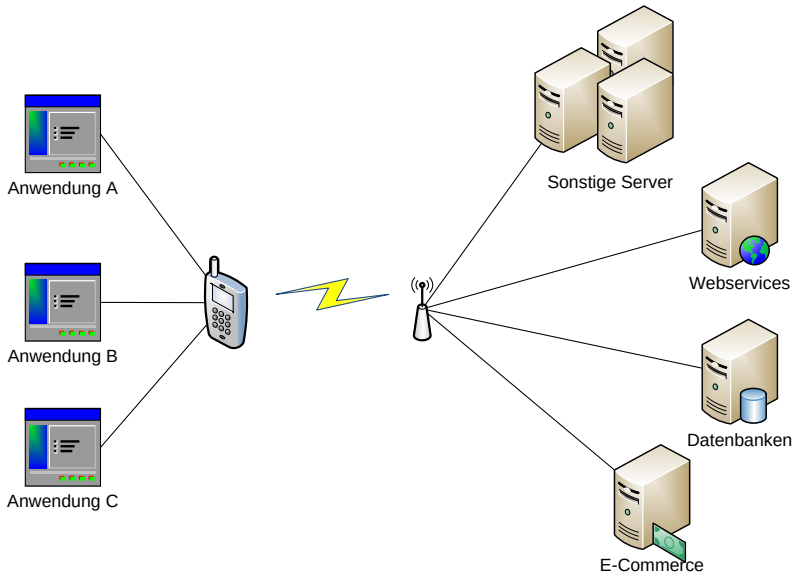


Abbildung 3.1: Anforderungen und Verwendungszweck des Systems

4 Stand der Forschung und Machbarkeitsstudie

Dieses Kapitel betrachtet die Machbarkeit der Anforderungen anhand des aktuellen Forschungsstands. Zuerst wird der strukturelle Aufbau einer Lösung betrachtet, dann wird auf die einzelnen Kommunikationspartner, also auf das Mobilfunkgerät und den Server, eingegangen. Anhand dieser Machbarkeitsstudie wird schließlich eine Wahl für die Implementierung getroffen.

4.1 Struktureller Aufbau

Aus dem Kapitel 3.2: *Anforderungsanalyse* sind sieben Anforderungen zu erfüllen:

- Zugriff auf Daten
- Nachrichten zum BlackBerry schicken
- Offlinefähigkeit durch Asynchronität
- Modularer Aufbau
- Sichere Datenübertragung
- OTA Konfigurierbarkeit
- Geringes Datenvolumen

Für die BlackBerry-Geräte gibt es eine am Markt befindliche Lösung, die ein ähnliches Spektrum an Funktionen unterstützt wie in den Anforderungen gefordert:

- Flowfinity. Eine Lösung des Unternehmens Flowfinity Wireless Inc. Flowfinity ist eine Middleware, die es ermöglicht, mobile Geräte an bestehende ERP[1] oder CRM[2] Systeme anzubinden. Allerdings fallen hier, zusätzlich zur Anpassung an die Unternehmensstruktur, Lizenzkosten an. Dies ist insbesondere dann ärgerlich, wenn nicht die volle Funktionalität benötigt wird. Flowfinity bietet als günstigere Alternativen auch Out-of-the-Box Lösungen an. Allerdings passen diese meist nicht auf das spezielle Anwendungs-Szenario des Kunden. Darüber hinaus laufen Anwendungen auf dem BlackBerry in einer extra Umgebung.

Diese Lösung ist für die gestellten Anforderungen nicht geeignet: Clientseitig soll der Zugriff auf Daten im Internet oder Intranet ermöglicht werden. Es muss die Offlinefähigkeit bereitgestellt werden und das modulare Anbinden von weiteren Anwendungen muss möglich sein. Außerdem soll eine Konfiguration mittels OTA erreichbar sein.

Serverseitig ist es nötig, den Zugriff auf Daten zu ermöglichen. Es müssen Daten vom Mobilfunkgerät entgegen genommen und an dieses geschickt werden können. Auch muss die Implementierung modular um zusätzliche Funktionen erweiterbar sein.

Alle Anforderungen sollen durch eine einzige Lösung realisiert werden. **Die Implementierung einer solchen Gesamtlösung bezeichnet man als Middleware.** Die Middleware kapselt die Funktionen zu den Anforderungen (siehe Abbildung 4.1). Diese Funktionalitäten werden mit Schnittstellen nach außen versehen und so anderen Entwicklern zur Verfügung gestellt.

Da sich die Hardwareplattformen und Anforderungen zwischen mobilen Geräten auf der einen und Servern auf der anderen Seite unterscheiden, werden nun beide Plattformen getrennt betrachtet.

4.2 Client

Für die Umsetzung der Anforderungen steht clientseitig nur eine Möglichkeit zur Verfügung: Die Entwicklung eines Services, der anderen Anwendungen die Funktionen der

[1] Enterprise Resource Planing
[2] Customer Realationship Management

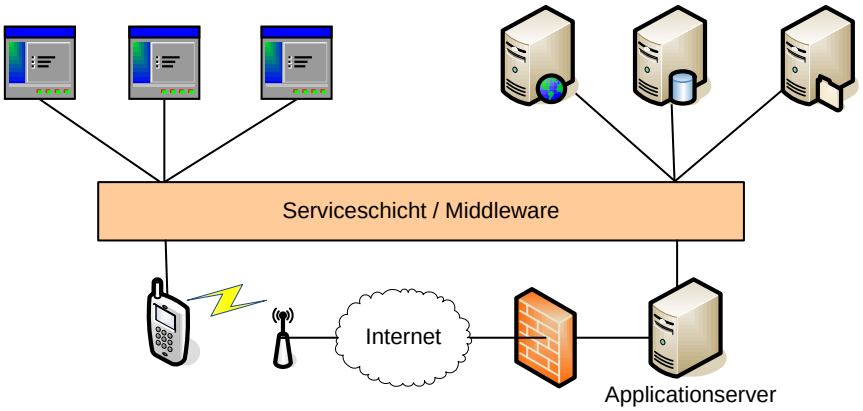


Abbildung 4.1: Middleware

Middleware zur Verfügung stellt. Der Service wird als eigene Applikation implementiert. Dies hat mehrere Vorteile:

- Ein einzelner Service, den sich mehrere Anwendungen teilen, beansprucht weniger Ressourcen, als eine Implementierung für jede Anwendung einzeln.
- Es entstehen keine Ressourcenkonflikte und Deadlocks[3] durch mehrere Implementierungen.
- Ein einzelner Service kann leichter gewartet werden.
- Eine einfache und einheitliche Konfiguration für alle Anwendungen kann gewährleistet werden.

Der Service ermöglicht zusätzlich, dass auch dann die Aufgaben der Middleware wahrgenommen werden können, wenn keine der „Nutzer-Anwendungen“ aktiv ist. Die einzelnen Anwendungen registrieren sich bei der Middleware über bestimmte Schnittstellen und nutzen so deren Funktionen.

Mit den BlackBerry-Mobilfunkgeräten ist clientseitig eine Plattform vorgegeben. Um die Middleware als Service auf diesem Gerät zu implementieren, steht nur eine Programmiersprache zur Auswahl: Java. Wie in Kapitel 2.4: *BlackBerry* beschrieben, unterstützt der BlackBerry die Java ME, erweitert mit RIM-eigenen APIs.

[3]Prozesse warten gegenseitig auf die Freigabe von Ressourcen

4.2.1 Java ME

Die Java Micro Edition (Java ME) ist eine spezielle Variante der Java-Programmierspra-che. Sie unterstützt nur einen Teil dieser Sprache, erweitert diese aber um spezielle Funktionen für mobile Endgeräte (Abbildung 4.2 verdeutlicht dies). Java ME unterscheidet zwischen Konfigurationen und Profilen. Eine Konfiguration definiert einen minimalen Funktionsumfang für eine bestimmte Klasse an Geräten. Sie wird als Connected Limited Device Configuration (CLDC) bezeichnet und beinhaltet auch die KVM[4]. Ein Profil erweitert eine Konfiguration um spezielle Leistungsmerkmale für einen bestimmten Typ von Geräten. Sie wird als Mobile Information Device Profile (MIDP) bezeichnet. Anwendungen, die für MIDP-Geräte erstellt wurden, werden als MIDlets bezeichnet.

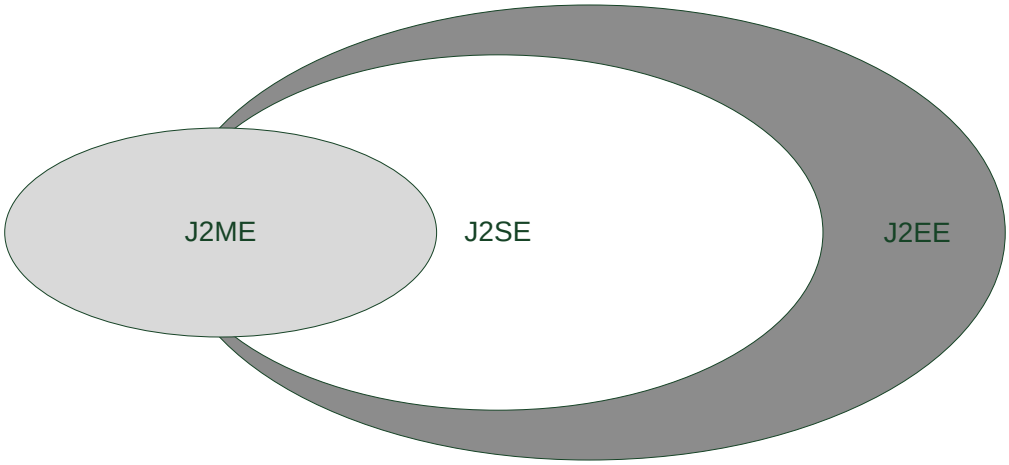


Abbildung 4.2: Unterschiede zwischen Java ME, Java SE und Java EE (siehe Schmatz (2004))

Diese MIDlets laufen innerhalb von MIDlet-Containern und werden von der Application Management Software (AMS) aufgerufen. Dazu besitzen sie keine *main()*-Methode, sondern erweitern eine Klasse *MIDlet* und implementieren abstrakte Methoden, um die Anwendung starten, pausieren und beenden zu können.[5]

[4] Kilobyte Virtual Machine, eine Laufzeitumgebung für Java-Programme
[5] SUN (2007)

4.2.2 BlackBerry API

Das Unternehmen Research in Motion setzt bei seinen BlackBerry-Geräten auf Java. Entwicklern stellt es mehrere Möglichkeiten zur Verfügung, um Java-Anwendungen für die BlackBerrys entwickeln zu können. Zum einen ist eine vollständige KVM-Umgebung in das Betriebssystem des Mobilfunkgerätes integriert. Entwickler erhalten die Möglichkeit, eigene MIDlets zu erstellen und den vollständigen Funktionsumfang der Java ME zu nutzen. Darüber hinaus stellt RIM eigene Bibliotheken zur Verfügung, um spezielle Funktionen des BlackBerry nutzen zu können. Eine besondere Eigenheit in diesem Bereich ist folgendes: Der Entwickler hat die Möglichkeit, anstatt eines MIDlets, eine eigenständige Anwendung zu erstellen, die unabhängig von einem MIDlet-Container ist: Eine **UiApplication**. Die Nutzung dieses Typs ermöglicht den Zugriff auf BlackBerry-spezifische Funktionen wie die Kontrolle des Navigationsrads. Im Gegensatz zu MIDlets besitzen solche Anwendungen eine *main()*-Methode.

Neben diesen Erweiterungen der Java ME stellt RIM auch zwei Entwicklungsumgebungen für seine BlackBerrys zur Verfügung, die registrierte Entwickler kostenlos nutzen können. Zu diesen IDEs gehören auch ein Debugger und ein Simulator. Außerdem ein Server, um Nachrichten mit der BlackBerry-Infrastruktur austauschen zu können.

Alle Anwendungen, unabhängig ob sie als MIDlet oder als UIApplikation erstellt wurden, können mittels OTA (Over-The-Air) installiert werden. Es ist also möglich, diese Anwendungen in einem Browser auf dem BlackBerry-Gerät auszuwählen und so den Installationsvorgang der Anwendung anzustoßen. Ein direktes Installieren über ein Kabel entfällt somit, dem Anwender ist eine Installation einer Anwendung von jedem Ort, zu jeder Zeit möglich.

4.3 Server

Um den Server zu realisieren, wird auf ein Komponentensystem zurückgegriffen. Die Vorteile eines solchen Systems werden im nächsten Kapitel beschrieben. Zusätzlich werden verschiedene Komponentensysteme vorgestellt und eine Auswahl für eines der Modelle getroffen.

4.3.1 Komponentensysteme

Definition einer verteilten Komponente in der UML (Unified Modelling Language): Eine verteilte Komponente stellt eine modulare, verteilbare und ersetzbare Einheit eines Systems dar, welche ihren Inhalt kapselt und eine oder mehrere Schnittstellen nach außen zur Verfügung stellt.

Konponentensysteme stellen diesen Komponenten Ablaufumgebungen bereit. Diese Ablaufumgebungen, so genannte *Container*, nehmen dem Entwickler und dem Integrator Arbeit ab. Hierzu gehören laut Mandl (2005):

- Der Programmierer muss sich nicht mehr um die Erzeugung von Komponenteninstanzen kümmern. Die Verwaltung des Lebenszyklus von Komponenten wird vom Komponentensystem übernommen.
- Komponentensysteme ermöglichen den nebenläufigen Ablauf mehrerer Instanzen einer Komponente ohne zusätzliche Programmierung. Der Programmierer einer Komponente nutzt ein einfaches, sequenzielles Programmiermodell. Die Parallelisierung übernimmt der Container. Komponenten werden vom Container bei Bedarf erzeugt und freigegeben.
- Komponentensysteme stellen eine Reihe von Systemdiensten bereit, die vom Komponentenentwickler verwendet werden können bzw. implizit verwendet werden. Hierzu gehören Tranksaktionsdienste, Persistenzdienste, Behandlung von Datenbankverbindungen und Threads.
- Komponentensysteme stellen Basismechanismen für die Einbettung von Komponenten dar. Über Regeln wird festgehalten, welche Schnittstellen eine Komponente bereitstellen muss, damit sie im Container ablauffähig ist.
- Mechanismen für die Skalierbarkeit und auch für Fehlertoleranz und Lastverteilung werden durch den Container unterstützt. Komponenten können mehrfach gestartet werden.

Server mit derartigen Ablaufumgebungen werden *Application-Server* genannt.

Das Common Object Request Broker Architecture (CORBA) Component Model (CCM) der OMG, SUNs Java Platform Enterprise Edition (Java EE) / Enterprise JavaBeans

(EJB) und Microsofts .NET Enterprise Services sind solche Komponentensysteme. Diese werden im folgenden Teil vorgestellt.

CORBA Components Model (CCM) CCM ist durch die Object Management Group (OMG) spezifiziert. Die Basis von CCM ist CORBA. CCM ist eine Erweiterung, die CORBA um komponentenbasierte Entwicklung erweitert. Clients greifen über CORBA auf diese Komponenten zu, um deren Dienste zu benutzen. CORBA Implementierungen sind programmiersprachenunabhängig und existieren beispielsweise von Orbix, MICO und IBM.

Java Platform, Enterprise Edition (Java EE) Java EE ist eine Spezifikation für verteilte serverseitige Applikationen in der Java-Welt. Die Spezifikationen werden im Java Community Process (JCP) verabschiedet. Die Java EE-Schnittstellenspezifikationen sind plattform- und implementierungsabhängig. Hierzu existieren einige Implementierungen, wie zum Beispiel WebSphere (IBM), WebLogic (BEA) und JBOSS (JBOSS).

.NET Enterprise Services Die .NET Enterprise Services von Microsoft sind der Komponentenstandard der Windows-Welt. Sie sind der Nachfolger von COM+ und setzten auch dort auf. Die Komponenten laufen hier ebenfalls in Containern ab, allerdings werden diese hier als Kontext bezeichnet. Anders als die anderen beiden Modelle, sind die .NET Enterprise Services keine bloße Spezifikation, sondern eine konkrete Implementation des Herstellers Microsoft.

Begründung der Auswahl Für die Implementierung der Anforderungen wurde Java EE gewählt. Die Entscheidung hierfür hat verschiedene Gründe:

- Es sind zahlreiche Implementierungen für Java-EE-Server verfügbar, darunter viele *Open-Source Implementierungen* wie zum Beispiel JBoss oder Apache Geronimo. Zusätzlich wird eine Referenzimplementierung von Sun Microsystems zur Verfügung gestellt.
- Für Java existieren viele **kostenlose Entwicklungsumgebungen**und Debugger, darunter umfangreiche Werkzeuge wie Eclipse und Netbeans.

- Java ist **plattformunabhängig**. Die Java EE Implementierungen laufen auf allen Betriebssystemen, für die eine Java Virtual Machine (JVM) zur Verfügung steht. Motto: „Program once, run anywhere“.
- Da Java EE nur eine **Spezifikation** ist, sind Anwendungen nicht auf Implementierung eines Herstellers beschränkt. Man hat so die Flexibilität, zwischen den verschiedenen Implementierungen der Herstellern zu wählen, um seine Bedürfnisse möglichst gut abzudecken.
- Java eignet sich sehr gut für **Prozess- und Systemintegration**. Beispielsweise ist die Anbindung von Legacy-Systemen[6] durch die Java Connector Architecture möglich.
- Java ist weit **verbreitet** und wird von vielen Programmierern beherrscht. Dies erleichtert die Weiterentwicklung und die Wartbarkeit.
- Eine **einheitliche Programmiersprache** auf Client- und Serverseite erleichtert das Arbeiten.

4.3.2 Java EE

Die grundlegende Struktur eines Java EE Application Servers ist in Abbildung 4.3 dargestellt. Sie zeigt, dass der Application-Server zwei Container-Typen unterstützt: Web-Container und EJB-Container. Web-Container verwalten Web-Komponenten wie Servlets und Java Server Pages (JSP). EJB-Container verwalten Enterprise JavaBeans (EJBs). Beide Container stellen Laufzeitumgebungen für die jeweiligen Komponenten dar. Servlets und EJBs sind also verteilbare Komponenten, die auf der Serverseite installiert werden.

Zusätzlich stellt der Server einige technische Funktionalitäten wie Sicherheit, Kommunikation zwischen den Komponenten und Namens- und Verzeichnisdienste bereit. Des weiteren kapselt der Server den Zugriff auf die Ressourcen des zugrunde liegenden Betriebssystems wie Dateisystem und Netzwerk. Java EE regelt außerdem das Zusammenspiel von verschiedenen Java-Standards. Einige dieser APIs werden hier erläutert, um die Möglichkeiten von Java EE zu zeigen.

[6] Altsysteme, historisch gewachsene Anwendungen

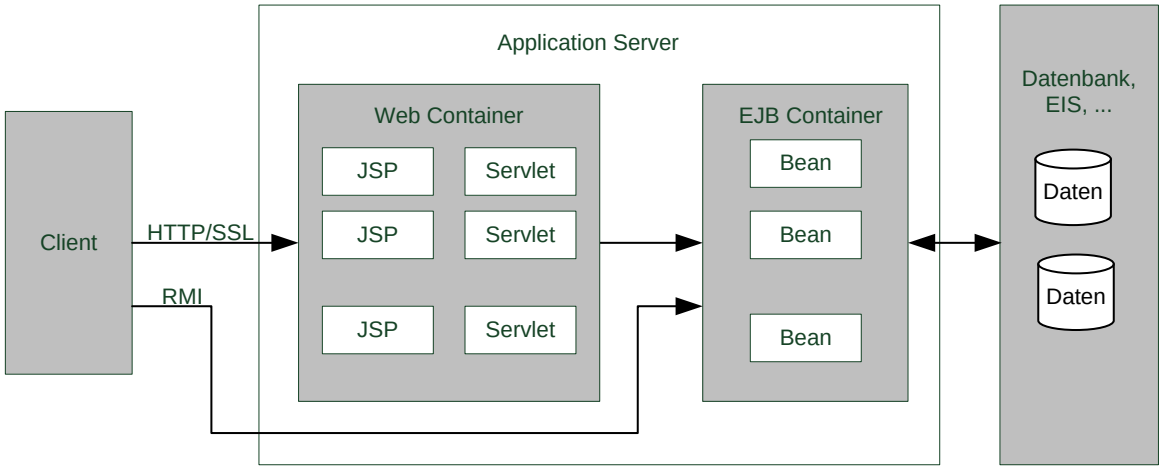


Abbildung 4.3: Java EE Architektur

- *JMS (Java MessagingService)*: Dies ist eine API für den Zugriff auf ein Nachrichtenwarteschlangensystem. Sie ermöglicht asynchrone Kommunikation zwischen verteilten Anwendungen.
- *JTA (Java Transaction API)*: Ermöglicht die Teilnahme von Komponenten an Transaktionen.
- *JDBC (Java Database Connectivity)*: Ermöglicht den Zugriff auf relationale Datenbanken über einen einheitlichen Mechanismus.
- *JCA (Java Connector Architecture)*: Soll einen einheitlichen Mechanismus für den Zugriff auf verschiedene Informationssysteme (EIS) zur Verfügung stellen.
- *JNDI (Java Naming and Directory Service)*: Zugriff auf einen Verzeichnisdienst (Naming Service).
- *JavaMail*: Schnittstellen für den Zugriff auf ein Mail System zum Absenden und Empfangen von E-Mails.

Java EE eignet ich somit nicht nur um einen einfachen Server zu erstellen, sondern bietet Werkzeuge, um komplette Prozesse und Systeme zu integrieren.

Im weiteren Verlauf werden Technologien der Java EE genauer beschrieben, mit denen der Serverteil realisiert werden kann. Darüber hinaus wird kurz auf Webservices

eingegangen, da sie zusätzliche Möglichkeiten für die Nutzung von Java EE ermöglichen.

Servlets Servlets sind speziell für Webanwendungen entwickelt. Der zugehörige Servlet-Container stellt eine Umgebung für die parallele Abarbeitung von Anfragen zur Verfügung (siehe Unterkapitel 4.3.1). Es existieren spezielle HTTP-Servlets. Diese werden über HTTP-Requests angesprochen, welche vom Application-Server an den Servlet-Container weiter gereicht werden. Sie können HTTP-Get- und -Post-Anfragen bearbeiten. Die Entsprechenden Methoden dafür werden vom Programmierer mit beliebigem Java Code überschrieben. Das Servlet gibt dann den ebenfalls vom Programmierer festgelegten HTTP-Response zurück.

EJB Enterprise Java Beans sind im Java-Umfeld die Standard-Komponentenarchitektur. Sie laufen ebenfalls in speziellen Containern ab. Für die Implementierungen eines EJBs steht der vollständige Java Code bereit. EJBs besitzen im Gegensatz zu Servlets keine GUI-Funktionalität und werden auch nicht über HTTP angesprochen. Sie sind objektorientierte Komponenten, die über Remote Method Invocation(RMI) oder Internet Inter ORB Protocol (IIOP, Corba) angesprochen werden können. Spezielle EJBs können sich im Gegensatz zu Servlets Zustände merken, um mehrere Methodenaufrufe zusammenhängend zu verwenden.

Webservices Webservices bieten durch die konsequente Nutzung von XML plattformübergreifenden und sprachunabhängigen Zugriff auf Funktionen über das Intra- oder Internet. Webservices werden über ein standardisiertes Protokoll (SOAP) angesprochen. Die Definition eines Webservices wird mittels WSDL (Web Service Description Language) beschrieben. SOAP und WSDL basieren beide auf XML. Zusätzlich existiert ein Verzeichnisdienst für das Auffinden von Webservices (UDDI). Webservices ermöglichen, als zusätzliche Schicht zwischen Client und einen Java EE Application-Server, zusätzliche Plattform- und Programmiersprachenunabhängigkeit. SOAP kann über HTTP oder SMTP übertragen werden.[7][8]

[7] Monson-Haefel (2004)
[8] Mandl (2005)

4.3.3 Möglichkeiten

Nach der Beschreibung der verschiedenen technischen Möglichkeiten, die Java EE bietet, soll hier anhand dieser Möglichkeiten eine geeignete Lösung für die Umsetzung gefunden werden. Dabei müssen zwei Punkte beachtet werden:

- Wie in Abschnitt 2.4: BlackBerry beschrieben, ist die Datenübertragung vom BlackBerry zum BlackBerry Enterprise Server (BES) über HTTP standardmäßig verschlüsselt. Es bietet sich also an, die Daten über HTTP zu übertragen.
- RIM bietet mit BlackBerry Connect seine Push-Technologie auch für Geräte anderer Hersteller an. Zusätzlich arbeitet RIM im Moment an einem BlackBerry Emulator für Windows Mobile. Damit diese Geräte später die Kommunikations-Plattform nutzten können, sollte die Serverschnittstelle sprachunabhängig sein.

Ohne diese spezielle Anforderungen wäre eine Lösung mit EJBs am geeignetsten. Ein objektorientierter Zugriff auf serverseitige Komponenten, welche bei Bedarf noch zustandsbehaftet sind, würde einen komfortablen Weg bieten, die Kommunikation zwischen Client und Server zu lösen - zumal RMI optional über HTTP übertragen werden kann und so die standardmäßige HTTP-Verschlüsselung des BlackBerrys nutzen könnte. RMI ist allerdings auf Java beschränkt und so nicht sprachenunabhängig. Um dieses Problem zu umgehen, könnte zusätzlich ein Webservice zwischen Client und Server gelegt werden (siehe Abbildung 4.4). Die Clients würden dann den Webservice über SOAP ansprechen und dieser über RMI die Java Beans. Da SOAP über HTTP übertragen werden kann, würde auch hier die BlackBerry Verschlüsselung greifen. Allerdings geht hierbei der objektorientierte Ansatz verloren.

Ein Vorteil hierbei ist die Nutzung von ausschließlich existierenden Protokollen. Es muss kein eigenes Protokoll entwickelt werden. Für Java ME existieren OpenSource SOAP Bibliotheken, mit denen Webservices anhand einer *High-Level-API* angesprochen werden können (zum Beispiel kSOAP). Auch für die Kommunikation mittels RMI muss nicht auf *Low-Level-APIs* zurück gegriffen werden. Somit werden potenzielle Fehlerquellen ausgeschlossen. Nachteile sind die zusätzliche Instanz zwischen Client und Application-Server. Dies kostet zusätzlichen Entwicklungsaufwand und verschlechtert durch das „Konvertieren“ von SOAP in RMI die Performance. Zusätzlich ist eine SOAP Nachricht recht umfangreich und bringt bei kleinen Nachrichtenpaketen einen beachtlichen Overhead an

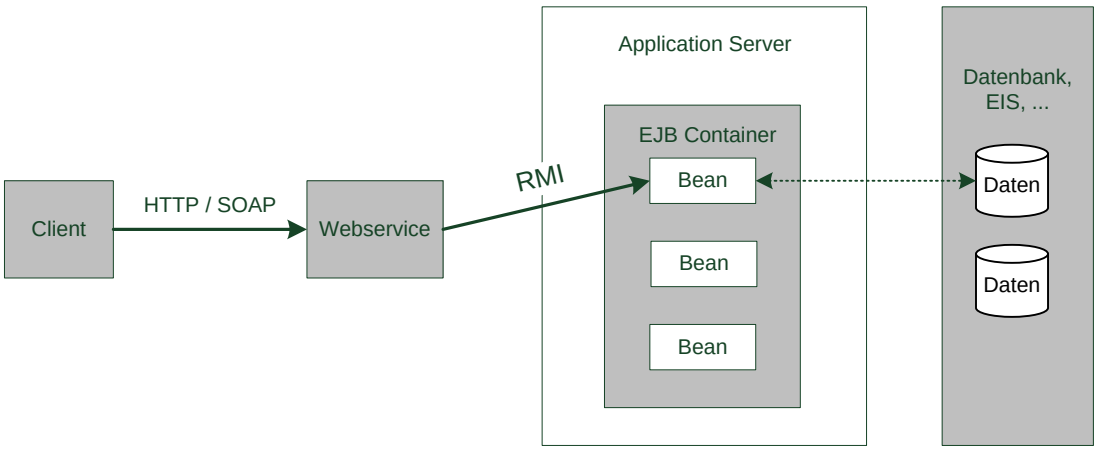


Abbildung 4.4: Architektur mit Webservices und EJBs

Daten mit sich. Dies spielt gerade bei mobilen Clients eine große Rolle (Siehe Abschnitt 2.1: *Mobile Anwendungen*).

Die Alternative hierzu ist eine direkte Nutzung von Servlets (siehe Abbildung 4.5). Servlets werden über HTTP angesprochen, nutzen also die BlackBerry Verschlüsselung. Allerdings kann hier nicht, wie bei der Nutzung vom Webservices, auf ein bestehendes Protokoll zurück gegriffen werden. Es würde also noch die Entwicklung eines eigenen Protokolls anfallen. Dieses Protokoll müsste wie SOAP, sprach- und plattformunabhängig sein.

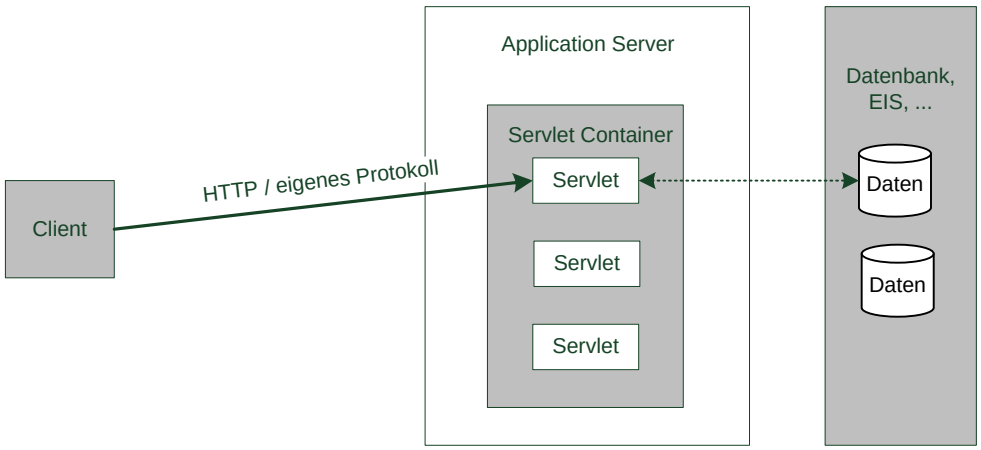


Abbildung 4.5: Architektur mit Servlets

Der Vorteil bei der Nutzung von Servlets ist, dass die zusätzliche Instanz durch den Webservice weg fällt. Dieser Implementierungsaufwand entfällt also. Allerdings muss ein eigenes Protokoll entwickelt werden. Dies schafft wiederum zusätzlichen Aufwand und bringt eine gewisse Fehleranfälligkeit mit sich. Allerdings kann das Protokoll so gestaltet werden, dass es die Anforderungen möglichst genau erfüllt und gleichzeitig einen geringen Overhead an Daten erzeugt.

4.4 Ausgewählte Variante

Nachdem die verschiedene Möglichkeiten gezeigt worden sind, um die Anforderungen zu erfüllen, wird hier die Begründung für die letztendlich gewählte Variante geliefert.

Auf der Clientseite existieren, wie in Abschnitt 4.2 beschrieben, keine alternativen Lösungsmöglichkeiten zur Erfüllung der Anforderungen. Es muss ein eigenständiger Service implementiert werden, der den anderen Anwendungen seine Funktionalitäten zur Verfügung stellt, aber seine Aufgaben auch dann erledigen kann, wenn keine der Anwendungen aktiv ist.

Auf der Serverseite stehen zwei Varianten zur Auswahl. Zum einen die Nutzung von Webservices und EJBs, zum anderen Servlets. Die Entscheidung fiel auf die Nutzung der Servlet-Architektur. Der Nachteil, ein eigenes Protokoll entwickeln zu müssen, wird gleichzeitig durch die dadurch entstehenden Vorteile, zum Beispiel geringen Overhead, aufgehoben. Zusätzlich fällt die Webservice-Instanz weg. Diese hätte zusätzlich zu Client und Server implementiert und gewartet werden müssen. Auch die Performance ist zu beachten. BlackBerry Lösungen in großen Unternehmen umfassen mehrere hundert Geräte, welche im Extremfall alle gleichzeitig Aktionen über den Server ausführen wollen. Da die Kommunikation über einen zentralen Server laufen soll, muss dieser entsprechend performant sein.

Die Gesamtarchitektur stellt sich wie in Abbildung 4.6 gezeigt dar. Der Service dient den Anwendungen auf der Clientseite als eine Art Multiplexer. Die Nachrichten der verschiedenen Anwendungen werden entgegengenommen und einheitlich übertragen. Die Serverseite dient als eine Art Demultiplexer, sie trennt die Nachrichten wieder entsprechend ihres Inhaltes auf.

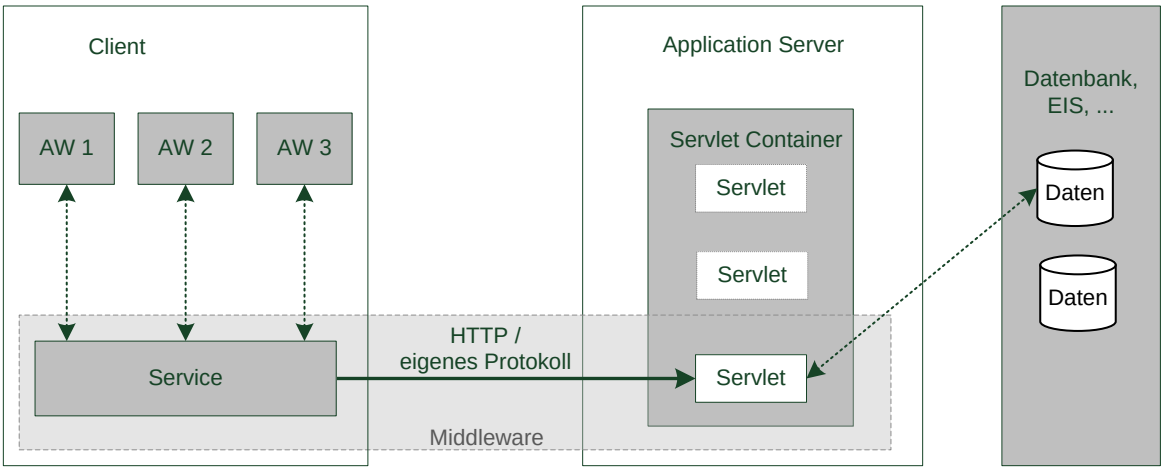


Abbildung 4.6: Gesamtarchitektur der Middleware

Auf den Entwurf und das Design der einzelnen Komponenten wird im nächsten Kapitel eingegangen. Im Kapitel darauf wird dann die konkrete Realisierung gezeigt.

5 Entwurf und Design

Aus den in der Machbarkeitsstudie getroffenen Entscheidungen wird in diesem Kapitel eine Middleware zur Verwirklichung der Anforderungen entworfen. Entwurfsentscheidungen werden erläutert und anhand von grafischen Modellen dargestellt. Die Diagramme und Codebeispiele in diesem Kapitel sind nicht immer vollständig, sondern zeigen Basisfunktionalitäten und elementare Stellen, die zum Gesamtverständnis der Middleware beitragen.

5.1 Aufteilung in Schichten

Um die Komplexität der Middleware zu reduzieren, wird diese in Schichten unterteilt. Dieses Vorgehen bietet den Vorteil, dass beide Schichten unabhängig voneinander entwickelt werden können. Es müssen lediglich Schnittstellen definiert werden, damit eine Schicht auf die Funktionalitäten der anderen Schicht zugreifen kann. Die Schichten auf gleicher Ebene kommunizieren jeweils über ein eigenes Protokoll miteinander. Dies kann ebenfalls unabhängig von einer anderen Schicht entwickelt werden. Näheres zum Schichtenmodell siehe Kapitel 2.3:*Schichtenmodell*.

5.1.1 Definition der Schichten

Die Middleware wurde zur Entwicklung in folgende zwei Schichten aufgeteilt (siehe Abbildung 5.1):

- **Schicht 2: Schnittstellen für die Anwendungen**
  Diese Schicht ist für die Logik innerhalb der Middleware verantwortlich. Dies umfasst die Handhabung von mehreren Anwendungen auf einem Gerät, das Einbinden

von Modulen und deren Ausführbarkeit und die Over-The-Air-Konfiguration. Sie stellt den Anwendungen, sowohl auf Client als auch auf Serverseite Schnittstellen bereit, um Nachrichten zu verschicken und zu empfangen. Sie wird auch als *Serviceschicht* bezeichnet.

- **Schicht 1: Kommunikation innerhalb der Middleware** Diese Schicht kann als *Kommunikationsschicht* verstanden werden. Ihre Aufgabe besteht darin, die Nachrichten zuverlässig zwischen Client und Server zu übertragen.

Die weiteren Kapitel erläutern den Aufbau dieser Schichten genauer, wobei die Anforderungen in folgenden Schichten gelöst werden:

Schicht 2 (Serviceschicht)

- **Zugriff auf Daten**. Diese Schicht bereitet die Nachrichten für die Anwendungen auf oder kapselt sie für die Übertragung zum Server.
- **Modularer Aufbau**. Anwendungen können sich bei der Middleware anmelden und ihre Funktionen nutzen.
- **OTA Konfigurierbarkeit**. Die Middleware kann vom Server aus konfiguriert werden.

Schicht 1 (Kommunikationsschicht)

- **Nachrichten zum BlackBerry schicken.** Die Implementierung der Push-Funktion von RIM ermöglicht die Übermittlung von Nachrichten an den BlackBerry.
- **Offlinefähigkeit**. Dies wird erreicht durch einen **asynchronen** Aufbau der Schicht, also durch die Nutzung unabhängiger Threads.
- **Sichere Datenübertragung**. Sie ist gewährleistet, da als Übertragungsprotokoll HTTP verwendet wird. HTTP wird bei den BlackBerry-Geräten standardmäßig verschlüsselt 2.4.
- **Geringes Datenvolumen**. Der Einbau eines Kompressionsverfahrens ermöglicht ein geringes Datenvolumen.

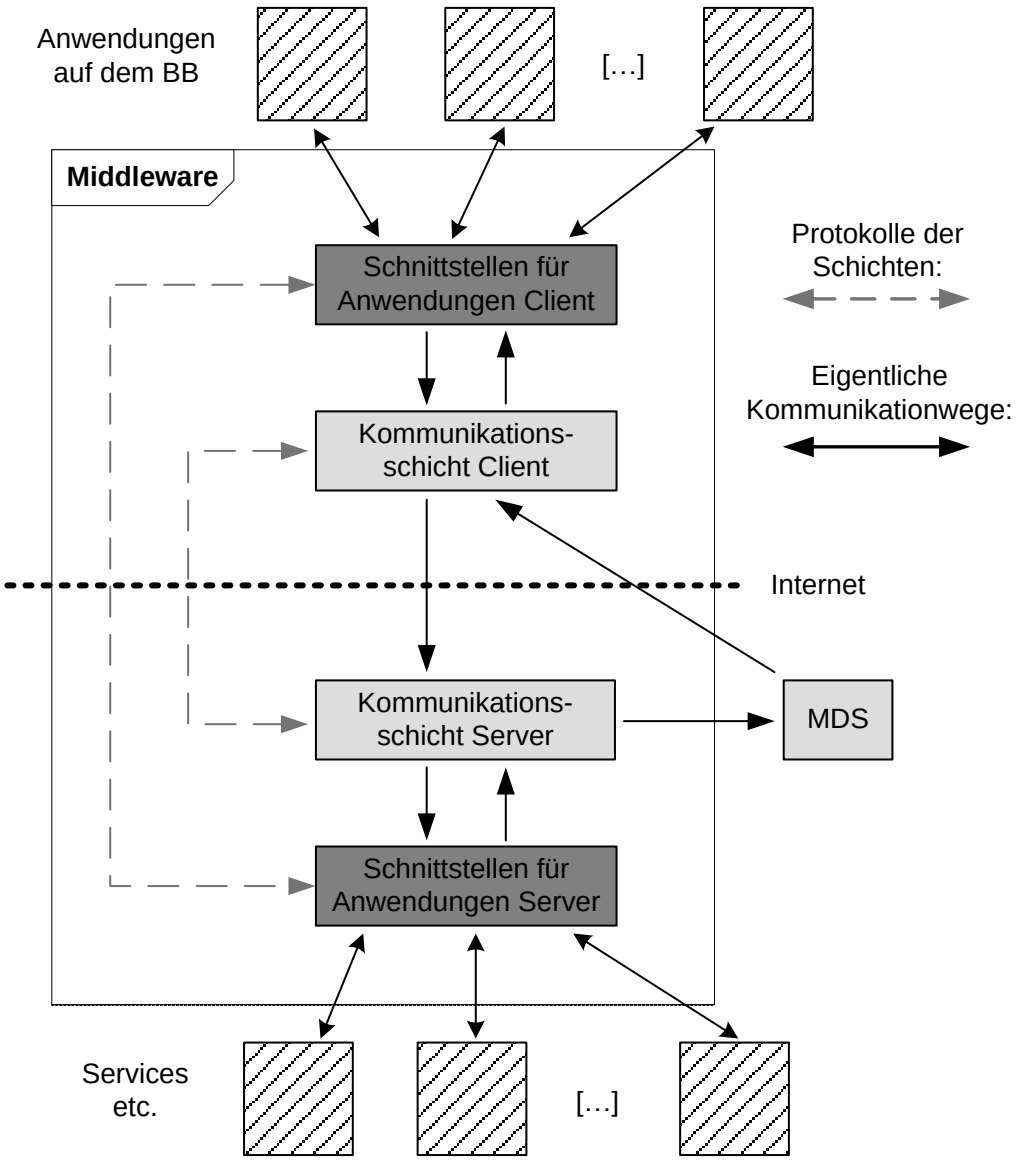


Abbildung 5.1: Aufteilung der Middleware und Kommunikationsrichtungen

5.1.2 Definition der Schnittstellen

Für den Nachrichtenaustausch zwischen den Schichten sind auf Client- und Serverseite Schnittstellen zu definieren. Diese sind auf beiden Seiten identisch und werden deshalb hier nur einmal aufgeführt.

- *newOutgoingMsg:* Schnittstelle, um Nachrichten von der Service- an die Kommunikationsschicht übergeben zu können. Hierdurch kann die Serviceschicht Nachrichten verschicken.
- *newIncomingMsg:* Schnittstelle, um Nachrichten von der Kommunikations- an die Serviceschicht übergeben zu können. Dadurch kann die Kommunikationsschicht empfangene Nachrichten zur Verarbeitung an die Serviceschicht weiterleiten.

Darüber hinaus sind noch Methoden zu implementieren, die es ermöglichen, die Abarbeitung der Kommunikationsschicht zu konfigurieren, zu pausieren oder zu beenden. Diese werden hier allerdings nicht weiter behandelt.

5.1.3 Programmstruktur

Die Abbildungen 5.2 und 5.3 zeigen die Schnittstellen der beiden Schichten im späteren Programm. Beide Schichten kommunizieren lediglich über die dazu bereitgestellten Schnittstellen miteinander. Die Parameter die beide Schichten übergeben bekommen und verarbeiten sind *Byte-Arrays*. Dies hat den Vorteil, dass die Kommunikationsschicht sich nicht mit Objekten auseinandersetzen muss, die in der Serviceschicht benutzt werden. Durch die Verwendung des einfachen Datentyps *Byte-Array*, wird somit eine Unabhängigkeit der Schichten erreicht.

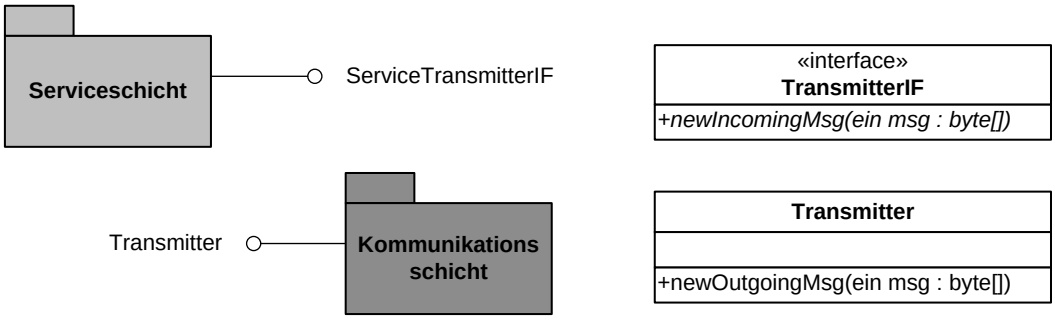


Abbildung 5.2: Schichten und Schnittstellen auf der Clientseite

Auf Clientseite (siehe Abbildung 5.2) stellt die Serviceschicht ein Interface (*TransmitterIF*) bereit, das von der Kommunikationsschicht benutzt werden kann. Dagegen greift die

Serviceschicht ohne ein Interface auf die *Transmitter*-Klasse zu. Dies wird dadurch möglich, dass die Kommunikationsschicht lediglich die Methoden öffentlich (*public*) macht, die auch von der Serviceschicht benutzt werden dürfen. Dies spart Ressourcen. Bei der Serviceschicht ist dies nicht möglich, da sowohl von Anwendungen als auch von der Kommunikationsschicht darauf zugegriffen wird und diese verschiedene Schnittstellen zur Verfügung gestellt bekommen müssen.

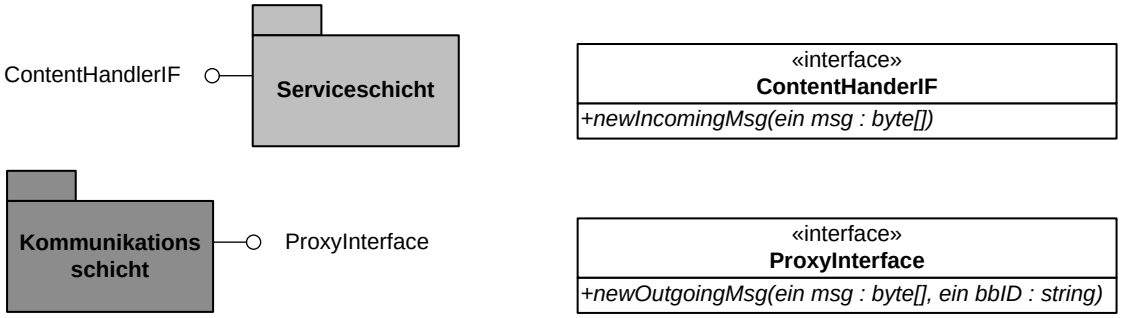


Abbildung 5.3: Schichten und Schnittstellen auf Serverseite

Die Schnittstellen auf Serverseite, sind ähnlich denen auf Clientseite (siehe Abbildung 5.3). Allerdings werden beide Schnittstellen über *Interfaces* realisiert. Die Schnittstelle der Kommunikationsschicht muss zusätzlich einen Parameter übergeben bekommen mit dem sich das Zielgerät eindeutig ansprechen lässt.

5.2 Schnittstellen für die Anwendungen

Nach der Aufteilung in Teilprojekte und Definition der Schnittstellen beschäftigt sich dieser Abschnitt mit den Anforderungen an die Schicht 2. Dazu wird dieses Kapitel nochmals in einzelne Komponenten unterteilt, die relativ unabhängig voneinander entwickelt werden. Es wird auf das benutzte Protokoll, die Schnittstellen für die Anwendungen, den Service auf der Clientseite und auf das Servlet eingegangen.

5.2.1 Protokoll

In Kapitel 4: *Stand der Forschung und Machbarkeitsstudie* ist die Entscheidung gegen die Nutzung eines existierendes Protokolls gefallen. In diesem Kapitel wird deshalb auf die Entwicklung eines eigenen Protokolls für die Kommunikation innerhalb der Schicht 2 der Middleware eingegangen. Hierzu werden erst die Anforderungen an das Protokoll beschrieben und danach die eigentliche Realisierung.

5.2.1.1 Anforderungen

Das Protokoll muss einige Funktionalitäten bereitstellen, um die Anforderungen erfüllen zu können. Hierzu werden zuerst die funktionalen Einträge von dem zu transportierenden Inhalt getrennt (Abbildung 5.4). Der funktionale Teil, der Header, enthält immer dieselben Einträge und kann, im Gegensatz zum Inhalt, vom Entwickler nicht angepasst werden. Er garantiert die Funktionalität innerhalb der Middleware.

Abbildung 5.4: Schematischer Aufbau des Protokolls

Für die Funktionalität sind zum einen die eindeutige Adressierung einer Anwendung auf einem BlackBerry, zum anderen die Zuordnung einer Antwort zu einer gestellten Anfrage nötig. Da das Protokoll verschiedene Daten transportieren soll, muss für den Empfänger außerdem ein Eintrag existieren, der es ihm ermöglicht, verschiedene Inhalte zu identifizieren und darauf entsprechend zu reagieren. Ohne diese Einträge wäre die Funktionalität nicht möglich. Zusätzlich werden zwei Einträge eingeführt, die nichts mit der Basisfunktionalität zu tun haben. Eine Versionsnummer, um später Probleme mit verschiedenen Versionen vorzubeugen, und ein Zeitstempel. Dieser ermöglicht das Implementieren von Kontrollmechanismen.

Der Header besteht also aus folgenden Einträgen:

- **Versionsnummer:**
  Die Client- und Serverkomponenten können so auf ältere bzw. neuere Versionen entsprechend reagieren. Neue Versionen können so abwärtskompatibel realisiert werden. Bei älteren oder nicht kompatiblen Versionen werden Nachrichten verworfen und richten so keinen Schaden an. Dies ermöglicht das parallele Betreiben von verschiedenen Versionen der Middleware bzw. des Protokolls.

- **Nachrichten-ID:**
  Dadurch sind Nachrichten eindeutig identifizierbar. Antworten können so den richtigen Anfragen zugeordnet werden. Dies ist ein elementarer Bestandteil der asynchronen Kommunikation, da mehrere Nachrichten verschickt werden können, bevor eine Antwort eintrifft. Die Antworten müssen außerdem nicht in der Reihenfolge eintreffen, in der die Anfragen geschickt worden sind. Ohne diese eindeutige Zuordnung wäre eine asynchrone Kommunikation nicht möglich.

- **Geräte-ID:**
  Erlaubt die eindeutige Identifizierung und Adressierung von BlackBerry-Geräten. Jeder BlackBerry hat eine solche einmalige ID, die fest an das Gerät gebunden ist. Diese Adresse wird von dem Push-Service, der von der BlackBerry-Infrastruktur bereitgestellt wird, benötigt, um Daten zurück an den BlackBerry zu senden.

- **Anwendungs-ID:**
  Nachdem die Daten auf dem richtigen Gerät sind, können sie so der richtigen Anwendung zugeordnet werden. BlackBerry-Anwendungen werden durch ihren Na-

men identifiziert. Auf einen Gerät können also nie zwei Anwendungen mit demselben Namen existieren. In Kombination mit der Geräte-ID können so beliebige Anwendungen auf beliebigen BlackBerry adressiert werden.

- **Zeitstempel:**
  Er wird beim Senden der Nachricht gesetzt und erlaubt das Erstellen von Statistiken. Da die Uhrzeiten zwischen Client (BlackBerry) und Server nicht synchronisiert werden, kann man zwar nicht auf die Übertragungszeit schließen, aber Pakete erkennen, die zu lange unterwegs waren. Zusätzlich lässt sich eindeutig die Reihenfolge der Nachrichten ermitteln, die von einem Gerät verschickt worden sind. Der Zeitstempel wird gesetzt, wenn die Nachricht an die Übertragungsschicht übergeben wurde, und nicht, wenn sie tatsächlich übertragen wird.
- **Inhalts-Typ:**
  Er wird beim Erstellen der Nachricht gesetzt, um den Inhalt auch auf der Gegenseite richtig extrahieren und verarbeiten zu können. Wenn die Nachricht gelesen wird, wird anhand des Inhalts-Typs entschieden, welches das richtige Modul zum Bearbeiten der Nachricht ist. Erst dieses Modul liest dann die Daten aus und verarbeitet sie.

5.2.1.2 Wahl des Übertragungsformates

Um Informationen zwischen zwei Systemen auszutauschen, benötigt man, neben dem definierten Inhalt, auch ein Format, in dem die Daten übertragen werden. Einige wünschenswerte Anforderungen an das Datenformat sind:

- Plattformunabhängigkeit
- Erweiterbarkeit
- Strukturierung
- Weite Verbreitung
- Geringer Bedarf an Speicherplatz und Rechenleistung
- Eine möglichst geringe Datenmenge

Um die Datenmenge, die übertragen werden muss, zu optimieren, ist ein selbst implementiertes binäres Format im Allgemeinen der beste Ansatz. Man kann das Format dann genau auf die Bedürfnisse der eigenen Applikation zuschneiden. Allerdings sind binäre Formate schlecht erweiterbar und schränken die systemunabhängige Nutzung ein. Sie sind somit ungeeignet. Anders textbasierende Übertragungsformate: Hierfür bieten sich CSV (Character Separated Values) oder XML (Extensible Markup Language) an. Mit CSV lassen sich allerdings nur schwer komplexe Strukturen darstellen. Zusätzlich existiert kein allgemeiner Standard. Somit muss man sich um das Kodieren der Nachrichten selbst kümmern.[1] Im Gegensatz dazu existieren bei XML durch das W3C (World Wide Web Consortium) festgelegte Standards und Implementierungen, die auch für den BlackBerry zur Verfügung stehen.[2] Zudem erfüllt XML, bis auf die Effizienz, alle der oben genannten Anforderungen:

- Plattformunabhängiges Datenformat
- Erweiterbar, d.h. es können selbst definierte Element- und Attributnamen verwendet bzw. hinzugefügt werden
- Trennung von logischer Struktur (XML-DTDs bzw. XML-Schemas), Daten und Repräsentation
- Für den Menschen lesbar
- Verbreiteter Standard, wodurch der Aufwand der Einarbeitung sehr gering ausfällt

Durch den relativ großen Bedarf an Speicherplatz und eine aufwändige Verarbeitung, die einiges an Rechenleistung braucht, muss man bei der Verwendung von XML allerdings einen Kompromiss zwischen Performance und Kompatibilität eingehen. Aufgrund der immer größer werdenden Leistungsfähigkeit von mobilen Endgeräten und der Steigerung der Übertragungsgeschwindigkeit durch beispielsweise UMTS fallen diese Nachteile allerdings immer geringer ins Gewicht. Hinzu kommt, dass die Effizienz ein generelles Problem von textbasierten Übertragungssystemen ist. Ob diese bei einer eigenen CSV-basierenden Implementierung besser wäre, ist fraglich.[3]

[1] Wikipedia Foundation (2007b)
[2] World Wide Web Consortium (2007)
[3] Borchert (2003)

5.2.1.3 Aufbau des Protokolls

Nach der Entscheidung für XML als Übertragungsprotokoll muss nun festgelegt werden, wie die zu übertragenden Daten, die in dem Unterkapitel Anforderungen festgelegt wurden, aufgebaut sind. Hierzu wurde zunächst eine XML-Schema Datei erstellt, die den Aufbau definiert. Diese kann später auch für die Validierung einer Nachricht benutzt werden, bevor diese weiterverarbeitet wird.

Zunächst werden die Header-Einträge umgesetzt (Listing 5.1). Die MessageID wird als Attribut in dem BBCMsg-Element, welches das Root-Element der gesamten Nachricht ist, realisiert. Die anderen Einträge werden dann einfach als Kind-Elemente von BBCMsg umgesetzt. Bis auf den Inhalt werden alle Einträge mit Datentypen realisiert, die in XML bereits vorhanden sind. Für den Inhalt wird zunächst ein abstrakter Inhalts-Typ (content-base) eingeführt, um diesen später durch verschiedene Inhalte ersetzen zu können.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified">
3    <!-- Erstellen des Message Elements (root)-->
4    <xs:element name="BBCMsg" type="Message" />
5    <!-- Definition des Message Elements -->
6    <xs:complexType name="Message">
7      <xs:sequence>
8        <xs:element name="PIN" type="xs:String" />
9        <xs:element name="AppID" type="xs:string" />
10       <xs:element name="Time" type="xs:float" minOccurs="0"/>
11       <xs:element name="Content" type="content_base" />
12     </xs:sequence>
13     <xs:attribute name="ID" type="xs:float" use="required" />
14   </xs:complexType>
15   <!-- Definition des abstracten content_base Elements zur
        einheitlichen behandlung aller spaeteren Inhalte-->
16   <xs:complexType name="content_base" abstract="true" />
```

Listing 5.1: Protokoll Definition: Header (XML-Schema)

Um den abstrakten Inhalts-Typen zu ersetzen, müssen alle Inhalte von diesem Typ erben. Dies geschieht bei der Definition der eigentlichen Inhalte. Dies wird an einem Beispielin-

halt für eine Datenbankabfrage deutlich(Listing 5.2). Der Rückgabewert (Result) wird hier vereinfacht als Text (String) dargestellt.

```
17  <!-- Definition eines ersten Beispiel Inhalts: Datenbankabfrage
        ; Erbt von content_base-->
18  <xs:complexType name="102">
19    <xs:complexContent>
20      <xs:extension base="content_base">
21        <xs:sequence>
22          <xs:element name="User" type="xs:string" />
23          <xs:element name="Pwd" type="xs:string" />
24          <xs:element name="Driver" type="xs:string" />
25          <xs:element name="URI" type="xs:string" />
26          <xs:element name="Query" type="xs:string" />
27      <xs:element name="Result" type="xs:string" minOccurs="0"/>
28        </xs:sequence>
29      </xs:extension>
30    </xs:complexContent>
31  </xs:complexType>
```

Listing 5.2: Protokoll Definition: DB-Abfrage-Inhalt (XML-Schema)

Um die Austauschbarkeit des Inhalts zu verdeutlichen, hier noch ein weiteres Beispiel. Diesmal ein Konfigurationsinhalt, um gegebenenfalls die Server-Adresse zu ändern (Listing 5.3). Auf diesem Server läuft später der Server-Teil der Middleware. An ihn werden alle Nachrichten vom BlackBerry aus verschickt und von dort aus weiterverarbeitet (siehe Abschnitt 5.2.3: Proxyservlet).

```
32  <!-- Definition eines zweiten Beispiel Inhalts: Konfiguration
        des Services; Erbt von content_base-->
33  <xs:complexType name="501">
34    <xs:complexContent>
35      <xs:extension base="content_base">
36        <xs:sequence>
37          <xs:element name="proxyIP" type="xs:anyURI" />
38        </xs:sequence>
39      </xs:extension>
40    </xs:complexContent>
41  </xs:complexType>
42 </xs:schema>
```

Listing 5.3: Protokoll Definition: Konfigurations-Inhalt (XML-Schema)

Die Namen der Beispielinhalte sind numerisch. Eine textuelle Bezeichnung kann sinnvoller erscheinen, allerdings ist die Behandlung von Zahlen (Integern) im späteren Programm einfacher und schneller. Sie wird deshalb hier bevorzugt.

5.2.1.4 Übertragung

Zur Verdeutlichung des formalen Aufbaus des Protokolls werden hier zwei Beispiele zu den XML-Schemas gezeigt. Zunächst die Datenbankabfrage (Listing 5.4). Der Typ, der im *Content*-Tag angegeben ist (Zeile 5), entspricht dem Namen in der Typendefinition (Listing 5.2, Zeile 18). Der abstrakte Typ wird so durch einen der erstellten Typen ersetzt.

```
1  <BBCMsg ID="1579546472">
2    <PIN>24f71167</PIN>
3    <AppID>dummyDB_App</AppID>
4    <Time>1173890076940</Time>
5    <Content xs:type="102">
6      <User>root</User>
7      <Pwd>root_pwd</Pwd>
8      <Driver>com.microsoft.sqlserver.jdbc.SQLServerDriver</Driver>
9      <URI>jdbc:sqlserver://da-dyma-as-05\dummyDB:2726</URI>
10     <Query>SELECT * FROM tbl_dummy;</Query>
11   </Content>
12 </BBCMsg>
```

Listing 5.4: Nachricht zur Datenbankabfrage

Für den Konfigurationsinhalt gilt dasselbe. Es muss nur der entsprechende Typ eingetragen werden (Listing 5.4, Zeile 5). In das Typ-Attribut im *Content*-Tag wird wieder der entsprechende Name aus der Typendefinition (Listing 5.3, Zeile 33) übernommen.

```
1  <BBCMsg ID="1579546472">
2    <PIN>24f71167</PIN>
3    <AppID>BBC_Service</AppID>
4    <Time>1173890076940</Time>
5    <Content xs:type="501">
6      <proxyIP>da-dyma-sa-11:8080/BBCServlet</proxyIP>
7    </Content>
```

```
8 </BBCMsg>
```

Listing 5.5: Nachricht zum konfigurieren der Servlet-Adresse

Diese Struktur erlaubt später die Validierung der Nachrichten, ohne dass dafür zusätzliche Funktionen für jeden Inhaltstypen geschrieben werden müssen. Inhaltstypen können allgemeingültig behandelt werden. Es muss lediglich eine XML-Schema Datei nach diesem Muster existieren bzw. erstellt werden. Zusätzlich dient das Typ-Attribut im *Content*-Tag im späteren Programm zur Identifikation des Inhaltes und zum Bestimmen der weiteren Verarbeitung. Diese Daten werden dann als Inhalte der HTTP-Nachrichten versendet.

5.2.1.5 Protokollstruktur im Programm

Nachdem der Aufbau des Protokolls festgelegt ist, wird die Struktur im eigentlichen Programm umgesetzt. Durch das Verwenden von XML und der Nutzung von Vererbung kann diese fast ohne Veränderung ins Programm übernommen werden. Dies wird im folgenden Klassendiagramm (Abbildung 5.5) deutlich.

Beschreibung der Klassen:

- **Message**
  Die *Message* Klasse bildet den *BBCMsg*-Typ des XML-Schemas ab. Alle Einträge, die im Header gebraucht werden, sind hier gekapselt. Zusätzlich werden eine Reihe von Funktionen und Konstruktoren bereitgestellt:
  - **Message(content:Content)** Ein *Message*-Objekt wird erstellt und bekommt einen *Content* übergeben. Bei dem Setzen des Inhalts wird dessen Typ im *Messasge*-Objekt gespeichert um ihn später wieder auslesen zu können. Diese Methode wird benutzt, nachdem ein *Content*-Objekt erstellt wurde und verschickt werden soll.
  - **Message(msg:byte[])** Eine seriallisierte XML Nachricht, wie in den Listings 5.4 und 5.5 zu sehen, wird so zu einem XML Dokument Objekt (*Document)*. Dies bildet die XML-Struktur im Programm ab. Es wird bei eingehenden Nachrichten benutzt.

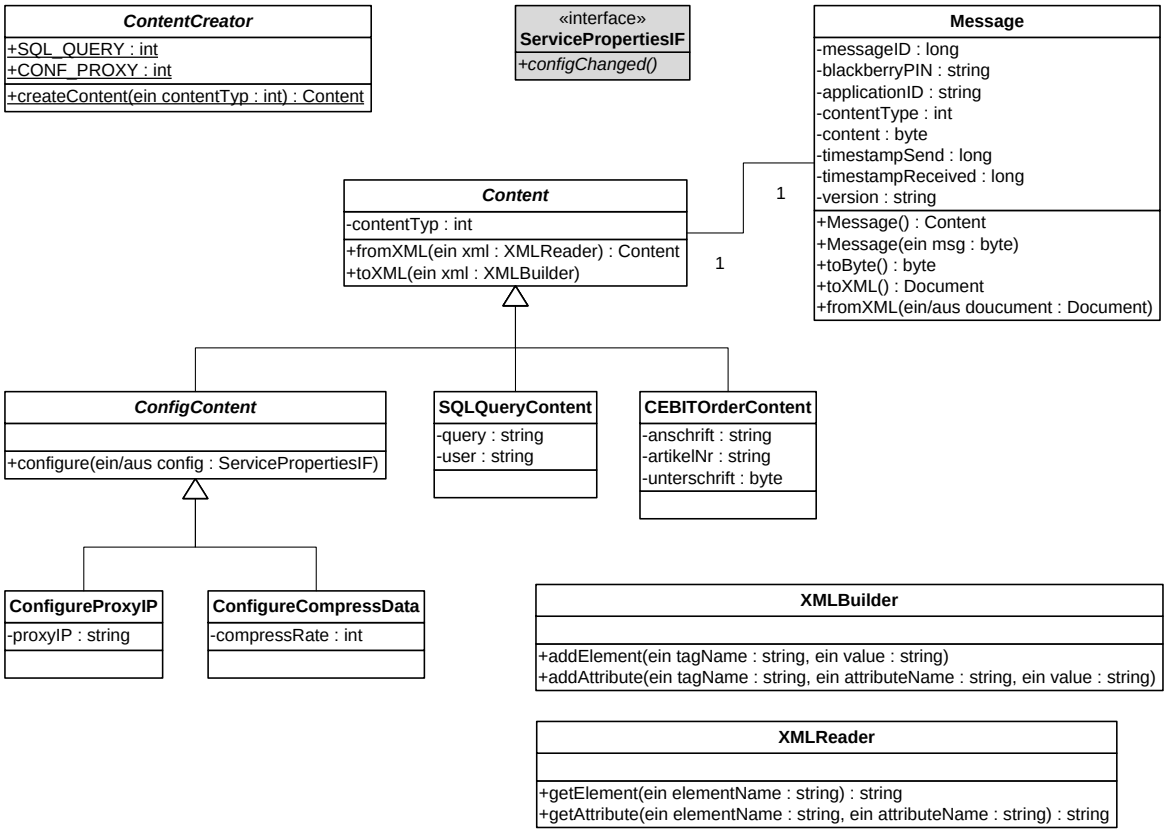


Abbildung 5.5: Klassendiagramm Inhalts-Typen

- **fromXML(doc:Document):void** Die Daten werden aus dem *Document*-Objekt gelesen und in die *Message* Attribute geschrieben.
- **toXML():Document** Das Gegenstück zu fromXML(). Die Daten werden in das *Document*-Objekt geschrieben.
- **toByte():byte[]** Das Gegenstück zu Message(byte[ ]). Die Daten aus dem *Document* werden seriallisiert und so später über HTTP übertragen.

- **Content**

  Der abstrakte *Content* ist die Basisklasse für alle späteren *Content* Typen. Nur Spezialisierungen von dieser Klasse können von einem *Message*-Objekt aufgenommen werden. Als Attribut hat es eine Typen-ID. Dadurch wird gewährleistet, dass

jeder *Content* später diese ID hat. Zusätzlich beinhaltet es einige abstrakte Funktionen, die dadurch in jeder Spezialisierung implementiert werden müssen:

- **fromXML(reader:XMLReader):Content** Wird beim Deseriallisieren einer Nachricht von dem *Message*-Objekt aufgerufen, in dem der Inhalt gekapselt ist, um das *Content*-Objekt mit den entsprechenden Werten aus der XML-Datei zu füllen. Wird beim Empfangen benutzt um den Inhalt wieder herzustellen.ialisieren einer Nachricht von dem *Message*-Objekt aufgerufen, in dem der Inhalt gekapselt ist, um das *Content*-Objekt mit den entsprechenden Werten aus der XML-Datei zu füllen. Wird beim Empfangen benutzt um den Inhalt wieder herzustellen.
- **toXML(builder:XMLBuilder):void** Das Gegenstück zu *fromXML()*. Wird beim Senden benutzt, um aus dem Objekt eine XML-Struktur zu generieren.
- **start():void** Diese Funktion existiert nur auf der Serverseite. Sie definiert, wie die am Server angekommenen Daten weiterverarbeitet werden. Sie wird aufgerufen, nachdem alle Werte des *Content*-Typs zur Verfügung stehen, zum Beispiel die eigentliche Ausführung einer Datenbankabfrage.

Erst diese gemeinsamen Schnittstellen machen die gemeinsame Verarbeitung aller *Content*-Typen möglich.

- **ContentCreator**
  Diese Klasse stellt eine statische Methode bereit, um *Content*-Objekte zu erstellen. Zusätzlich stellt sie Variablen zur Verfügung, welche die IDs der einzelnen *Content*-Typen enthalten. Dadurch sind die *Conent*-Typen, und speziell deren IDs, an einer zentralen Stelle zu verwalten. Zusätzlich wird der *ContentCreator* benutzt, um während der Laufzeit des Programms den anhand der ID bestimmten richtigen *Content*-Typ zu erstellen.
- **SQLQueryContent**
  Eine Spezialisierung des abstrakten *Content*-Typs, der auf dem Server eine Datenbankabfrage ausführt und das Ergebnis an den Client zurücksendet. Dieser Typ ist nicht auf ein Programm zugeschnitten und dient als allgemeine Schnittstelle an eine serverseitige Datenbank.

- **CEBITOrderContent**
  Ein weiteres Beispiel, diesmal ein speziell auf ein Programm zugeschnittener Inhalt. Es wurde in einer Demo-Anwendung für die CeBIT benutzt.

- **ConfigContent**
  Eine weitere Spezialisierung von *Content*. Diese Klasse ist ebenfalls abstrakt und dient als Basis, um Konfigurations-Inhalte an den Client zu senden. Sie enthält die abstrakte Methode:

  - **configure(config:ServicePropertiesIF):void** Diese Funktion wird immer aufgerufen, wenn ein *ConfigContent* auf dem Gerät eintrifft. Sie bekommt eine Schnittstelle zur Bearbeitung der Konfiguration übergeben, die von den späteren Spezialisierungen benutzt werden kann, um die Konfiguration anzupassen.

- **ConfigProxyIP**
  Eine Konfigurations-Nachricht zur Anpassung der Server-Adresse (Serverseite der Middleware). Sie kann vom Server aus an die Clients verschickt werden. Die Clients schicken ihre Nachrichten zukünftig an die neue Adresse.

- **ConfigureCompressData**
  Eine weitere Spezialisierung des *ConfigContent*. Sie dient zum Einstellen der Kompressionsrate, mit der die Daten komprimiert werden, bevor sie über HTTP versendet werden.

- **XMLBuilder und XMLReader**
  Diese beiden Klassen dienen als Hilfsklassen bei der Erstellung von XML Dokumenten. Sie kapseln ein *Document*-Objekt und stellen vereinfachte Schnittstellen zur Verfügung, um schreibende oder lesende Operationen auszuführen. Diese Vereinfachung schränkt allerdings die Erstellung bzw. das Auslesen von komplexen Strukturen etwas ein. Das Dokument kann jedoch auch direkt bearbeitet werden, wenn nötig. Da die meisten *Content*-Spezialisierungen eine einfache Struktur haben, wird dies nur selten nötig und erleichtert die Arbeit (siehe Listing 5.6). Zudem kann der benutzte XML-Parser durch einen anderen ersetzt werden, ohne dass andere Stellen im Programm davon betroffen sind.

```
1 //Beispiel ohne XML Builder:
2 child = document.createElement(BLACKBERRY_PIN);
3 child.appendChild(document.createTextNode(_blackberryPIN));
4 msg.appendChild(child);
5 //Beispiel mit  XML Builder:
6 xml.addElement(MESSAGE, BLACKBERRY_PIN, _blackberryPIN);
```

Listing 5.6: Beispiel: Vereinfachung durch XMLBuilder

5.2.2 Client

Auf der Clientseite, also auf dem BlackBerry, ist die Middleware als ein Service implementiert. Dieser läuft permanent im Hintergrund. Bei diesem Service können sich Anwendungen registrieren und dann über ihn Nachrichten verschicken. Der Service empfängt auch alle eingehenden Nachrichten und stellt diese, anhand der Anwendungs-ID, der richtigen Anwendung zu.

5.2.2.1 Anforderungen

Die Anforderungen des Services können unterteilt werden in Funktionalitäten, die er den Anwendungen bereitstellen muss, und Funktionalitäten, die gegeben sein müssen, damit der Service überhaupt seine Aufgaben erfüllen kann.

Für die Anwendungen müssen folgende Funktionalitäten bereitgestellt werden:

- Das **Schicken** von Nachrichten. Die Anwendung muss das Verschicken nur anstoßen. Sie bekommt sofort eine Rückmeldung, der Service führt die gewünschten Operationen aus und kümmert sich darum, dass die Antwort die richtige Anwendung erreicht.
- Das **Empfangen** von Nachrichten. Bei einer eingehenden Nachricht wird ein Ereignis in der Anwendung ausgelöst, in dem die Nachricht weiter behandelt werden kann.
- Anwendungen müssen sich **registrieren** können. Nur Pakete registrierter Anwendungen werden verarbeitet, Pakete nicht registrierter Anwendungen werden verworfen.

- Anwendungen müssen sich **ein- und ausloggen** können. So ist dem Service bekannt, ob die Anwendung gestartet ist oder nicht. Wenn eine Anwendung läuft, während ein Paket ankommt, kann dieses direkt zugestellt werden. Falls nicht, wird es zwischengespeichert und der Anwendung bei ihrem nächsten Login übergeben.
- Die Schnittstellen müssen **einfach aufrufbar** sein. Es soll kein Unterschied bestehen, ob auf eigene Funktionen oder, mittels Interprozesskommunikation, auf die des Services zugegriffen wird.

Zusätzlich muss der Service selbst noch einige Anforderungen erfüllen, um als Kommunikationsplattform nutzbar zu sein:

- Er sollte vom Server aus **konfigurierbar** sein, also Over-The-Air (OTA). Das heißt, der Service ist immer konfigurierbar, sobald sich das Gerät in Reichweite des Mobilfunkanbieters befindet. Der Grundstein dafür wurde schon im Unterkapitel 5.2.1: *Protokoll* mit den Konfigurationsnachrichten gelegt. Diese müssen nach Ankunft nur noch entsprechend verarbeitet werden.
- Bestimmte Bereiche und Funktionen müssen gegen **Mehrfachzugriffe** geschützt werden. Es kann passieren, dass mehrere Anwendungen den Service gleichzeitig nutzen. Hierdurch dürfen keine Probleme entstehen.
- Der Service muss **zuverlässig** sein. Er darf nur minimale Ausfallzeiten haben, da er als Kommunikationsplattform für Anwendungen dient, welche ansonsten nur im Offline-Betrieb laufen könnten.
- Er muss **Nachrichten zwischenspeichern** können. Falls eine Anwendung nicht aktiv ist, wenn eine Nachricht für sie eintrifft, müssen die Nachrichten so lange im Service gehalten werden, bis die Anwendung wieder aktiv wird. Für den Fall, dass in dieser Zeit das Gerät oder der Service abgeschaltet werden, muss diese Sicherung persistent sein.

5.2.2.2 Adapter für die Anwendung

Um die Funktionalitäten für die Anwendung verfügbar zu machen, ist, da der Service als eigener Prozess läuft, die Realisierung von Interprozesskommunikation nötig. Damit man

sich beim Entwickeln der Anwendung nicht darum kümmern muss, wird die Interprozesskommunikation zwischen den Anwendungen und dem Middleware-Service vollständig in eine Klasse (Adapter) gekapselt. Diese muss nur noch in die Anwendung eingebunden werden (siehe Abbildung 5.6). Somit ist das Registrieren, das Ein- und Ausloggen sowie das Übergeben der zu sendenden Nachrichten an den Service von der Anwendungsseite aus gelöst. Um Nachrichten zu empfangen, muss noch die in Unterabschnitt 5.1.3 definierte Schnittstelle *BBConnectorIF* in die Anwendung eingebunden werden. Über diese Schnittstelle kann der Adapter die Nachrichten, welche er vom Service übergeben bekommt, an die Anwendung weiterleiten. Dort können diese weiterverarbeitet werden. Somit sind die Funktionalitäten, die der Service bietet, von der Anwendung wie lokale Funktionen aufrufbar.

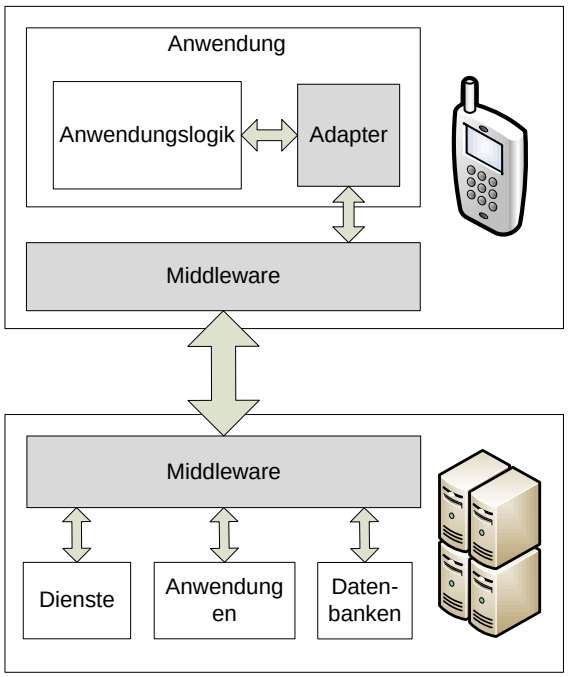


Abbildung 5.6: Interprozesskommunikation

Der Adapter verwaltet auch die ID mit der sich die Anwendungen eindeutig identifizieren lassen. Wie schon in Unterabschnitt 5.2.1 beschrieben, ist diese ID der eindeutige Name der Anwendung. In Abbildung 5.7 ist zu sehen, wie der Adapter im Programm umgesetzt worden ist.

Beschreibung der Klassen:

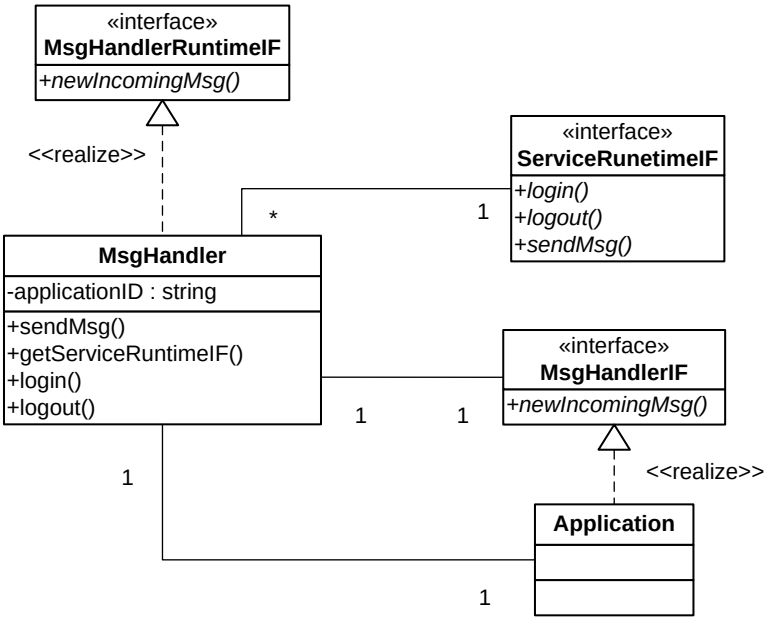


Abbildung 5.7: Klassendiagramm Anwendungsadapter: Message Handler

- **Application**
  Die Anwendungsklasse muss das in Kapitel 5.1.3: *Programmstruktur* definierte Interface *MsgHandlerIF* einbinden oder eine Klasse bereitstellen, die das Interface einbindet. Eine Referenz auf dieses Interface muss dem zu erstellenden Objekt *MsgHandler* übergeben werden. Die Anwendung greift über die MsgHandler Klasse auf die Funktionen des Services zu.

- **MsgHandler**
  Die MsgHandler Klasse wird von der Anwendung erstellt und bekommt eine Referenz auf das *MsgHandlerIF* übergeben. Über dieses Interface schickt es ankommende Nachrichten an die Anwendung weiter. Die Klasse muss selbst auch ein Interface einbinden, um die entsprechenden Nachrichten vom Server zu bekommen. Über dieses *MsgHandlerRuntimeIF* läuft sämtliche Interprozesskommunikation. Über die Methode *getServiceRuntimeIF* beschafft der MsgHandler sich eine Referenz auf die Schnittstelle *ServiceRuntimeIF*. Diese Schnittstelle wird vom Service so angeboten, dass Interprozesskommunikation darüber möglich ist.

Die Schnittstelle des Services wird lediglich vom Adapter angesprochen, anstatt von der

Anwendung selbst. Vom Service zur Anwendung hin läuft auch jede Kommunikation über den Adapter.

5.2.2.3 Service

Der Service ist das „Herz“ der Middleware auf der Clientseite. Über ihn läuft jede Kommunikation mit den Anwendungen. Die Anforderungen an den Service waren Konfigurierbarkeit, Zuverlässigkeit und das sichere Zwischenspeichern der Nachrichten. Die Threadsicherheit wird durch geschützte Bereiche realisiert, auf die immer nur eine Quelle zur selben Zeit zugreifen kann. Wie das genau gelöst ist, wird in Kapitel 6: *Realisierung* beschrieben, da dies programmiersprachenabhängig ist. Die Over-The-Air (OTA)-Konfigurierbarkeit wird mittels den Konfigurationsnachrichten ermöglicht, die schon im Unterabschnitt 5.2.1: *Protokoll* definiert worden sind. Diese müssen vom Service nur noch entsprechend verarbeitet werden. Auf die Stabilität des Services wird später in einem eigenen Abschnitt eingegangen. Die persistente Zwischenspeicherung der Nachrichten, falls eine Anwendung nicht aktiv ist, wird über einen Nachrichtenspeicher gelöst. Dieser wird immer ausgelesen wenn sich eine Anwendung am Service anmeldet. Anhand der *Anwendungs-ID* im Nachrichten-Header wird kontrolliert, ob eine Nachricht für die entsprechende Anwendung vorhanden ist. Diese wird dann übergeben und aus dem Speicher gelöscht. Damit der Service weiß, ob er eine Nachricht zwischenspeichern soll, muss er eine Liste über alle Anwendungen führen, die auf dem Gerät vorhanden sind und den Service nutzen. Beim ersten Login einer Anwendung wird diese in ein Register aufgenommen. Nur Nachrichten an Anwendungen, die dort registriert sind, werden vom Service verarbeitet.

Der Nachrichtenfluss im Service ist in Abbildung 5.8 schematisch dargestellt.

Eingehende Nachrichten werden zuerst auf Konfigurationsinhalte untersucht (siehe Verarbeitung **A**). Ist dies der Fall, wird die Konfiguration entsprechend verändert. Andernfalls wird überprüft, zu welcher Anwendung die Nachricht gehört, und dieser entsprechend zugestellt (siehe Verarbeitung **B**). Die Kriterien zur Verarbeitung einer Nachrichten stehen im Header. Nach der Verarbeitung wird die Nachricht ohne Header an das Ziel weitergeleitet. Im Falle von Anwendungen wird das Paket an die Adapter weitergeleitet. Diese übergeben es dann der Anwendung. Das Zustellen kann, wenn diese aktiv ist, sofort er-

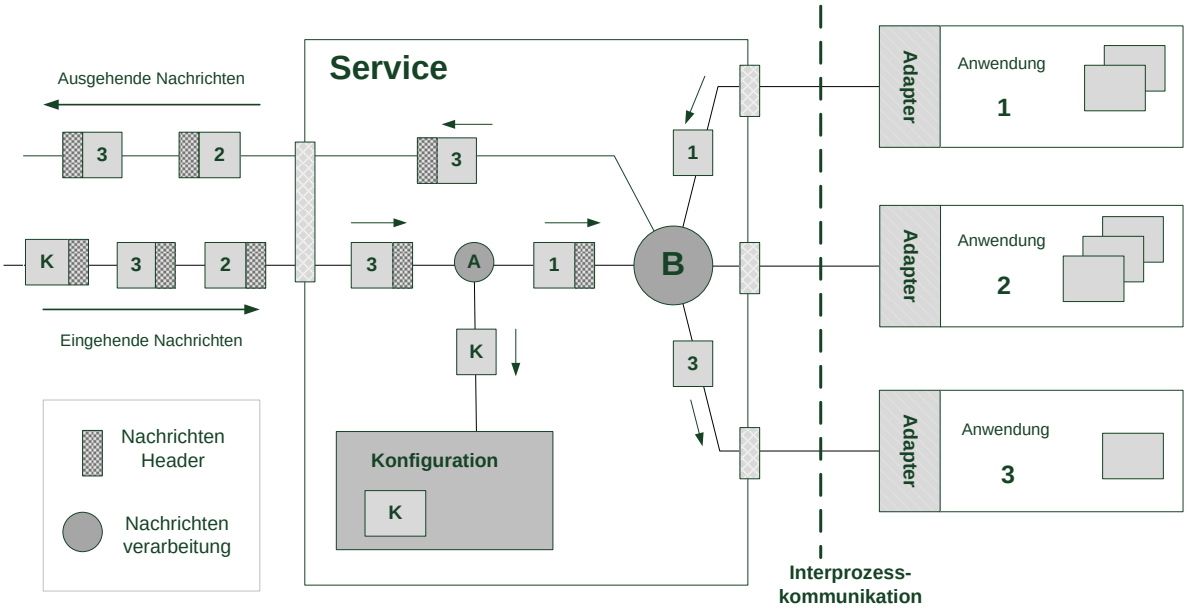


Abbildung 5.8: Nachrichtenverlauf im Service

folgen. Ist die Anwendung nicht aktiv, wird die Nachricht erst zwischengespeichert und beim nächsten Login übergeben.

Dasselbe Prinzip gilt beim Senden der Nachrichten. Der Inhalt wird von den Anwendungen erstellt und an den Adapter übergeben. Dieser übergibt diese dann an den Service. Der Service fügt dem Inhalt einen Header hinzu und leitet die Nachricht an die Kommunikationsschicht weiter. Diese verschickt die Nachricht an den Server, der in der Konfiguration eingetragen ist.

Im Programm ist der Service wie in Abbildung 5.9 aufgebaut.

Beschreibung der Klassen und Schnittstellen:

- **Service**
  Die Hauptklasse des Service. Sie erbt von *Application* und ist damit eine eigenständige Anwendung. Der Service ist standardmäßig so eingestellt, dass er automatisch gestartet wird, wenn das Gerät aktiviert wird. Er läuft im Hintergrund, bietet aber auch ein User-Interface (UI). In diesem UI können Konfigurationseinstellungen allerdings nur eingesehen werden, geändert werden kann die Konfiguration nur über entsprechende Nachrichten vom Server. Der Service kann hier auch manuell

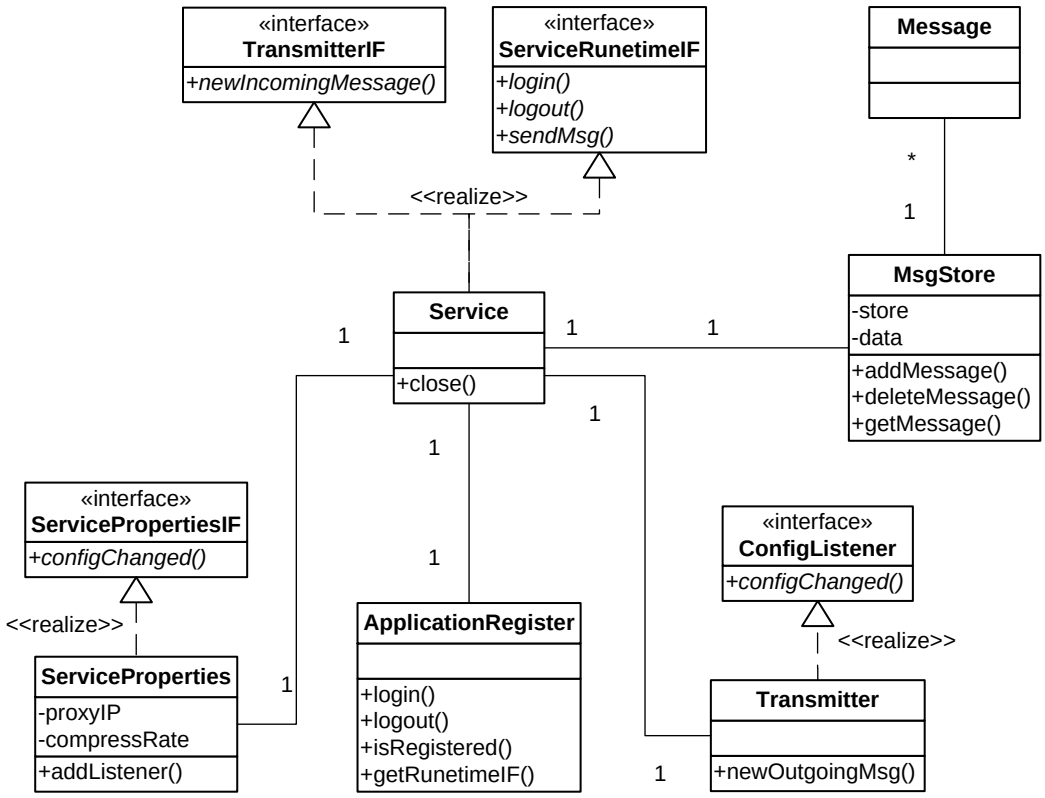


Abbildung 5.9: Klassendiagramm Service

gestoppt und gestartet werden. Er implementiert die *ServiceRuntimeIF* und die *TransmitterIF* Schnittstelle.

- **ServiceProperties**
  Hier sind alle Einstellungen des Services persistent gespeichert, zum Beispiel die Adresse des Servers oder die Kompressionsrate der zu verschickenden Nachrichten. Objekte die die *ConfigListener*-Schnittstelle implementiert haben, können sich hier registrieren und werden bei Änderungen in der Konfiguration benachrichtigt. Diese Klasse implementiert zusätzlich die *ServicePropertiesIF*-Schnittstelle.

- **ServicePropertiesIF**
  Diese Schnittstelle wird Konfigurationsnachrichten zur Verfügung gestellt, um die Konfiguration zu ändern. Sie „kontrolliert“, welche Einstellungen geändert werden dürfen.

- **ConfigListener**
  Die *ConfigListener*-Schnittstelle kann von Klassen implementiert werden, die über Änderungen in der Konfiguration des Services informiert werden müssen (siehe: *ServiceProperties* Klasse).

- **ApplicationRegister**
  Diese Klasse ist für das Registrieren von Anwendungen zuständig. Hier wird persistent gespeichert, welche Anwendungen auf dem Gerät installiert sind, die den Service nutzen. Der momentane Zustand einer Anwendung, also ob sie Ein- oder Ausgeloggt ist, ist hier ebenfalls gespeichert. Zusätzlich werden hier die Referenzen der einzelnen Anwendungen verwaltet, die für die Interprozesskommunikation notwendig sind. Diese werden dem Service bei dem Login der Anwendung übergeben.

- **MsgStore**
  Hier werden die Nachrichten zwischengespeichert, die zu einer Anwendung gehören, die im Moment nicht aktiv ist. Bei einem Login einer Anwendung wird dieser Speicher überprüft. Falls ein Eintrag für eine Anwendung vorhanden ist, wird dieser zugestellt und aus dem Speicher gelöscht.

- **Transmitter**
  Die Transmitter-Klasse kapselt die komplette Konfigurationsschicht. Nachrichten die verschickt werden sollen, werden an diese Schicht übergeben. Der Service bekommt über die *TransmitterIF* Schnittstelle eingehende Nachrichten von der Konfigurationsschicht. Diese Klasse implementiert die *ConfigListener*-Schnittstelle und ist bei der *ServiceProperties* Klasse registriert, um über Änderungen der Konfiguration informiert zu werden.

Die Schnittstellen *TransmitterIF* und *ServiceRuntimeIF* werden für die Interprozesskommunikation und für die schichtenübergreifende Kommunikation benötigt.

Stabilität des Services

Da der Service permanent im Hintergrund laufen soll und mehreren Anwendungen als Kommunikationsplattform dient, muss dieser besonders stabil und sicher laufen. Er darf sich nicht durch die Übergabe falscher Werte an den Schnittstellen oder durch Fehler, die während der Interprozesskommunikation auftreten können, aufhängen oder abstürzen. Fehler müssen also abgefangen und an die Anwendungen weitergegeben werden,

damit diese entsprechend darauf reagieren können. Dies wird durch eine hierarchisch aufgebaute Fehlerbehandlung erreicht. Dadurch kann auf detaillierte Fehlermeldungen des gesamten Programms reagiert werden. Gleichzeitig können schwere Fehler über einfache Schnittstellen weitergegeben und in der Anwendung ausgelesen werden. Ein Auszug aus der Exception Hierarchie zeigt Abbildung 5.10:

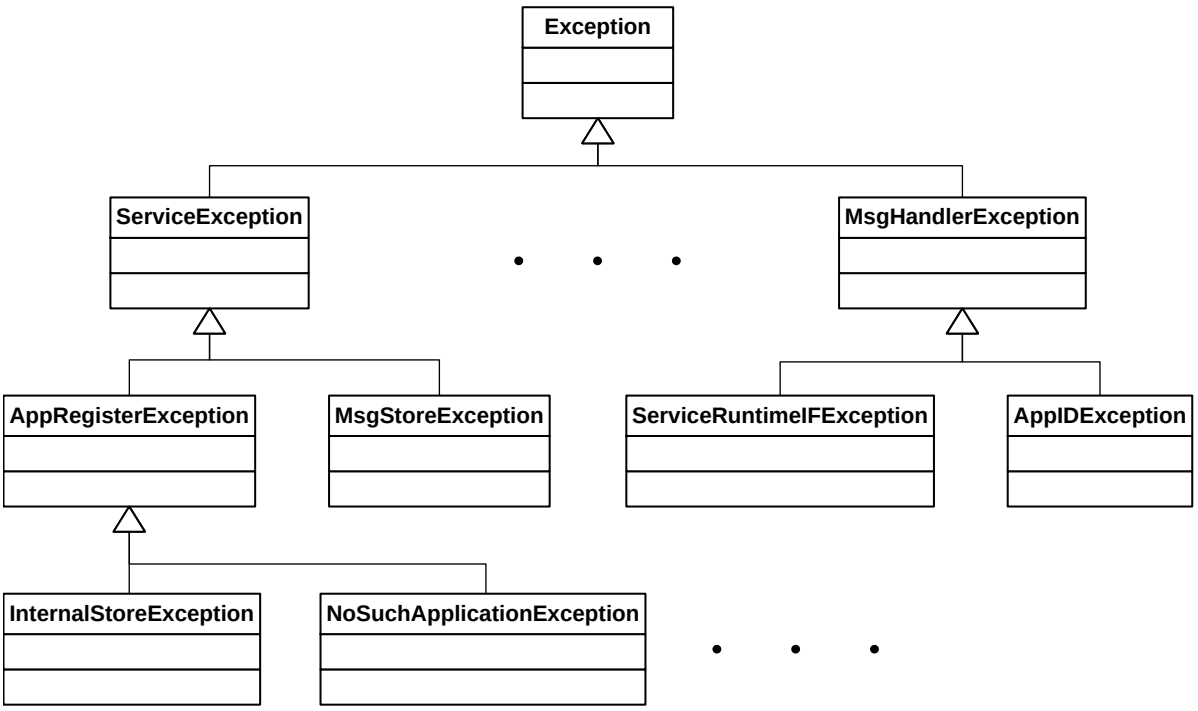


Abbildung 5.10: Auszug aus der Exception Hierarchie

5.2.3 Server

Wie in 4.4 beschrieben läuft der Serverteil der Middleware in einem Servlet. Alle Nachrichten laufen über diesen zentralen Knoten.

5.2.3.1 Anforderungen

Auf Serverseite müssen eingehende Nachrichten wieder extrahiert werden. Danach werden diese, je nach Inhalt, entsprechend verarbeitet. Die Ergebnisse können anschließend wieder zum Client zurückgesendet werden. Hier werden ähnliche Konzepte verwendet wie auch auf Clientseite. Es muss zu jedem Inhalts-Typ (*Content*) auf dem Client, ein entsprechendes Gegenstück auf Serverseite implementiert werden. Zusätzlich muss es eine Möglichkeit geben, die Aktionen festzulegen, die beim eintreffen einer Nachricht ausgeführt werden sollen (zum Beispiel Datenbankabfragen).

5.2.3.2 Struktur im Programm

Im Programm ist die Struktur ähnlich wie auf Clientseite (siehe Abbildung 5.5). Deswegen wird hier nur auf die Unterschiede eingegangen. In Abbildung 5.11 sind die elementaren Klassen der Serverseite dargestellt.

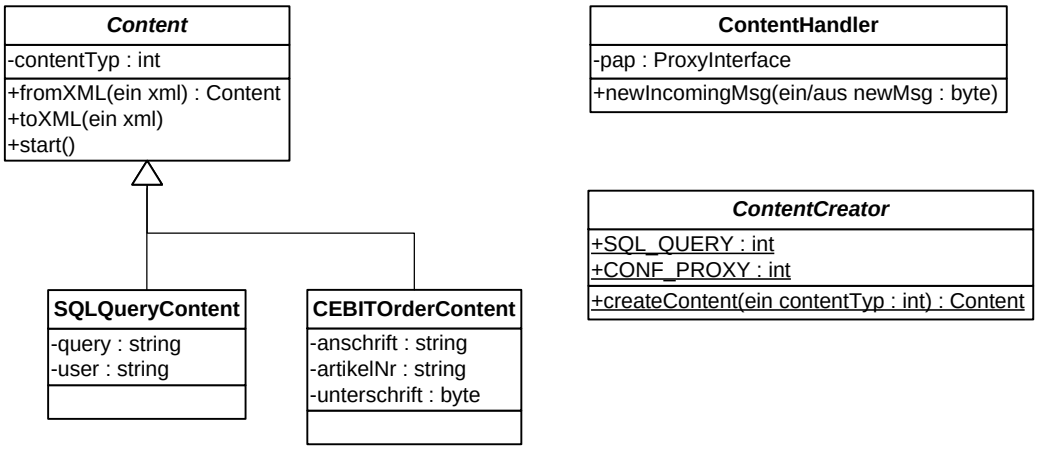


Abbildung 5.11: Klassendiagramm Server

Beschreibung der Klassen:

- **Content**
  Diese Klasse entspricht fast der auf dem Client. Lediglich die abstrakte Funktion *start* ist hinzugekommen. In dieser Funktion stehen die Aktionen, die auf Serverseite mit der Nachricht ausgeführt werden sollen. Zum Beispiel eine Datenbankabfrage, eine Bestellung oder Ähnliches.

- **ContentHandler**
  Ein Objekt dieser Klasse wird von der Serviceschicht erstellt wenn eine Nachricht eintrifft. Ihr wird die Schnittstelle *ProxyInterface* übergeben, um Antworten an den Client schicken zu können. Die Methode *newIncomingMsg* stellt dann die Nachricht wieder her und führt die *start* Methode des spezifischen Inhaltes aus.

Die restlichen Klassen sind, wie oben schon erwähnt, der Clientseite sehr ähnlich und können dort nachgeschlagen werden (siehe design:client).

5.2.4 Zusammenspiel der Komponenten

Hier soll anhand von Sequenzdiagrammen das Zusammenspiel, der in den letzten Unterkapiteln vorgestellten Klassen, dargestellt werden. In Abbildung 5.12 wird das Senden einer Nachricht vom Client dargestellt. Die Abbildung 5.13 zeigt das Empfangen einer Nachricht auf Serverseite (dabei sind die Grau gefärbten Objekte Teile der Kommunikationsschicht).

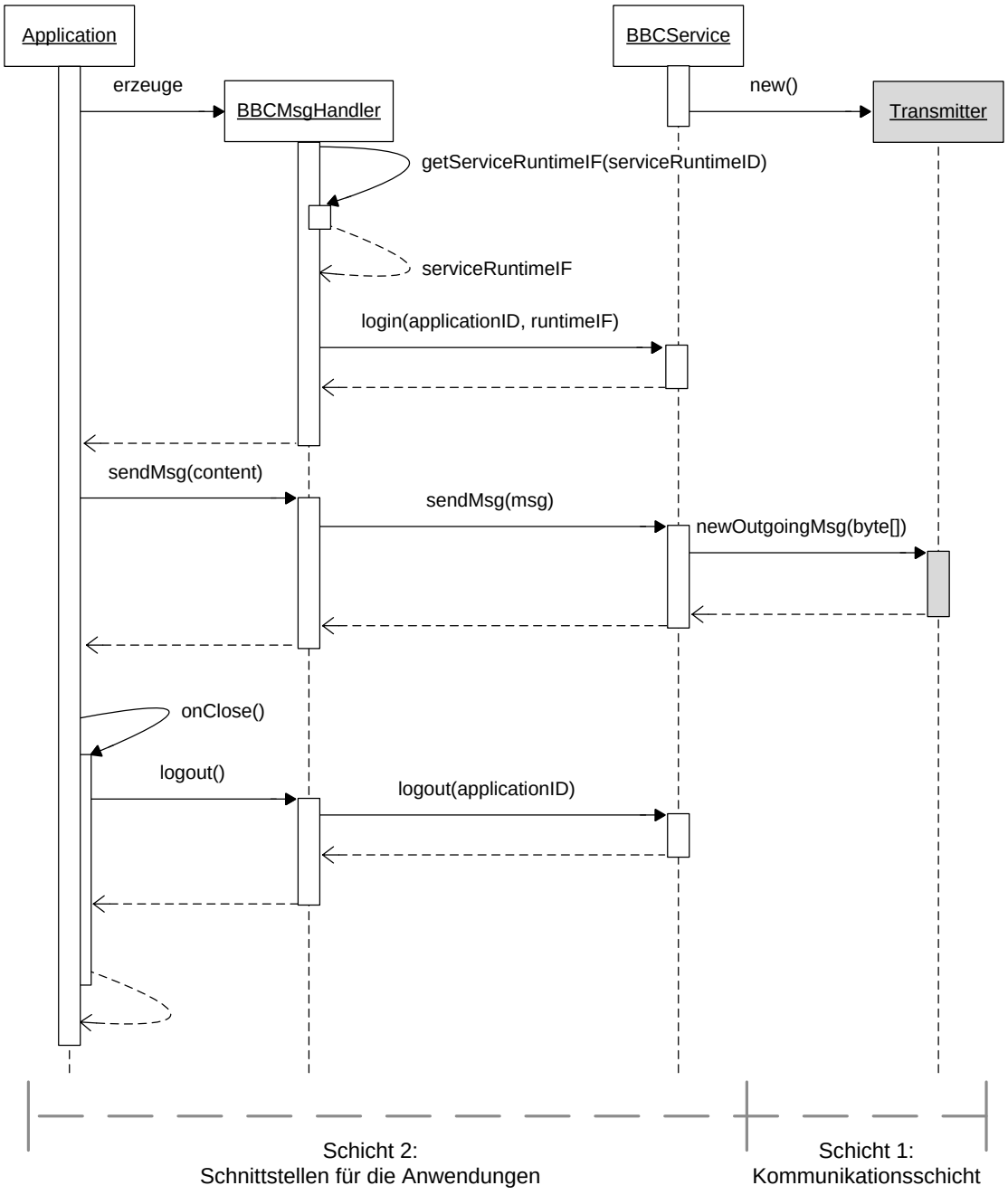


Abbildung 5.12: Sequenzdiagramm: Senden einer aynchronen Nachricht mit Login und Logout der Anwendung

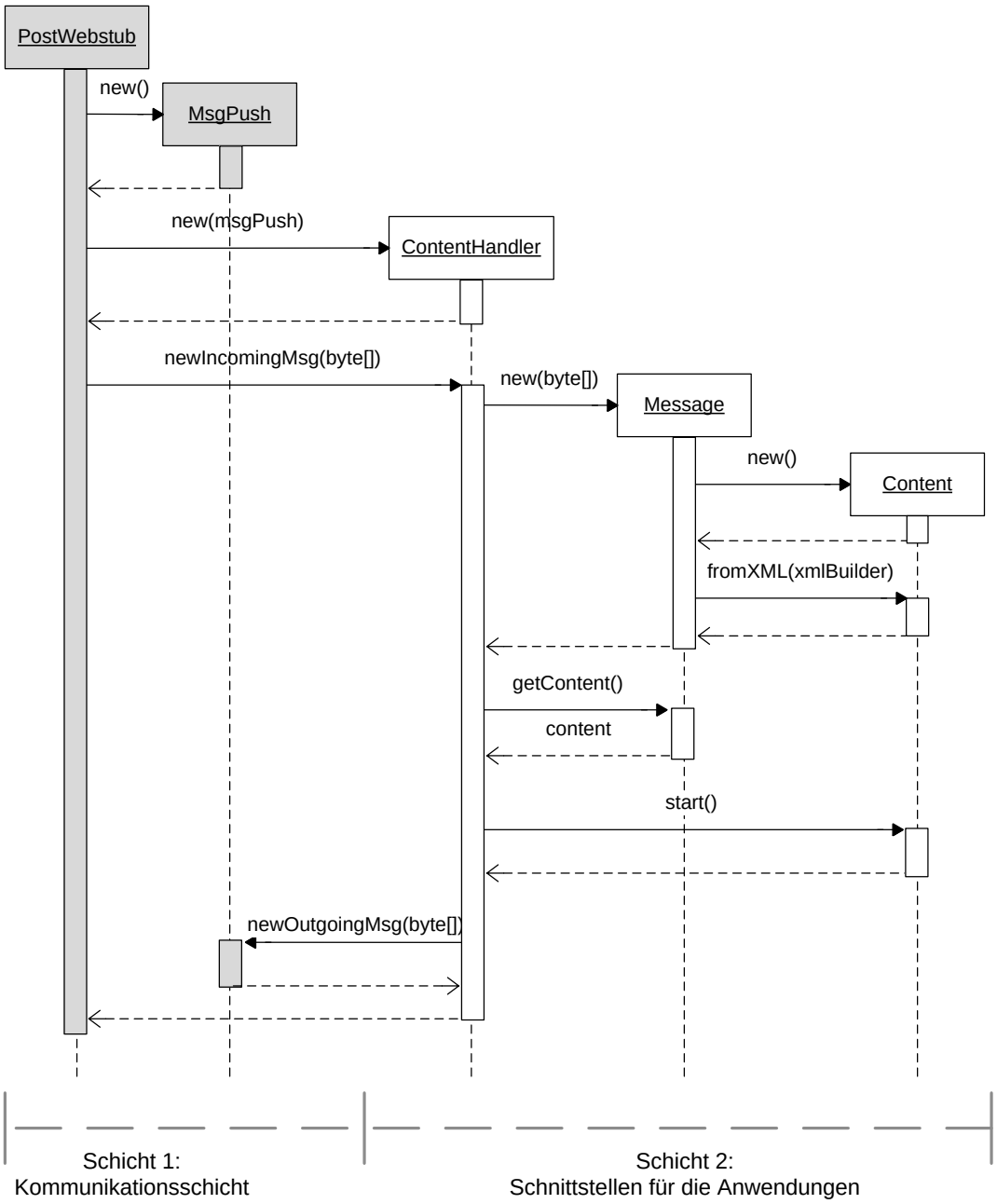


Abbildung 5.13: Sequenzdiagramm: Empfangen einer Nachricht auf Serverseite

5.3 Kommunikation innerhalb der Middleware

Dieses Kapitel beschreibt die Schicht 1 - die Kommunikationsschicht. Sie ist für die Kommunikation zwischen der Client-Komponente und der Server-Komponente zuständig und stellt diese Funktionalitäten der Serviceschicht zur Verfügung.

Im Folgenden wird, anhand der Ergebnisse der Machbarkeitsstudie, der Aufbau des Protokolls erläutert, also wie beide Teile, der Clientteil auf dem BlackBerry und der Serverteil auf einem Application-Server im Unternehmen, miteinander kommunizieren. Es werden Lösungen zu den Anforderungen entworfen, auf die Eigenheiten mobiler Datenübertragung eingegangen und gezeigt, wie diese Problematiken behoben werden. Der strukturelle Aufbau der Kommunikationsschicht wird im danach folgenden Kapitel spezifiziert, sowohl auf Clientseite als auf Serverseite. Abschließend wird kurz auf Sicherheitsaspekte bei der Übertragung empfindlicher Daten und deren Einbau in die Middleware eingegangen.

5.3.1 Anforderungen

Im Folgenden wird auf Grund der Anforderungen aus Kapitel 3.2 und der Aufteilung aus 5.1 auf die Probleme eingegangen, die zu beachten sind. Nach diesen Designkriterien wird der Aufbau der Kommunikationsschicht realisiert. Für für die Kommunikation sind folgende Aufgaben zu beachten:

- **Push-Nachrichten zum BlackBerry**
  Zur Verwirklichung der Push-Funktionalität ist auf Clientseite ein Prozess nötig, der eintreffende Nachrichten entgegennimmt. Er muss ständig aktiv sein und auf einem zu definierenden Port lauschen. Serverseitig müssen Methoden bereitgestellt werden, um einen Nachrichten-Push zu generieren.
  ⇒ Push-Dienst und auf eintreffende Nachrichten lauschender Prozess.
- **Offlinefähigkeit**
  Zur Gewährleistung einer einwandfreien Funktion muss die Middleware clientseitig bei sendebereiten Daten mit Zeiträumen ohne Mobilfunknetz zurechtkommen.

Während dieser Zeit darf die Funktionsweise der gesamten Middleware nicht beeinträchtigt werden. Es ist daher nötig, die zu sendenden Nachrichten zwischenzupuffern, während auf Empfang des Gerätes gewartet wird. Erst bei Empfangsbereitschaft eines Carriers[4] werden die Übertragungsversuche unternommen. Serverseitig wird ein Servlet-Container verwendet.

Es ist also erforderlich, einen Puffer zu realisieren, der die Nachrichten aufnehmen kann. Dieser soll persistent auf dem Gerät speicherbar sein, um Datenverlust durch einen Systemabsturz zu vermeiden.
⇒ Es folgt, einen Puffer zu realisieren, als FIFO-Speicher ausgelegt.
Weiterhin soll der Kommunikationsversuch nur unternommen werden, wenn erstens auch ein Carrier vorhanden ist, und zweitens auch Nachrichten zur Übertragung vorliegen.
⇒ Ein Listener ist zu implementieren, der auf Statusänderungen des Carriers reagiert.

- **Dateneffizienz**
  Um die Datenmenge gering zu halten, ist es nötig, sie zu komprimieren, da die Kommunikationsschicht den Aufbau und den Inhalt der Nachricht nicht mehr ändern kann und soll.
  ⇒ Es muss eine Komprimierungsroutine eingebaut werden.

Zu diesen Anforderungen kommen clientseitig weitere Probleme durch verteilte Systeme hinzu. Eine mobile Anwendung ist eine Form eines verteilten Systems 2.2:

- **Asynchronität**
  Zur Realisierung der geforderten Offlinefähigkeit (siehe Kapitel 2.2: *Verteilte Systeme*) und für die verschiedenen parallel ablaufenden Prozesse ist **asynchrones** Verhalten nötig. Die asynchrone, parallele Abarbeitung von Codeblöcken ist nötig, damit die Middleware nicht blockiert, während sie kommuniziert. Auf Grund der Java-Plattform auf den BlackBerry Geräten wird eine threadbasierte Implementierung gewählt.

  Zu beachten ist hierbei, dass Multithreading in Java abhängig ist vom Betriebssystem, auf dem die JVM oder KVM läuft. Die Virtuelle Java Maschine greift

[4] Mobilfunknetz-Betreiber

auf Funktionen des Betriebssystems zurück. Bei Multithreading schaltet das Betriebssystem, das alle Ressourcen eines Rechners verwaltet, nur schnell zwischen einzelnen Threads hin und her. Gerade laufende Threads werden gestoppt und die Prozessorzeit wird anderen Threads zugeteilt. Für die Umschaltung gibt es mehrere gängige Verfahren: Präemptives Multitasking und kooperatives Multitasking sind die häufigsten Implementierungen bei aktuellen Betriebssystemen. Präemptives Multitasking zeichnet sich dadurch aus, dass das Betriebssystem die Kontrolle über die einzelnen Threads hält und die Prozessorzeit abhängig von Prioritäten und nach festen Zeiten an die Threads vergibt. Kooperatives Multitasking dagegen zeichnet sich dadurch aus, dass die einzelnen Threads von sich aus ihre Rechenzeit an den nächsten Thread abgeben. Blockiert hier ein Thread, blockiert das gesamte System. Das BlackBerry-Gerät benutzt ein proprietäres Betriebssystem, weswegen eine gesicherte Aussage hier nicht möglich ist.

- **Kommunikationsform**
  Ein grundlegendes Problem der Middleware ist der plötzliche Verbindungsabriss bei einer Übertragung. Es ist daher wichtig, welche Kommunikationsart zu wählen ist.

  Die gewählte Kommunikationsart beim Senden von Nachrichten vom Client zum Server, ist das Rendezvous. Das bedeutet, die Kommunikation ist synchron und meldungsorientiert. Der Client öffnet eine Verbindung zum Server und überträgt die Nachrichten. Wird der Empfang vom Server als korrekt bestätigt, so wird die Verbindung wieder getrennt. Die Sendung vom Server zum Client dagegen ist asynchron und meldungsorientiert. Sie ist also vom Typ Datagramm. Das liegt daran, dass der Server die Antwortnachricht nicht an den BlackBerry selbst versendet, sondern an eine Zwischeninstanz des BlackBerry-Netzwerkes, Den sogenannten BlackBerry Enterprise Server (BES). Dieser leitet die Nachricht dann an das entsprechende Endgerät weiter. Eine Rückmeldung im Sinne von Ergebnissen ist nicht nötig. Es ist lediglich zu kontrollieren, ob die Nachricht erfolgreich gesendet wurde. Zu beachten ist noch, dass sich die Kommunikation durch die Implementierung für die nächst höhere Schicht als asynchroner, entfernter Dienstaufruf darstellt oder als Datagramm, falls keine Rückantwort erwartet wird.

  Da der Empfänger, also der Server, als Java-Servlet innerhalb eines Application

Servers laufen soll, kann er mehrere Fragesteller gleichzeitig bedienen. Die Adressierung ist daher asymmetrisch, und direkt, weil der Server mit einem Unique Ressource Identifier (URI) erreichbar ist.

Auf Grund des Fazits aus der Machbarkeitsstudie wird HTTP als Übertragungsprotokoll gewählt. Innerhalb der Middleware werden HTTP-POST Nachrichten verschickt.

- **Fehlersemantik**
  Die verwirklichte Fehlersemantik ist „At-Least-Once“. Hierfür wird sich eine Eigenheit des unter dem HTTP liegenden TCP-Protokolls zu Nutze gemacht. Da TCP ein zuverlässiges, verbindungsorientiertes Transportprotokoll ist, erkennt es selbstständig Datenverluste und behebt diese automatisch. Der Drei-Wege-Handshake des TCP, dient dabei dem Aufbau einer Kommunikationsverbindung, der Teardown deren Abbau. Innerhalb dieser Grenzen kann das Protokoll Datenfehler beheben.

  Erst wenn die Nachricht korrekt und vollständig empfangen wurde, schickt der Server eine HTTP-200 Antwort zum Client zurück und beendet die Verbindung. Hierfür nimmt der Server den gesamten Bytestrom des Client entgegen und versucht diesen zu entschlüsseln. Gelingt ihm dies, so wurde die Nachricht korrekt gesendet. Ein weiterer Vorteil in diesem Vorgehen besteht darin, dass es so relativ einfach ist, in die Entschlüsselung des Datenstromes auch gleichzeitig noch Packroutinen einzubauen, um das Datenaufkommen zu senken.

Nachdem nun die Designkriterien festgelegt sind, wird auf den Aufbau der Kommunikationsschicht eingegangen. Dies geschieht zum besseren Verständnis anhand eines typischen Verlaufs einer Nachrichtenversendung in Schicht 1 (siehe Abbildungen 5.14 und 5.15).

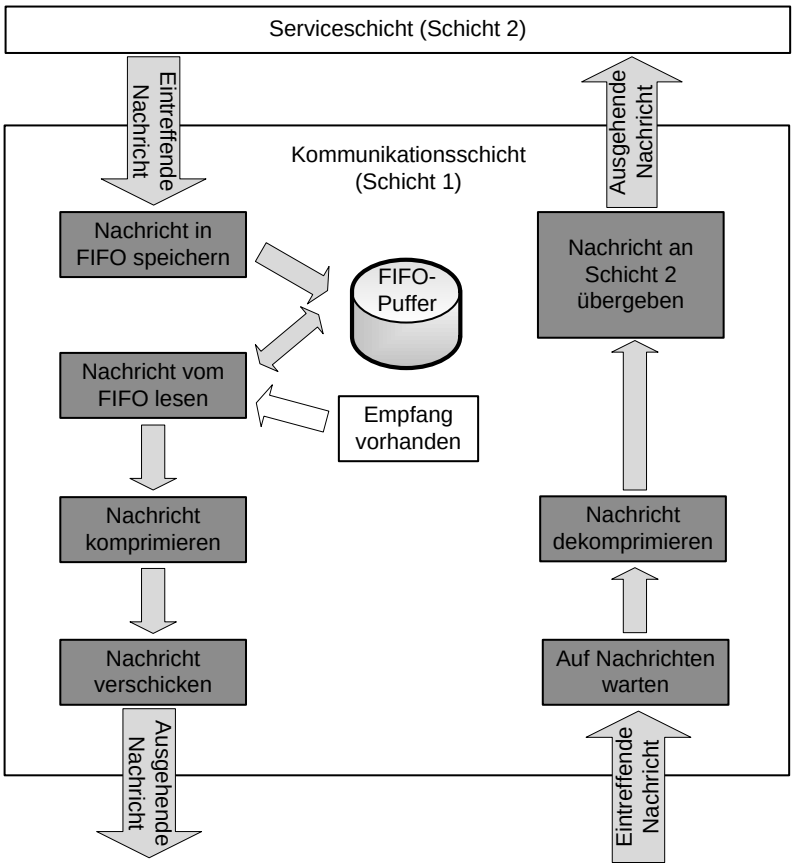


Abbildung 5.14: Verlauf einer Nachrichtensendung auf dem Client

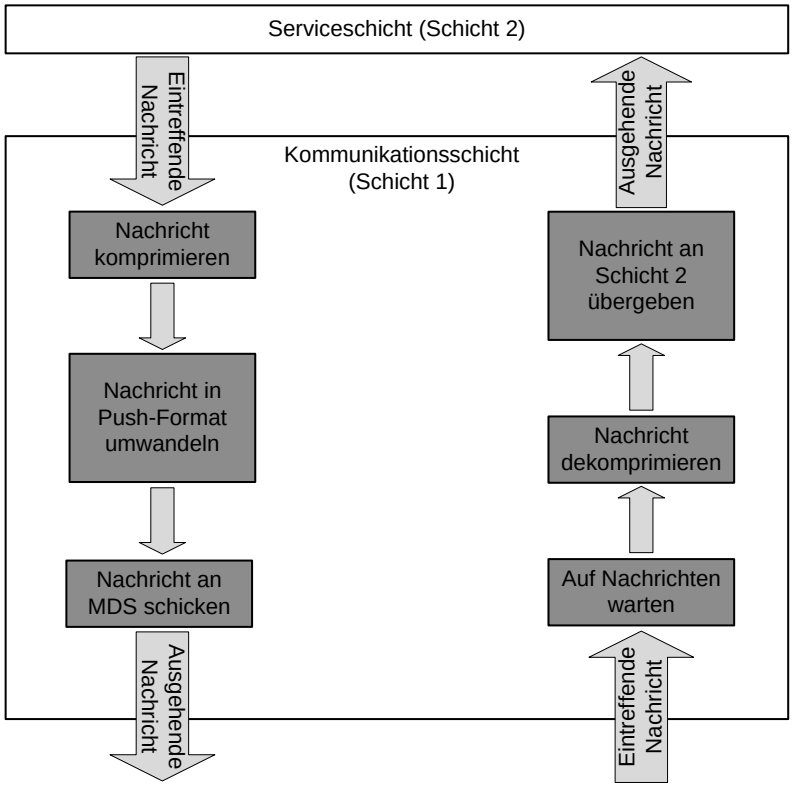


Abbildung 5.15: Verlauf einer Nachrichtensendung auf dem Server

5.3.2 Struktureller Aufbau der Kommunikationsschicht

Anhand der Abbildungen 5.14 und 5.15 wird nun der strukturelle Aufbau der Kommunikationsschicht auf Server- und Clientseite genau definiert.

5.3.2.1 Clientseite

Allgemein Die gesamte Schicht wird durch mehrere Threads aufgebaut. Die Kommunikation nach außen erfolgt durch threadbasierte Prozeduren, ebenso wie der Datenaustausch innerhalb der Schicht. Die Verbindung zwischen eingehenden, zu sendenden Daten und dem eigentlichen Versenden dieser Daten erfolgt über eine Warteschlange. Sie dient als Puffer, um beide Teile - den schreibenden und den lesenden Teil - voneinander zu trennen und unabhängig zu machen (Abbildung 5.16). Darüber hinaus ist ein Listener zu implementieren, der auf Statusänderungen des Mobilfunkempfangs reagiert. Eingehende

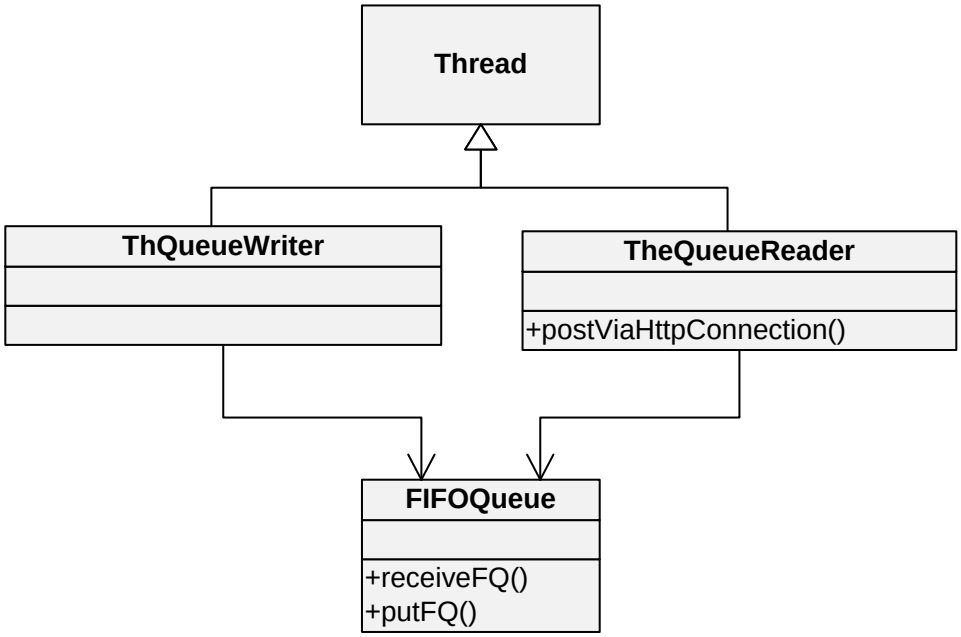


Abbildung 5.16: Warteschlange mit beiden darauf zugreifenden Klassen

Nachrichten von der oberen Schicht, werden von einem Thread in die Warteschlange geschrieben. Ein weiterer Thread fragt diese Schlange ab, entfernt vorhandene Nachrichten und versendet sie. Eingehende Nachrichten vom Serverpart werden entgegen genommen und durch eine Schnittstelle an die Serviceschicht übergeben.

Nachrichten an den Server schicken Nachdem der *StatusListener* (siehe weiter unten) den Empfang eines Mobilfunknetzes gemeldet hat, wird, nachdem eine Nachricht von der Serviceschicht in die Warteschlange geschrieben wurde, diese von der Klasse *ThQueueReader* ausgelesen und komprimiert. Anschließend wird eine HTTP-Verbindung zum Server geöffnet und die Nachricht übertragen. Die Serverantwort wird ausgewertet und anhand dessen entschieden, ob ein Übertragungsfehler stattgefunden hat und die Nachricht zurück in die Warteschlange geschrieben wird. Ist das der Fall, wartet der Thread nach dem Zurückspeichern in den FIFO eine gewisse Zeitspanne und versucht die Übertragung dann erneut. Den gesamten Ablauf zeigt Abbildung 5.19.

Warteschlange Sie dient als Datenpuffer für ausgehende Nachrichten. Konzipiert ist sie als FIFO, also als First-In-First-Out-Puffer: Nachrichten werden anhand ihres Eintreffens an der Kommunikationsschicht bei der ersten beginnend abgearbeitet. Die Warteschlange ist nur den beiden Threads *ThQueueReader* und *ThQueueWriter* zugänglich. Während *ThQueueWriter* neu zu sendende Nachrichten in diesen Puffer schreibt, liest *ThQueueReader* diese aus und versendet sie an das Gegenstück der Kommunikationsschicht auf dem Server. Die Warteschlange wird bei jeder Änderung persistent auf dem Gerät gespeichert, um Datenverlust bei einem plötzlichen Programm- oder Geräteausfall zu vermeiden.

Statuslistener Damit die Kommunikationsschicht auf Statusänderungen des Mobilfunkempfangs reagieren kann, ist es nötig, einen Listener zu implementieren, der diese Statusänderungen vom Gerät mitgeteilt bekommt und entsprechend reagiert. Ein Listener ist ein Prozess, der auf Ereignisse reagiert, indem er sich bei einer Instanz anmeldet und von dieser informiert wird[5]. Dieser Listener stoppt und startet den Sende-Thread, ab-

[5] Tanenbaum u. van Steen (2003)

hängig vom Status des Mobilfunkempfangs. So ist es möglich, nur dann Ressourcen zu verbrauchen, wenn eine Übertragung prinzipiell überhaupt möglich ist.

Daten packen Um ein möglichst geringes Datenvolumen zu erhalten und so die Kosten zu senken und die Effizienz zu steigern, werden die Nachrichten gepackt. Zum Komprimieren und Dekomprimieren der Daten wird GZIP verwendet. GZIP ist ein offenes Packformat, dass durch die General Public License (GPL) lizenziert ist und somit quelltextoffen ist. GZIP basiert auf dem Deflate-Algorithmus, einer Kombination der Huffman-Kodierung mit einem Algorithmus von Lempel, Ziv, Storer und Szymanski[6]. Es ist auf den meisten Plattformen verfügbar, so auch für die Java-Umgebung der BlackBerry-Geräte.

Vom Server eintreffende Nachrichten Um Nachrichten empfangen zu können, ist ein weiterer Thread nötig: *PAPReceiver*. Diese Klasse öffnet einen Port auf dem BlackBerry-Gerät und wartet auf ankommende Nachrichten vom Server beziehungsweise auf Push-Nachrichten der BlackBerry-Server (nähere Informationen siehe Kapitel 2.4: *BlackBerry*). Die ankommenden Daten werden entkomprimiert und einem weiteren Thread übergeben, der diese über die Schnittstelle *TransmitterIF* an die Serviceschicht weiterleitet. Dazu implementiert die Serviceschicht die Prozedur *newIncomingMessage()*.

Öffentliche Prozeduren Um Nachrichten von der Serviceschicht entgegennehmen zu können, wird eine Hauptklasse *BBTransmitter* entworfen. Diese Komponente stellt sich für die obere Schicht als BlackBox dar. Sie verfügt über öffentliche Prozeduren, mit denen sie konfiguriert und eingeschränkt gesteuert werden kann:

1. **newOutgoingMessage()**
   Die wichtigste Methode innerhalb der Kommunikationsschicht. Sie wird von der Serviceschicht aufgerufen, um Nachrichten zu versenden. Die Prozedur übergibt die Nachricht an die *ThQueueWriter*-Klasse, die sich um das Speichern in der Warteschlange kümmert. Dies geschieht direkt über den Konstruktor der Klasse, welcher auch gleich die eigene *run()*-Methode ausführt. So ist der Einsprungpunkt, also *newOutgoingMessage()* des BBConnectors nicht blockiert und steht für weitere

[6] RFC (1996)

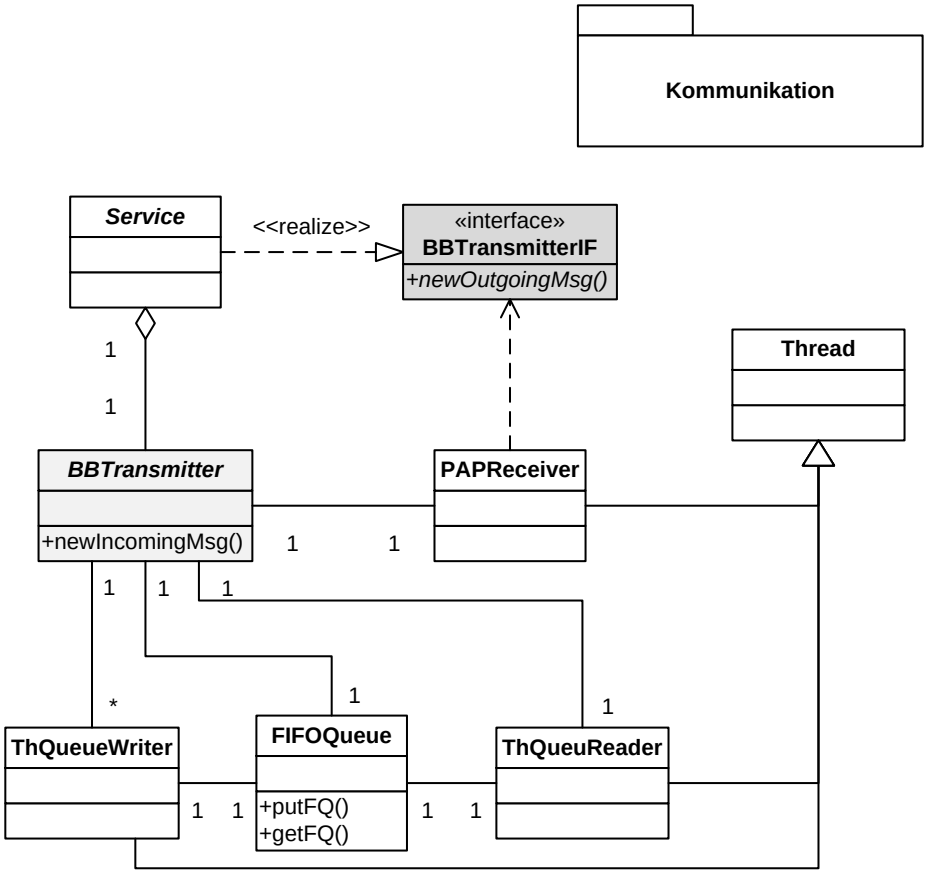


Abbildung 5.17: Klassendiagramm der Kommunikationsschicht

Nachrichten zur Verfügung (siehe 5.18). Die Prozedur erwartet zur Weiterverarbeitung ein Bytearray.

2. setDataCompression()
   Diese Funktion dient zur Einstellung der Kompressionsrate der Nachrichten. Als Kompressionsalgorithmus ist GZIP gewählt. Als Kompressionsraten kann zwischen 0 (keiner Kompression) und 9 (höchste Kompression) gewählt werden.

3. setProxyUrl()
   Diese Methode dient dem setzen der Server-URL. Die *BBConnector*-Klasse ruft hier wiederum ihre als *Private* markierte *ThQueueReader*-Klasse auf und übergibt dieser die neue Empfängeradresse. Ein gerade stattfindender Kommunikationsversuch bleibt davon unbetroffen. Sollte sich die Serveradresse innerhalb eines Über-

tragungsversuches ereignen, so führt das einfach zu einer Fehlermeldung, weil die alte Adresse nicht erreichbar ist. Die aktuelle Nachricht wird also wieder in den FIFO-Puffer gesichert, und der Übertragungsversuch beginnt von neuem, wenn die Adressänderung übernommen wurde.

4. reRunQueueReader()
   Über diese Prozedur kann die Serviceschicht die Kommunikationsschicht veranlassen, alle Fehlerzähler zurückzusetzen und die Übertragungsversuche neu zu beginnen. Der tatsächliche Beginn neuer Übertragungsversuche ist abhängig vom Empfang eines Mobilfunknetzes: Der zu Beginn beschriebene Listener hat eine höhere Priorität.

5. close()
   Diese Prozedur dient der kompletten Abschaltung der Kommunikationsschicht. Alle Threads werden über ihre entsprechenden Funktionen beendet, der Listener wird abgeschaltet und nimmt keine Ereignisse mehr entgegen. Die Warteschlange bleibt im lokalen Speicher vorhanden und kann bei erneutem Start der Kommunikationsschicht weiterverwendet werden.

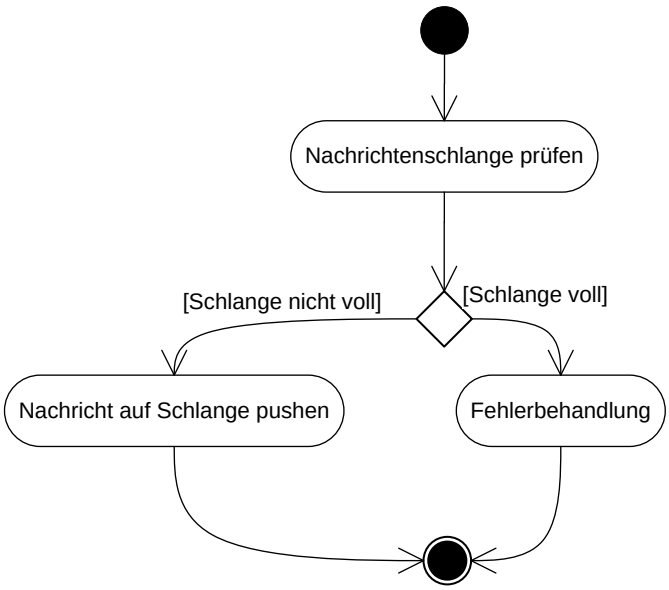


Abbildung 5.18: Ablaufdiagramm von Nachricht-Versenden Teil 1

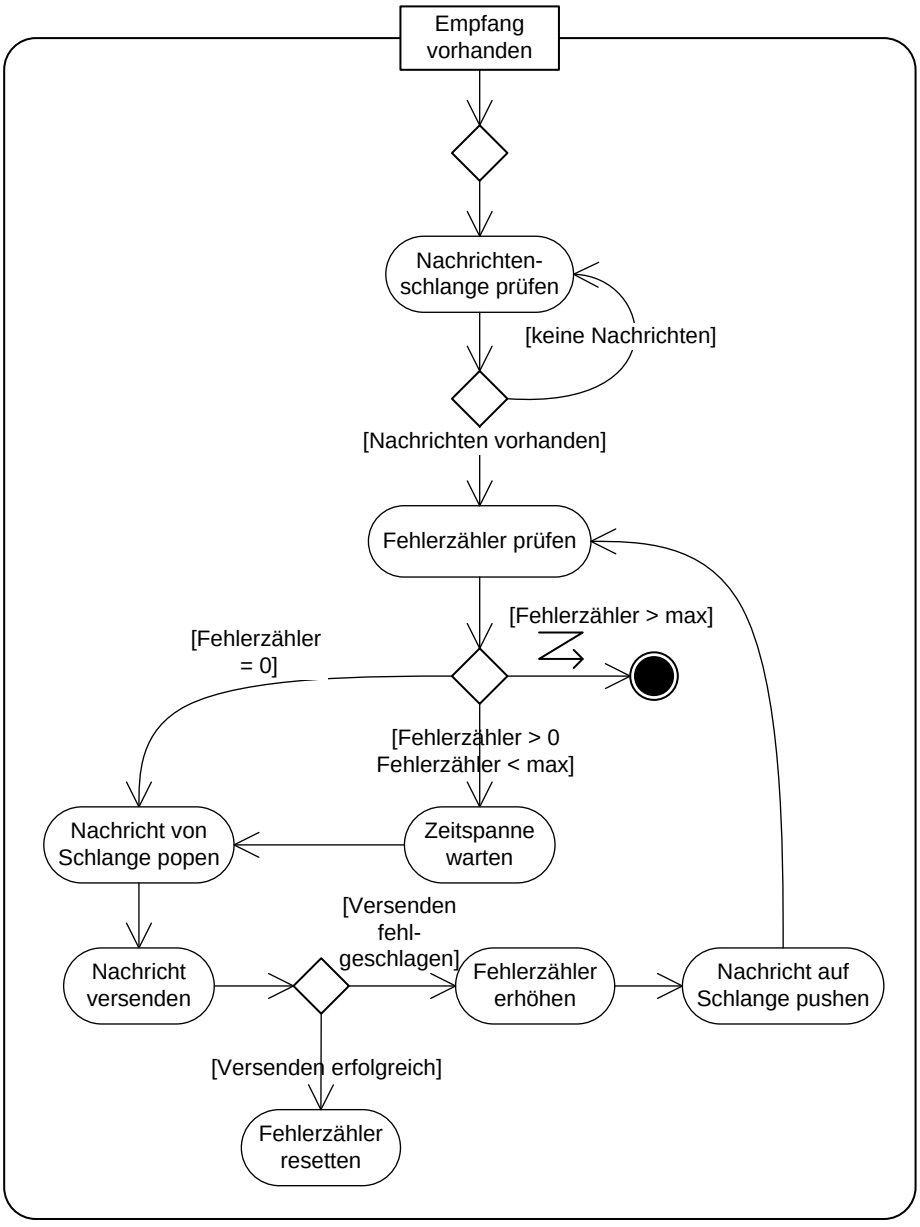


Abbildung 5.19: Ablaufdiagramm von Nachricht-Versenden Teil 2

5.3.2.2 Serverseite

Die Serverseite nimmt Nachrichten vom Client entgegen und leitet sie an die Serviceschicht weiter. Die Kommunikationsschicht stellt darüber hinaus der Serviceschicht Schnittstellen zur Verfügung, um Nachrichten mittels des RIM eigenen Push-Dienstes an die BlackBerry-Geräte versenden zu können. Die serverseitige Middleware läuft innerhalb eines Applikationsservers, der neben der Entgegennahme von HTTP-Anfragen auch eine JVM bereitstellt, innerhalb der alle Funktionen abgearbeitet werden.

Vom Client eintreffende Nachrichten Der Kommunikationsteil auf Serverseite, der Nachrichten vom BlackBerry-Gerät entgegennimmt, ist die Klasse *BBConnectorServlet*. Sie ist als Servlet aufgebaut. Ein Servlet ist eine Java-Klasse, die innerhalb eines Applikationsservers läuft und HTTP-Anfragen entgegennimmt. Hierzu erweitert es die Klasse *HttpServlet* und implementiert die Prozeduren *doGet()* und *doPost()*. Diese Klasse nimmt die HTTP-Post Anfrage des Clients entgegen und dekomprimiert die empfangene Nachricht. Anschließend wird die Nachricht an den Thread *PostWebStub* übergeben und dem Client die erfolgreiche Übermittlung mitgeteilt. Die Mitteilung geschieht über die Standard-HTTP-Statuscodes (RFC 2616RFC u. a. (1999)). Da diese Klasse ein Servlet darstellt, kann sie, wie eine normale Webseite auch, mehrere Anfragen gleichzeitig entgegen nehmen: Die Anzahl gleichzeitiger Verbindungen ist abhängig von den Einstellungen des Applikationsservers, innerhalb dem das Servlet ausgeführt wird.

Die Klasse *PostWebStub* ruft innerhalb ihrer *run()*-Methode die Schnittstelle *newIncomingMessage()* der Klasse *ContentHandler* aus der Serviceschicht auf und übergibt dieser das empfangene Bytearray.

Push-Nachrichten verschicken Um Nachrichten an ein BlackBerry-Gerät versenden zu können, implementiert *MsgPush* die Prozedur *newOutgoingMessage()* des Interface *ProxyInterface*. Diese Schnittstelle erwartet neben der Nachricht die GUID (Global Unique ID) des BlackBerrys, an den die Nachricht versendet werden soll. Anschließend wird die Nachricht der Klasse *PAPPush2* übergeben, welche einen Push-Auftrag generiert. Dann wird die Nachricht an den MDS des Unternehmens geschickt. Dieser leitet die Nachricht dann an die Server von RIM weiter, welche sie über den richtigen Mobilfunkanbieter

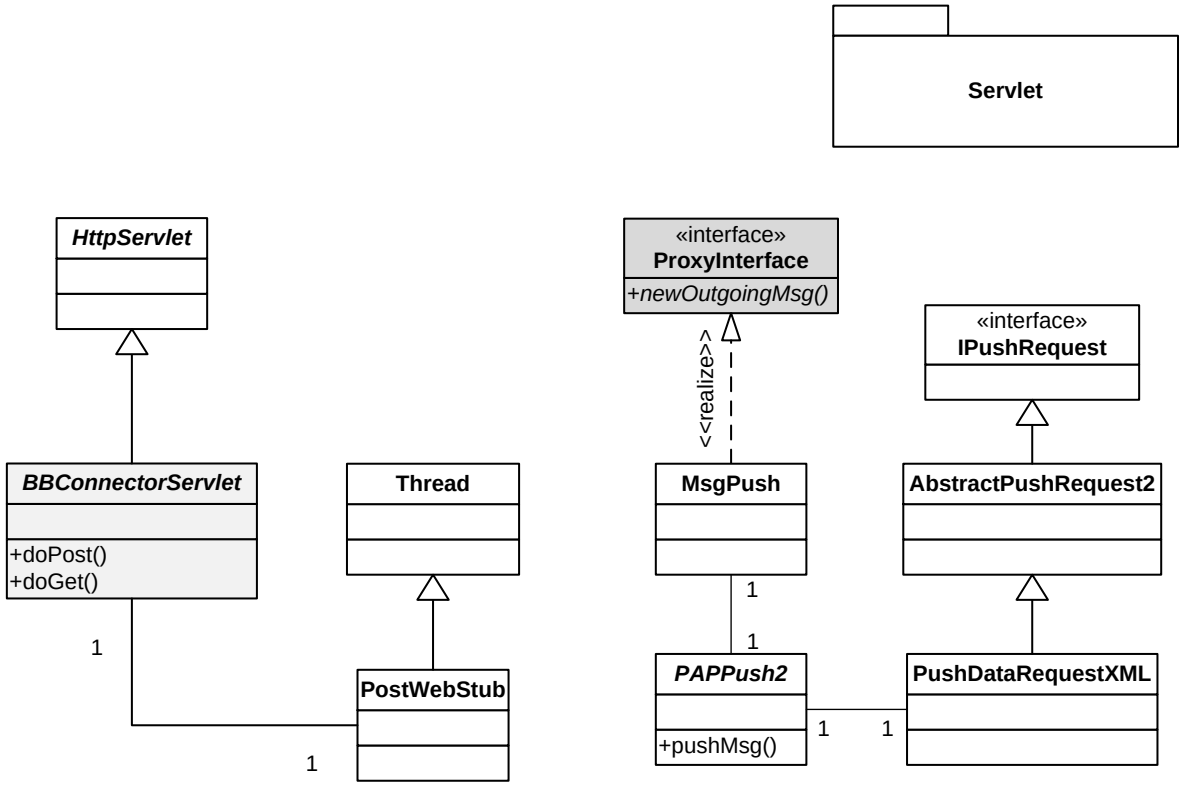


Abbildung 5.20: Klassendiagramm des Servlets

dann an das BlackBerry-Gerät senden, beziehungsweise pushen (weiterführende Informationen, siehe Kapitel 2.4: *BlackBerry*). Abbildung 5.20 stellt das Klassendiagramm der serverseitigen Kommunikationsschicht vereinfacht dar.

Die eigentliche Push-Nachricht Die Push-Nachricht, die generiert und dem MDS übermittelt wird, ist eine proprietäre Funktion des BlackBerry-Netzwerkes. Es gibt innerhalb diesem zwei verschiedene Push-Arten:

- RIM-Push. Ein veralteter Push-Auftrag für Geräte vor Firmware-Version 4.0.
- PAP-Push. Ein Push-Auftrag, der in der Art eines XML-Dokuments aufgebaut ist. Er sollte für alle neueren Geräte verwendet werden (ab Firmware 4.0)

```
1 --$(boundary)
2 Content-Type: application/xml; charset=UTF-8
```

```
3
4  <?xml version="1.0"?>
5  <!DOCTYPE pap PUBLIC "-//WAPFORUM//DTD PAP 2.0//EN"
6    "http://www.wapforum.org/DTD/pap_2.0.dtd"
7    [<?wap-pap-ver supported-versions="2.0"?>]>
8  <pap>
9  <push-message push-id="$(pushid)" ppg-notify-requested-to="$(
       notifyURL)">
10
11 <address address-value="WAPPUSH=$(pin)%3A100/TYPE=USER@rim.net"/>
12 <quality-of-service delivery-method="$(deliveryMethod)"/>
13 </push-message>
14 </pap>
15 --$(boundary)
16 $(headers)
17
18 $(content)
19 --$(boundary)--
```

Listing 5.7: XML-artiger Aufbau eines PAP-Pushs

PAP-Pushs werden in der Datenbank des jeweiligen MDS-Server zwischengespeichert, bis eine Auslieferung erfolgreich war. So kann der sichere Empfang der Nachricht auf dem BlackBerry gewährleistet werden. Außerdem sind die Mitteilungen so vor Ausfällen der Server geschützt. Die Datenbanken der MDS können jedoch maximal 1000 Push-Anfragen für ein Gerät vorhalten. Listing 5.7 zeigt den strukturellen Aufbau eines PAP-Pushs. In Zeile 18 käme die Nachricht der Serviceschicht, also der gesamte Inhalt des Bytearrays.

5.3.3 Sicherheitsaspekte

Für mehr Datensicherheit gibt es mehrere Möglichkeiten der zusätzlichen Verschlüsselung. Erstens ist es an der Stelle, an der der Bytestrom auf Korrektheit überprüft wird, möglich, eine zusätzliche Verschlüsselung einzubauen. Zweitens ist es durch die Verwendung des HTTP-Protokolls auch möglich, gleich eine HTTPS-Verbindung aufzubauen. Hierfür ist es lediglich notwendig, ein gemeinsames Zertifikat auszutauschen. Allerdings ist diese zusätzliche Verschlüsselung vorerst unnötig, da die Daten bei den BlackBerry-Geräten standardmässig verschlüsselt werden. Notwendig wird eine zusätzliche Verschlüsselung erst, sollte der Middleware-Server sich außerhalb der Unternehmensfirewall befin-

den. Dann wäre es theoretisch möglich, eine Man-In-The-Middle Attacke zu starten, indem man den unverschlüsselten Datenverkehr zwischen dem MDS (2.4) und dem Servlet der Middleware belauscht.

6 Realisierung

Nachdem der Entwurf für die Middleware im vorherigen Kapitel erläutert wurde, wird hier auf die eigentliche Realisierung eingegangen. Hierzu werden die beiden Schichten wieder getrennt behandelt.

6.1 Schnittstellen für die Anwendungen

In der Realisierung wird nicht auf alle Teile eingegangen, die im Entwurf beschrieben sind. Es wird hauptsächlich auf entscheidende Stellen im Code oder auf Implementierungen, die besondere APIs des BlackBerrys nutzen, eingegangen. Der gezeigte Code entspricht meist nicht dem, der im eigentlichen Programm zum Einsatz kommt. Er wurde für das bessere Verständnis und die Übersichtlichkeit vereinfacht. Es werden nur einzelne Methoden und Klassen vorgestellt, und auf Fehlerbehandlung wird größtenteils verzichtet. An Stellen, an denen „[...]“ im Code zu finden ist, sind sich wiederholende Blöcke oder ganze Funktionen ausgeblendet worden. Client und Server werden wieder getrennt behandelt. Abläufe, die auf Client- und Serverseite ähnlich sind, wie zum Beispiel das Kapseln der Inhalte, werden nur auf einer Seite beschrieben.

6.1.1 Implementierungen auf Clientseite

Auf der Clientseite sind die wichtigsten Komponenten die Interprozesskommunikation, die Verwaltung der verschiedenen Anwendungen und das Kapseln der verschiedenen Inhalte in den Header. Diese werden jeweils in den folgenden Unterabschnitten beschrieben.

6.1.1.1 Interprozesskommunikation

Die Interprozesskommunikation auf einem BlackBerry ist relativ einfach zu lösen. Hierzu existiert die Klasse *net.rim.device.api.system.RuntimeStore*. In dieser RIM-proprietären Klasse können Referenzen abgelegt werden, die dann auf dem BlackBerry global zur Verfügung stehen. Anhand eindeutiger *runtimeIDs* kann auf diese zugegriffen werden. Die Referenzen lassen sich dann von Anwendungen so nutzen, als wären es lokale Objekte. Damit nicht jede Anwendung auf jedes Runtime-Objekt Zugriff hat (zum Beispiel BlackBerry interne Objekte), existieren zusätzlich Sicherheitsmechanismen. Diese werden hier allerdings nicht weiter behandelt. Listing 6.1 zeigt, wie der Adapter mittels Runtime Objekten auf den Service zugreift.

```
1  public final class BBConnectorMsgHandler implements
       MsgHandlerRunetimeInterface {
2
3      private static final long SERVICE_ID = 0xa4a14bcc88b8d664L;
4    private ServiceRuntimeInterface _service;
5    private String _applicationID;
6
7      public BBConnectorMsgHandler(BBConnectorIF ownerApp) throws
         BBMsgHandlerException {
8          _ownerApp = ownerApp;
9          _applicationID = ApplicationDescriptor.
             currentApplicationDescriptor().getName();
10         _service = getServiceRIF(SERVICE_ID);
11         _service.login(_applicationID, this);
12     }
13
14   private ServiceRuntimeInterface getServiceRIF(final long
       runetimeID) throws ServiceRIFException{
15         // try to get the instance of ServiceRuntimeInterface
             from the RuntimeStore
16         RuntimeStore runtimeStore = RuntimeStore.getRuntimeStore
             ();
17         Object o = null;
18
19         synchronized(runtimeStore){
20             o = runtimeStore.get(runetimeID);
21         }
22         // check that the object is in the store, and it is of
             the right type
```

```
23          if (o instanceof ServiceRuntimeInterface){
24        // if it is cast the object
25              return (ServiceRuntimeInterface)o;
26          } else {
27              throw(new ServiceRIFException());
28          }
29      }
30    [...]
31  }
```

Listing 6.1: Auszug MsgHander: Interprozesskommunikation

Beim Erstellen des Adapters (*BBConnectorMsgHandler*) wird im Konstruktor eine Referenz auf den Service erstellt (Zeile 10). Hierzu wird die eindeutige RuntimeID (Zeile 4) benutzt. Danach wird mit Hilfe der Referenz die *login* Funktion des Services aufgerufen (Zeile 11). Hier gibt der Adapter auch eine Referenz auf sich selbst mit. Diese wird später vom Service benutzt wird, um auf den Adapter zuzugreifen. Da auf den Service so leicht mehrere Anwendungen gleichzeitig zugreifen können, erlaubt die Schnittstelle *ServiceRuntimeInterface* nur synchronisierte Zugriffe auf den Service. Somit kann immer nur eine Anwendung zur gleichen Zeit auf den Service zugreifen. In der Funktion *getServiceRIF* (Zeile 15) sind die einzelnen Abläufe dargestellt, um eine Referenz für die Interprozesskommunikation zu erhalten.

6.1.1.2 Anwendungsregister und persistente Speicherung

Im Anwendungsregister wird persistent gespeichert, welche Anwendungen auf dem Gerät den Service nutzen. Zusätzlich werden die Referenzen auf die Anwendungen gesichert, die für die Interprozesskommunikation nötig sind. Diese Referenz wird dem Service beim Login einer Anwendungen übergeben (siehe Listing 6.1, Zeile 11). Für die persistente Speicherung wird ebenfalls eine RIM proprietäre Klasse genutzt. Die *net.rim.device.api.system.PersistentStore* Klasse funktioniert, ähnlich wie die *RuntimeStore* Klasse, über eindeutige IDs. Objekte, die die Schnittstelle *net.rim.vm.Persistable* implementieren, können so persistent gespeichert und, mit Hilfe der ID, auch wieder gelesen werden. Listing 6.2 zeigt das persistente Speichern.

```
1 public class ApplicationRegister {
2
```

```
3     private PersistentObject _store;
4     private Hashtable _appRegister;
5     private Vector _persAppID;
6
7     public ApplicationRegister(long persObjectID) throws
          ApplicationRegisterException {
8         _persAppID = getContent(persObjectID);
9         _appRegister = new Hashtable();
10    }
11
12    private Vector getContent(long persObjectID) throws
          ApplicationRegisterException {
13        Object o = null;
14        //retrieve a reference to a PersistentObject
15        _store = PersistentStore.getPersistentObject(persObjectID
              );
16        synchronized (_store) {
17      //If store is empty, create Vector and put into the store
18            if (_store.getContents() == null) {
19                _store.setContents(new Vector());
20                _store.commit();
21            }
22            //retrieve the contents of the PersistentObject
23            o = _store.getContents();
24        }
25        if (o instanceof Vector){
26             return (Vector)o;
27        } else {
28            throw(new ApplicationRegisterException());
29      return null:
30        }
31    }
32
33  private void storeRegister(){
34        synchronized(_store) {
35            _store.setContents(_persAppID);
36            _store.commit();
37        }
38    }
```

Listing 6.2: Anwendungsregister: persistentes Speichern

Beim Erstellen des Anwendungsregisters (*ApplicationRegister*) wird die ID des gespeicherten Objektes übergeben (Zeile 7). Mit der Funktion *getContent* wird der persistente

Speicher ausgelesen und das Objekt zurückgegeben. Zusätzlich wird ein *Hashtable* erstellt. Dies dient dazu, die Referenzen, die für die Interprozesskommunikation mit den Anwendungen nötig sind, zu speichern (siehe Listing 6.3). Die Funktion *storeRegister* (Zeile 34) wird immer dann aufgerufen, wenn sich der Vektor geändert hat. Sie speichert dessen Zustand persistent.

```
38      public void login(String applicationID,
           MsgHandlerRunetimeInterface msgHandler) {
39          // If application is not registered at login
40          if (!isRegistered(applicationID)){
41              _persAppID.addElement(applicationID);
42              storeRegister();
43          }
44          _appRegister.put(applicationID, msgHandler);
45      }
46
47      public MsgHandlerRunetimeInterface getMsgHandler(String
           applicationID) throws NoSuchApplicationException,
           InternalStoreException {
48          Object o = null;
49          //Check if appID is valid
50      if(!isRegistered(applicationID))
51              throw(new NoSuchApplicationException());
52          if(!_appRegister.containsKey(applicationID))
53              return null;
54      //Get RuntimeInterface
55          o = _appRegister.get(applicationID);
56      //Is Object OK?
57          if(!(o instanceof MsgHandlerRunetimeInterface))
58              throw(new InternalStoreException());
59
60          return (MsgHandlerRunetimeInterface) o;
61      }
62    [...]
63  }
```

Listing 6.3: Anwendungsregister: Referenzen verwalten

In Listing 6.3 ist die *login* Funktion des Anwendungsregisters zu sehen (Zeile 38). Der Login, der in Listing 6.1 vom Adapter ausgeführt wurde, wird vom Service an das Anwendungsregister weitergegeben. Hier wird die Anwendung registriert und die Referenz zu den Anwendungen im Hashtable (Zeile 44) gespeichert. Die Funktion *getMsg-*

Handler wird vom Service benutzt, um sich anhand der AnwendungsID die Referenz, die für die Interprozesskommunikation im Hashtable gespeichert wurde, zu beschaffen.

6.1.1.3 Kapseln der Inhalte

Das Kapseln der Inhalte erledigt die Klasse *Message* selbst. Für die Konvertierung in XML wird die Klasse *XMLBuilder* verwendet. Sie kapselt den XML-Parser. Somit kann der Parser, ohne die restliche Funktionalität zu beeinträchtigen, durch einen anderen ersetzt werden. Der Vorgang in der *Message*-Klasse ist in Listing 6.4 dargestellt.

```
1  public class Message implements Persistable{
2
3    private int _msgID;
4      private String _blackberryPIN;
5      [...]
6      private int _contentType;
7      private Content _content;
8
9      public Message(Content content) {
10         //MsgAttribute setzten PIN, MsgID, ...
11         init();
12         setContent(content);
13     }
14
15     public Message(byte[] newMsg) throws IOException,
          ContentCreatorException {
16         fromXML(newMsg);
17     }
18
19   public byte[] toXML() throws ParserConfigurationException{
20
21     XMLBuilder xml = new XMLBuilder();
22     //MsgAttributes
23     xml.addElement(XMLBuilder.ROOT, MESSAGE);
24     xml.addAttribute(MESSAGE, MESSAGE_ID, _msgID);
25     xml.addElement(MESSAGE, BLACKBERRY_PIN, _blackberryPIN);
26     [...]
27     //Content
28     xml.addElement(MESSAGE, CONTENT);
29     xml.addAttribute(CONTENT, CONTENT_TYPE, _contentType);
```

```
30      xml.setContentRoot(CONTENT);
31      _content.toXML(xml);
32
33      return xml.toByte();
34    }
35    [...]
36  }
```

Listing 6.4: Message Klasse auf Clientseite

Beim Erzeugen des *Message* Objekts wird der *Content* mit übergeben (Zeile 9). Alternativ kann das Message Objekt auch aus einem *byte-Array* wiederhergestellt werden (Zeile 15). Dies dient dazu, aus ankommenden XML-Nachrichten wieder ein Message-Objekt zu machen. Dieser Vorgang wird später auf Serverseite gezeigt. Die Methode *toXML* dagegen wandelt das Objekt in ein byte-Array, in dem das Objekt im XML-Format abgelegt ist. Hierzu werden die einzelnen Attribute dem XMLBuilder übergeben (Zeile 21-26). Dann wird das Content-Tag angelegt (Zeile 28), das Typ-Attribut gesetzt (Zeile 29) und dem XMLBuilder bekannt gemacht, dass ab jetzt der Content übergeben wird (Zeile 30). Dieser wird seriallisiert, indem wieder die entsprechende *toXML* Methode aufgerufen wird (Zeile 31). Dieser Methode wird der präparierte XMLBuilder übergeben. In Listing 6.5 wird der weitere Verlauf im Content Objekt gezeigt.

```
1  public final class ConfigureProxyIP extends ConfigContent {
2
3      private static String IP = "ProxyIP";
4      private String _proxyIP;
5
6      public ConfigureProxyIP(int contentType){
7          super(contentType);
8      }
9
10   //Override abstract method from Content
11     public void toXML(XMLBuilder xml) throws
           ParserConfigurationException {
12          xml.addElement(XMLBuilder.CONTENT_ROOT, IP, _proxyIP);
13     }
14   //Override abstract method from ConfigContent
15     public void configure(ServicePropertiesIF serviceProperties)
           {
16          serviceProperties.setProxyIP(_proxyIP);
17          serviceProperties.configChanged();
```

```
18      }
19    [...]
20  }
```

Listing 6.5: Content Klasse auf Clientseite

Nachdem die abstrakte *toXML* Methode der Klasse Content von dem *Message* Objekt aufgerufen wurde, wird jetzt in die toXML-Methode des eigentlichen Typs gesprungen. In diesem Fall ein *ConfigureProxyIP*-Typ. Hier werden die entsprechenden Attribute dem *XMLBuilder* übergeben (Zeile 12). Normalerweise werden Konfigurationsinhalte nicht an den Server geschickt, aber an diesem Beispiel kann so gleichzeitig noch gezeigt werden, wie der Service durch die Konfigurationsinhalte angepasst wird. Dies geschieht in der Methode *configure* (Zeile 15). Diese wird vom Service aufgerufen, wenn der Inhalt als solcher erkannt wird. Dieser Methode wird ein Objekt übergeben, das Zugriff auf Konfigurationen des Services ermöglicht. Somit kann hier die Adresse des Proxy Servers angepasst werden (Zeile 16). Danach werden mit der Methode *configChanged* des Konfigurations-Objektes alle Listener, die bei der Konfiguration angemeldet sind, über die Änderungen informiert (Zeile 17).

6.1.2 Implementierungen auf Serverseite

Auf der Serverseite wird auf das Entpacken und Ausführen der einzelnen Inhalte eingegangen. Hierfür sind eine Reihe von Abläufen nötig, die in den folgenden Unterkapiteln vorgestellt werden.

6.1.2.1 Verarbeiten der eingehenden Nachrichten

Die Nachrichten, die von der Kommunikationsschicht übergeben werden, müssen entsprechend ihres Inhalts-Typs verarbeitet werden. Dies geschieht in der Klasse *ContentHandler*. Der Ablauf ist in Listing 6.6 dargestellt.

```
1 public class ContentHandler {
2     //Interface to Sub-Layer
3     ProxyInterface _pap;
4
5     public ContentHandler(ProxyInterface pap) {
```

```
6           _pap = pap;
7       }
8
9       public void newIncomingMessage(final byte[] newMsg) throws
            PAPException, ContentHandlerException,
            ContentCreatorException {
10      //Restore Msg-Object
11          Message msg = new Message(newMsg);
12      //Check Message
13      if(!checkBlackberryID(msg.getBlackberryPIN()))
14        throw new ContentHandlerException("Content Handler: No such
              blackberry device registered.");
15      [...]
16      //Start  actions in Content
17      msg.getContent().start();
18      //Send answer
19      _pap.newOutgoingMessage(msg.toByte(), msg.getBlackberryPIN())
            ;
20      }
21    [...]
22  }
```

Listing 6.6: Verarbeitung auf Serverseite

Jede eingehende Nachricht erzeugt ein *ContentHandler*-Objekt. Dieses Objekt bekommt von der Kommunikationsschicht eine Schnittstelle übergeben, mit der es die Antworten senden kann (Zeile 5). Danach wird von der Kommunikationsschicht die Methode *newIncomingMessage* aufgerufen (Zeile 9). Dieser wird der eingehende byte-Array übergeben, der die XML-Nachricht enthält. Diese Nachricht wird wieder in ein Message-Objekt gewandelt (Zeile 11). Wie dies genau funktioniert wird in den nächsten Unterkapiteln beschrieben. Zusätzlich können mit der Nachricht einige Kontrollen gemacht und Statistiken erstellt werden. Ein Beispiel dafür ist das Kontrollieren der BlackBerry PIN (Zeile 13, 14). Danach werden die speziellen Aktionen des Inhalts ausgeführt. Hierzu wird die Methode *start* des *Contents* aufgerufen. Dies ist wieder eine abstrakte Methode und somit von Typ zu Typ verschieden. Genaueres hierzu im nächsten Unterkapitel. Zum Schluss wird hier noch die Antwort mit dem modifizierten Inhalt zurückgeschickt (Zeile 19).

6.1.2.2 Inhalte wiederherstellen

Um aus dem byte-Array wieder eine Nachricht zu bekommen, ist eine Instanz nötig, die aus der Inhalts-Typen wieder das richtige Content-Objekt macht. Dies erledigt die Klasse *ContentCreator*, welche in Listing 6.7 aufgeführt ist.

```
1  public abstract class ContentCreator {
2
3      //Registered Content Types for Applications
4      public static final int SQL_QUERY = 102;
5      public static final int CEBIT_DEMO = 103;
6      //Registered Content Types for configuration
7      public static final int CONF_PROXY = 501;
8    [...]
9
10     //Creates and returns Content on basis of the choosen type
11     static public Content createContent(int contentType) throws
           ContentCreatorException {
12
13         switch (contentType){
14             //Applications
15             case WSSOAP:
16                 return new SOAPContent(WSSOAP);
17             case SQL_QUERY:
18                 return new SQLQueryContent(SQL_QUERY);
19             case CEBIT_DEMO:
20                 return new CEBITDemoContent(CEBIT_DEMO);
21             //Configuration
22             case CONF_PROXY:
23                return new ConfigureProxyIP(CONF_PROXY);
24       [...]
25             //No such content
26             default:
27                 throw new ContentCreatorException();
28         }
29     }
30 }
```

Listing 6.7: Inhalte wiederherstellen auf Serverseite

Sie enthält nur die statische Methode *createContent* (Zeile 11). Diese gibt, je nach übergebener ID, den richtige Content-Typ zurück. Außerdem enthält sie eine Auflistung aller

vorhandenen Content-Typen. Dies sind Variablen, die der Einfachheit halber die ID ersetzen können. Außerdem lassen sich die IDs so an einer zentralen Stelle verwalten. Diese Klassen wird vom Message Objekt benutzt, um den Content wieder herzustellen. Dieser Vorgang ist in Listing 6.8 dargestellt.

```
1  public class Message{
2    //Attributes
3      private int _msgID;
4      private String _blackberryPIN;
5      [...]
6      private int _contentType;
7      private Content _content;
8
9      public Message(byte[] newMsg) throws IOException,
          ContentCreatorException, MessageException {
10         fromByte(newMsg);
11     }
12   //Restores the attributes; Called from: fromBytes()
13     private void fromXML(XMLReader xml) throws MessageException{
14
15     _msgID = Integer.parseInt(xml.getAttribute(MESSAGE,
          MESSAGE_ID));
16     _blackberryPIN = xml.getElement(BLACKBERRY_PIN)
17     [...]
18     _contentType = Integer.parseInt(xml.getAttribute(CONTENT,
          CONTENT_TYPE));
19     //Restores Content on Basis of the type
20     _content = ContentCreator.createContent(_contentType).fromXML
          (xml);
21     }
22   [...]
23 }
```

Listing 6.8: Message Klasse Serverseite

Nachdem der Konstruktor der *Message*-Klasse aufgerufen wurde (Zeile 9), wird das byte-Array einem *XMLReader* übergeben. Die Methode fromXML stellt dann mit Hilfe dieses XMLReaders die einzelnen Attribute des Message-Objekts wieder her. Der Content wird mit Hilfe des ContentCreators und der entsprechenden Typen-ID wieder hergestellt und die entsprechende fromXML Methode aufgerufen (Zeile 20). Die Attribute des Message-Objektes und des entsprechend gekapselten Contents sind so wieder

verfügbar.

6.1.2.3 Abarbeiten der Inhalte

Die auf Serverseite auszuführenden Aktionen werden in den entsprechenden *Content*-Klassen definiert. Hierzu wird die abstrakte *start()* Methode der Klasse Content überschrieben. Zusätzlich muss die Methode fromXML überschrieben werden, um die Attribute entsprechend wieder herzustellen. Ein Beispiel hierfür zeigt Listing 6.9.

```
1  public class SQLQueryContent extends Content {
2      //Attributes
3      private String _sUsr;
4      private String _sPwd;
5      [...]
6      private String _result;
7      //Construktor; Called by ContentCreator
8      public SQLQueryContent(int contentType) {
9          super(contentType);
10     }
11   //Actions; Overrides the abstract method from Content
12     public void start() {
13
14         DatabaseProcessor dbProz = new DatabaseProcessor(_sUsr,
               _sPwd, _driver, _connUrl);
15         _result = dbProz.getTableContent(_query);
16     }
17   //Restores the attributes; Overrides the abstract method from
        Content
18     public Content fromXML(XMLReader xml) {
19
20         _sUsr = xml.getElement(USER);
21         _sPwd = xml.getElement(PWD);
22         [...]
23         return this;
24     }
25   [...]
26 }
```

Listing 6.9: Content: Datenbankabfrage

Dieser Content-Typ führt eine SQL-Abfrage auf eine relationale Datenbank aus. In Zeile 8 ist der Konstruktor zu sehen, der vom ContentCreator aufgerufen wird. In der

start()-Methode (Zeile 12) wird dann die eigentliche Abfrage ausgeführt. Dies geschieht in einer separaten Klasse. Diese *start()*-Methode wird vom ContentHandler, der in Unterabschnitt 6.1.2.1 gezeigt wurde, ausgeführt. Die *fromXML*-Methode (Zeile 18) wird zur wieder Herstellung der Attribute von der kapselnden *Message*-Klasse aufgerufen (siehe Listing 6.8).

6.2 Kommunikation innerhalb der Middleware

Diese Schicht bietet eine begrenzte Anzahl an öffentlichen Prozeduren, mit deren Hilfe Nachrichten entgegen genommen oder Einstellung zur Übertragung vorgenommen werden können. Dadurch, dass diese Kommunikationsschicht von der darüber liegen Serviceschicht eingebunden wird, stehen der oberen Schicht diese öffentlichen Prozeduren zur Verfügung. Außer möglicherweise auftretenden Fehlermeldungen erhält die obere Schicht keinerlei Rückmeldung über die Anfragen, die sie gestellt hat. In umgekehrter Richtung stellt die Serviceschicht eine BlackBox für die Kommunikationsschicht dar. Deswegen wurden Interfaces geschaffen, die die Serviceschicht implementieren muss. So kann sichergestellt werden, dass die untere Ebene der Middleware auch mit der darüber liegenden Ebene kommunizieren kann. Notwendig ist dies, da eintreffende Nachrichten an die Serviceschicht - und von dort zur eigentlichen Anwendung - weitergeleitet werden können müssen.

6.2.1 Implementierung auf Clienteite

6.2.1.1 Überblick

Die Transmitterklasse stellt den Einsprungpunkt für die darüber liegende Klasse zur Verfügung. Dazu startet die *BBTransmitter*-Klasse zwei weitere Threadklassen, die für das Versenden und Empfangen von Nachrichten zuständig sind: *ThQueueReader* für das Versenden und *PAPReceiver* für das Empfangen. Eine genaue Beschreibung des Zusammenspiels der Klassen und ihr Aufbau ist im Folgenden beschrieben.

6.2.1.2 Implementation

Wie beschrieben muss diese Schicht auf der BlackBerry-Seite mindestens zwei Prozeduren/Funktionen implementieren. Erstens: Entgegennehmen von Nachrichten von der darüber liegen Schicht und Nachrichten vom Server. Zweitens: Vorhandene Nachrichten an den Server oder die obere Schicht weiterleiten.

Entgegennahme von Nachrichten von der darüber liegenden Schicht Alle zu sendenden Nachrichten werden unabhängig vom Inhalt, als Bytestrom an die Kommunikationsschicht geliefert. Dazu implementiert die Transmitterklasse die Prozedur *newOutgoingMessage(byte[] msg) {...}*. Sie erstellt eine neue Instanz der Klasse *ThQueueWriter* und übergibt dieser den Bytestrom. *ThQueueWriter* schreibt den Bytestrom in die Warteschlange und beendet sich danach selbst. Die Klasse ist als eigener Thread ausgelegt, so dass es der oberen Schicht möglich ist, viele Nachrichten in kurzem Abstand abzusetzen, die alle sicher verarbeitet werden.

Versenden von Nachrichten Das Versenden von Nachrichten übernimmt die Klasse *ThQueueReader*. Ebenfalls als Thread ausgelegt, wird sie einmalig gestartet und wartet zum einen auf eine Verbindung zum Carrier und auf Nachrichten in der Warteschlange. Vorher, beziehungsweise beim Fehler einer der beiden Bedingungen, pausiert der Thread, um nicht unnötige Ressourcen des Systems zu verbrauchen. Wenn sich der BlackBerry im Empfangsbereich des Dienstanbieters befindet, löst das ein Ereignis im Betriebssystem aus. Dieses wird abgefragt und startet den Thread *ThQueueReader*. Ein weiteres Ereignis wird beim Verlust des Carriers ausgelöst und sorgt dafür, dass der Thread erneut pausiert. Sobald der Thread läuft, ruft er die Funktion *receiveFQ()* der Warteschlange auf. Dort wird die erste Nachricht aus diesem FIFO gelesen und weiterverarbeitet und letztlich versendet. Sollten keine Nachrichten bereit liegen, wird der Thread mit einem *Wait()* durch die Warteschlange belegt. Sobald von der schreibenden Klasse *ThQueueWriter* Daten in die Warteschlange geschrieben wurden, wird ein *notifyAll()*-Ereignis ausgelöst und der *ThQueueReader* beginnt die Daten auszulesen.

Das Versenden des Bytearrays an den Server ist in der Prozedur *postViaHttpConnection()* der *ThQueueReader*-Klasse implementiert. In Listing 6.10 ist zu sehen, dass diese Prozedur das *HTTPConnection*-Objekt der Java ME benutzt. Es wird eine HTTP-POST-Verbindung zu einer gegebenen URL aufgebaut und ein Inputstream und ein Outputstream geöffnet. Der Outputstream stellt dabei eine unidirektionale Verbindung zur Serverkomponente der Kommunikationsschicht her und versendet das Bytearray der Nachricht. Vorher wird dieses noch im GZIP-Format gepackt, um das Datenvolumen zu verringern. Das *flush()* nach dem Schreiben des Bytestroms auf den Outputstream dient dazu, den gesamten Inhalt aus einem eventuellen Puffer zu schreiben. Der zusätz-

lich geöffnete Inputstream nimmt die Antwort des Servers entgegen und wertet diese aus. Zwar liefert die Funktion *getResponseCode()* des *HTTPConnection*-Objekts die Antwort des Servers zurück, allerdings werden nur die Antworttypen erkannt, das heißt, ob eine Sendung grundsätzlich empfangen wurde oder nicht. Zum Beispiel kann der Antworttyp „OK“ verschiedene Bedeutungen haben, die man anhand der mitgelieferten Statuscodes erkennen kann:

- 200. Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.
- 202. Die Anfrage wurde akzeptiert, wird aber zu einem späteren Zeitpunkt ausgeführt. Das Gelingen der Anfrage kann nicht garantiert werden.
- 203. Die Anfrage wurde bearbeitet, das Ergebnis ist aber nicht unbedingt vollständig und aktuell.

Darüber hinaus gibt es noch weitere Statuscodes, nachzulesen im RFC 2616RFC u. a. (1999). Um diese zurückgelieferten Statuscodes auszuwerten, wird der Inputstream benötigt. Anhand dieser Nummer wird letztlich entschieden, ob die Nachricht erfolgreich versendet wurde oder ob sie bis zu einem weiteren Übertragungsversuch zurück in die Warteschlange gesichert wird.

```
1  private void postViaHttpConnection(String url, byte[] msg) throws
       Exception {
2    HttpConnection conn = null;
3    OutputStream os = null;
4    InputStream is = null;
5    try {
6      conn = (HttpConnection)Connector.open(url);
7      // Requestmethode und headers
8      conn.setRequestMethod(HttpConnection.POST);
9      _compression = gzip.getCompression();
10     if (_compression > 0) {
11       //conn.setRequestProperty("Content-Encoding", "gzip");
12       msg = gzip.compress(msg);
13     }
14     // Getting the output stream may flush the headers
15     os = conn.openOutputStream();
16     os.write( msg );
17     os.flush();
```

```
18
19      res = conn.getResponseCode();
20      if (res != HttpConnection.HTTP_OK)
21        throw new Exception(BBTransmitterException.QUEUE_READER);
              // Server-Verbindung fehlgeschlagen
22      else {
23        // Antwort verarbeiten
24        is = conn.openInputStream();
25        response = new ProxyResponse();
26        int answer = response.getProxyRespone(is);
27  [...]
28    } catch (Exception e) {
29      throw new BBTransmitterException(e.toString());
30    } finally {
31  [...]
32    }
33  }
```

Listing 6.10: Prozedur postViaHttpConnection()

Sollte ein Übertragungsfehler aufgetreten sein, wird neben der Zurückspeicherung der Nachricht in den FIFO auch ein Fehlerzähler erhöht. Dieser sorgt dafür, dass nach einer bestimmten Anzahl von Fehlversuchen eine bestimmte Zeitspanne gewartet wird, bevor ein erneuter Übertragungsversuch unternommen wird. Diese Wartezeit wird immer größer, je mehr Fehlversuche unternommen wurden, bis hin zu einer maximalen Zeitspanne. Die möglichen Versuche und die daraus resultierende Wartezeit können frei definiert werden. Sobald eine erfolgreiche Übertragung zustande kam, wird der Fehlerzähler wieder zurückgesetzt. Dieses Zurücksetzen ist auch von außen möglich, zum Beispiel wenn eine Nachricht empfangen wurde (mehr siehe Paragraph „Empfangen von Nachrichten vom Server“).

Warteschlange Sie ist das zentrale Verbindungsstück innerhalb der Kommunikationsschicht. Implementiert ist sie als FIFO-Speicher.

Während eines schreibenden Zugriffs ist die Warteschlange für weitere Schreibzugriffe gesperrt. Erreicht wird dies durch den *Synchronized*-Befehl von Java wie in Zeile 48, beziehungsweise 63 zu sehen (Listing 6.11). Er verhindert, dass ein derart eingeschlossener Code gleichzeitig von mehreren Threads ausgeführt wird. Insbesondere bei

Schreibanweisungen ist dies nötig, da nur so sicher Dateninkonsistenz verhindert werden kann.

```
48 synchronized public void putFQ( byte[] msg ) {
49   // Prüfen ob noch Platz in Queue (fifoQueue.length entspricht
       der Gröÿe aus new byte[size][] )
50   while(qSize >= fifoQueue.length) {
51     // gibt den LOCK(synchronized) des Objekts wieder frei
52     wait();
53   }
54   // Nachricht hinten anfügen
55   fifoQueue[tail] = new byte[msg.length];
56   for(int i = 0; i < msg.length; i++) {
57     fifoQueue[tail][i] = msg[i];
58   }
59 [...]
60   notifyAll();
61 }
62
63 synchronized public byte[] receiveFQ() {
64   while(qSize == 0) {
65     // gibt den LOCK(synchronized) des Objekts wieder frei
66     wait();
67   }
68   byte[] result = fifoQueue[head];
69 [...]
70   notifyAll();
71   return result;
72 }
```

Listing 6.11: Auszug aus der Warteschlangen-Klasse

Um Datenverluste bei Ausfall des Geräts zu vermeiden, wird die Warteschlange nach dem Beschreiben oder Auslesen von Daten persistent gespeichert. Dazu dient die Klasse *PersistentStorage*, die das *PersistentObject* verwendet. Das *PersistentObject* ist eine Implementierung von RIM und erlaubt das Speichern von Objekten auf dem BlackBerry. Es ist schneller als ein gewöhnliches RecordStore-Objekt, das durch Java ME zur Verfügung gestellt wird, funktioniert aber ähnlich wie eine Datenbank. Außerdem ist es in Ermangelung eines Dateisystems auf den BlackBerry-Geräten die einzige Möglichkeit Daten persistent zu speichern.

Statuslistener Der Ereignis-Listener reagiert auf Änderungen der Empfangsqualität des Mobilfunknetzes. Er implementiert die Klasse *RadioStatusListener* von RIM und startet, falls der BlackBerry sich im Empfangsbereich des Mobilfunkanbieters befindet, den *ThQueueReader*-Thread, um Daten zu übertragen. Falls plötzlich kein Empfang mehr vorhanden ist, wird der Sende-Thread durch eine Prozedur informiert, worauf dieser pausiert. Dies ist nötig, da Threads nicht von außen gestoppt werden sollten, um Daten-Inkonsistenz zu vermeiden. Die Prozedur innerhalb des *ThQueueReader*, beendet eine Schleife, in der die *run()*-Methode des Threads eingebettet ist. Dadurch kann die Abarbeitung noch beendet werden. Die genaue Implementierung ist in Listing 6.12 dargestellt. Die Klasse selbst ist innerhalb des *BBConnector* eingebettet.

```
119 private class ConnectionListener implements RadioStatusListener {
120   public void networkStateChange(int state) {
121     // Verbindungstest
122     switch (state) {
123     case GPRSInfo.GPRS_STATE_READY:
124       // Carrier vorhanden, senden
125       thsender.setConnection(true);
126       if (!thsender.isAlive()) {
127         try {
128           thsender.start();
129         }
130         catch (Exception e) { ... }
131       }
132       break;
133 [...]
134     case GPRSInfo.GPRS_STATE_IDLE:
135       // kein Carrier
136       thsender.setConnection(false);
137       break;
138     }
139   }
140 }
```

Listing 6.12: Listener-Klasse

Daten packen und entpacken Wie bereits angesprochen, nutzen sowohl die *ThQueueReader*-Klasse als auch die *PAPReceiver*-Klasse eine Pack- beziehungsweise Entpack-

routine. Die Funktionen der Klasse, die diese Routinen bereit stellen, bekommen dazu Bytearray übergeben und packen oder entpacken dieses. Das Ergebnis wird wiederum als Bytearray zurückgeliefert. Als Kompressionstyp wird GZIP verwendet. Die Umsetzung der Pack- und Entpackroutinen mit Hilfe der GZIP-Implementierung ist im Listing 6.13 zu sehen. Dazu wird ein *ByteArrayOutputStream* geöffnet und auf diesen der Inhalt des Bytearrays geschrieben: Entweder durch Entpacken der Daten mit Hilfe des *GZIPInput-Stream*-Objekts von RIM oder durch Packen der Daten durch *GZIPOutputStream*. Dieselbe Implementierung ist auch auf Serverseite zu finden, um auch dort mit gepackten Daten umgehen zu können und Nachrichten komprimieren zu können.

```
1  public byte[] compress(byte[] content) {
2    try {
3      ByteArrayOutputStream bout = new ByteArrayOutputStream();
4      GZIPOutputStream gzos = new GZIPOutputStream (bout,
           COMPRESS_RATE);
5      gzos.write( content );
6      gzos.close();
7      return bout.toByteArray();
8    } catch (IOException ex) {
9      return null;
10   }
11 }
12 [...]
13 public byte[] decompress(byte[] content) {
14   try {
15     ByteArrayOutputStream bout = new ByteArrayOutputStream();
16     ByteArrayInputStream bis = new ByteArrayInputStream( content
           );
17     GZIPInputStream gzis = new GZIPInputStream( bis );
18     int len;
19     for (; (len = gzis.read()) != -1;) {
20       bout.write(len);
21     }
22     return bout.toByteArray();
23   } catch (IOException ex) {
24     return null;
25   }
26 }
```

Listing 6.13: Auszug aus der Pack-Entpack-Klasse

Empfangen von Nachrichten vom Server Die Klasse *PAPReceiver* übernimmt das Empfangen von Nachrichten vom Server und die Übergabe der Nachrichten an die darüber liegende Schicht. Dazu ist die Klasse ebenfalls als Thread ausgelegt. Sobald die Middleware gestartet wird, sorgt die Hauptklasse *BBTransmitter* dafür, dass auch dieser Thread gestartet wird. Um Nachrichten vom Server entgegen nehmen zu können, wird der Port 100 des BlackBerrys geöffnet. Um empfangene Nachrichten weiterleiten zu können, bekommt der Konstruktor der Klasse einen Zeiger auf die weiterverarbeitende Klasse innerhalb der Serviceschicht übergeben (siehe Schnittstellen).

In Zeile 58 des Listings 6.14 wird die Funktion *acceptAndOpen()* des *StreamConnectionNotifier*-Objektes ausgeführt. Diese blockiert die weitere Abarbeitung des Threads so lange, bis am geöffneten Port Daten empfangen werden. Der Thread „lauscht“ somit auf ankommende Daten. Dadurch kann sichergestellt werden, dass Ressourcen nicht unnötig verschwendet werden, aber dennoch keine eintreffenden Nachrichten verloren gehen. Das *StreamConnectionNotifier*-Objekte ist Teil der Java-Implementierung von Java ME und ist als Interface für eine Socketkommunikation ausgelegt. Eine Socketkommunikation ist eine einfache TCP-Verbindung zwischen definierten Endpunkten eines Systems[1]. Aus diesem Grund wird der Datenstrom, wenn Nachrichten empfangen werden, in einen HTTP-Strom gecastet, um ihn weiterverarbeiten zu können. Wie bereits in Paragraph „Versenden von Nachrichten“ beschrieben, können dadurch weitergehende Informationen ausgetauscht werden. Wurde der Datenstrom vollständig empfangen, wird er in ein Bytearray gewandelt und entpackt. Anschließend wird eine Erfolgsmeldung zurückgegeben und die Verbindung beendet. Die so empfangene Nachricht wird dann an die nächsthöhere Schicht weitergeleitet. Dazu wird wieder ein Thread gestartet, der diese Schnittstellenkommunikation vornimmt, um nicht die restliche Ausführung der *PAPReceiver*-Klasse zu beeinflussen. Nachdem alle Verbindungen wieder getrennt wurden, wird der *PAPReceiver*-Thread aufgerufen, damit dieser die Möglichkeit hat, ebenfalls Daten zu versenden. Dies geschieht, da der Thread infolge zu vieler Fehlversuche einen längeren Zeitraum pausiert. Mit der Benachrichtigung soll erreicht werden, dass diese Pausezeit beendet wird, da eine gesicherte Erreichbarkeit des Servers gegeben ist. Nach der gesamten Abarbeitung kehrt der Empfangsthread wieder in Bereitschaft zurück und wartet auf eintreffende Nachrichten.

[1] Schütte (2006b)

```
try {
  synchronized(this) {
    _notify = (StreamConnectionNotifier)Connector.open(URL + ";
       deviceside=false");
  }
  while (!_stop) {
    //NOTE: the following will block until data is received
    stream = _notify.acceptAndOpen();
    try {
      input = stream.openInputStream();
      pushInputStream = new MDSPushInputStream((
         HttpServerConnection)stream, input);
      ByteArrayOutputStream bout = new ByteArrayOutputStream();
      int len;
      for (int j = 0; (len = input.read()) != -1;) {
        bout.write(len);
      }
      byte[] buffer = bout.toByteArray();
      _compression = gzip.getCompression();
      if (_compression > 0) {
        buffer = gzip.decompress( buffer );
      }
      //This method is called to accept the push
      pushInputStream.accept();

      parent.reRunQueueReader();
    } catch (IOException e1) {...}

    } finally {...}
  }
  ...
} catch (IOException ioe) {...}
```

Listing 6.14: Auszug aus der Klasse PAPReceiver

6.2.2 Implementierung auf Serverseite

6.2.2.1 Überblick

Das Servlet *BBConnectorServlet* ist die Hauptklasse der Kommunikationsschicht auf dem Server für eintreffende Anfragen vom Client. Sie nimmt einen eintreffenden Inputstream entgegen und wertet diesen aus. Außerdem werden hier alle relevanten Konfigu-

rationsparameter aus einer XML-Datei gelesen und die entsprechenden Einstellungen vorgenommen.

6.2.2.2 Implementation

Nachricht vom Client empfangen Wie in Unterkapitel 5.3.2.2: *Entwurf und Design: Server* beschrieben ist ein Java-Servlet für den Empfang der Nachrichten vom Client zuständig. HTTP-Post Nachrichten werden vom Applikationsserver an die *doPost()*-Methoden der entsprechenden Servlets weitergeleitet. Von dort werden der eintreffende und der ausgehende Datenstrom an die Prozedur *processRequest()* übergeben. Diese öffnet beide Datenströme und schreibt zum einen die empfangenen Daten in ein Bytearray, zum anderen schickt sie einen Statuscode zum Client zurück (siehe Listing 6.15). Zuerst entpackt das Servlet das empfangene Bytearray mittels GZIP und übergibt es an die Thread-Klasse *PostWebStub*, die die Weiterleitung der Nachricht an die Serviceschicht übernimmt. In keiner der genannten Klassen findet eine Überprüfung auf korrekten Inhalt der Nachricht oder einen berechtigten Sender statt, da hierzu der Nachrichteninhalt selbst überprüft werden muss.

```
42 protected void processRequest(HttpServletRequest request,
       HttpServletResponse response) throws ServletException,
       IOException {
43   InputStream input = null;
44   GZipper gzip = new GZipper(COMPRESS_DATA);
45   response.setContentType("text/html;charset=UTF-8");
46   PrintWriter out = response.getWriter();
47   if( request.getMethod().equals("POST") ) {
48     input = request.getInputStream();
49     try {
50       // Read a set of characters from the socket
51       ByteArrayOutputStream bout = new ByteArrayOutputStream();
52       int len;
53       for (int j = 0; (len = input.read()) != -1;) {
54         bout.write(len);
55       }
56       byte[] buffer = bout.toByteArray();
57       if (gzip.getCompression() > 0) {
58         buffer = gzip.decompress( buffer );
59       }
```

```
60        if (buffer.length > 0) {
61          new Thread(new PostWebstub( buffer, DEVICE_PORT, MDS_PORT
                , MDS_HOST, notifyURL, MDS_PAP_URI, MDS_PROTOCOL,
                COMPRESS_DATA));
62          out.write(answer_ok);
63        }
64        else {
65          out.write(answer_415);  // maybe i thought content was
                gziped but wasn't
66        }
67      } catch (Exception ex) {
68        out.write(answer_400);
69      }
70    } else {
71      out.write(answer_400);
72    }
73    out.flush();
74    out.close();
75  }
```

Listing 6.15: Servlet

Nachrichten an den Client pushen Die Kommunikation vom Server zum Client erfolgt nicht mittels einer HTTP-Verbindung. Zum Übermitteln von Daten wird der PAP-Push-Dienst der BlackBerry-Infrastruktur verwendet (näheres siehe die Kapitel 2.4: *BlackBerry* und 5.3.2.2: *Entwurf und Desing: Serverseite*). Die PAP-Push-Nachricht besteht aus einer XML-Struktur mit Informationen zum Empfänger sowie dem eigentlichen Nachrichteninhalt aus der Serviceschicht. Dieser Nachrichteninhalt in Form eines Bytearrays wird in die XML-Struktur des PAP-Pushs eingebettet und das gesamte Objekt an den MDS-Server des Unternehmens übermittelt, von wo die Nachricht letztlich auf das BlackBerry-Gerät gelangt.

Für die einfache Nutzung der Push-Funktionalität, existiert die Klasse *MsgPush*, die die Prozedur *newOutgoingMsg()* implementiert. Sie kann einfach von der Serviceschicht genutzt werden. Dazu bekommt die Serviceschicht einen Verweis auf MsgPush im Konstruktor übergeben, bevor die empfangenen Nachrichten an sie weitergeleitet werden. Das ist dadurch begründet, dass das Servlet beim Start Initialisierungsparameter aus einer XML-Datei ausliest und so das MsgPush direkt initialisiert (Listing 6.16). Dadurch entfällt eine aufwändige Konfiguration durch die Serviceschicht, und zum Generieren eines

Push reicht die Nachricht sowie die Geräte-PIN des BlackBerry.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java
   .sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<context-param>
  <param-name>DEVICE_PORT</param-name>
  <param-value>100</param-value>
</context-param>
<context-param>
  <param-name>MDS_PORT</param-name>
  <param-value>8080</param-value>
</context-param>
<context-param>
  <param-name>MDS_HOST</param-name>
  <param-value>da-dyma-bes-01.dymacon-int.de</param-value>
</context-param>
<context-param>
  <param-name>notifyURL</param-name>
  <param-value>http://127.0.0.1:9900</param-value>
</context-param>
<context-param>
  <param-name>MDS_PAP_URI</param-name>
  <param-value>/pap</param-value>
</context-param>
<context-param>
  <param-name>MDS_PROTOCOL</param-name>
  <param-value>http</param-value>
</context-param>
<context-param>
  <param-name>CHUNK_SIZE</param-name>
  <param-value>254</param-value>
</context-param>
<context-param>
  <param-name>COMPRESS_DATA</param-name>
  <param-value>9</param-value>
</context-param>
```

Listing 6.16: XML mit Einstellungen des Servlets

MsgPush leitet zu versendende Nachrichten an die Klasse *PAPPush2* weiter, welche die eigentliche Push-Nachricht generiert und versendet. Zuerst wird eine Verbindung

zum MDS-Server erstellt. Anschließend wird die Nachricht in die XML-Struktur des Push-Auftrages verpackt und an den MDS gesendet (Listing 6.17). Das Erstellen der XML-Push-Struktur geschieht mittels zweier Klassen: *PushDataRequestXML* und *AbstractPushRequest2*. In ihnen erfolgt zum einen der Aufbau der Struktur des XML-Dokuments, zum anderen werden alle für den Push benötigten Informationen eingefügt:

- pushId. Eine Zufallszahl, um den Push-Auftrag eindeutig zu kennzeichnen.
- DeliveryMethod.
  - CONFIRMED.
  - PRECONFIRMED.
  - UNCONFIRMED.
  - NOTSPECIFIED.
- bbID. Die PIN des Empfangsgerätes.
- PushRequestType. Die Art der Pushs:
  - DATAREQ. Der MDS soll Daten an einen BlackBerry weiter leiten.
  - CANCELREQ. Verwerfen eines bestehenden Auftrags.
  - STATUSREQ. Aktuellen Status eines BlackBerry abfragen.
- contentType. Der Typ der Daten im Rumpf der HTTP-Verbindung zum MDS[2]:
  - text/html
  - multipart/related
  - application/xml

[2] RFC u. a. (1999)

```
1  public synchronized void pushMsg(byte[] data, String bbID) throws
       IOException {
2    // Stream öffnen.
3    URL url = getPushURL();
4    URLConnection urlconn = url.openConnection();
5    HttpURLConnection conn = (HttpURLConnection)urlconn;
6    conn.setDoOutput(true);
7    conn.setRequestMethod("POST");
8    conn.setRequestProperty("Content-type", "multipart/related;
       type=\"application/xml\"; boundary=trenner");
9    conn.setRequestProperty(PushHeader.X_RIM_PUSH_TYPE.toString(),
       BrowserPushType.BROWSER_CONTENT.toString());
10   conn.setRequestProperty(PushHeader.X_RIM_PUSH_TITLE.toString(),
       "BBC_Message");
11   conn.setRequestProperty(PushHeader.X_WAP_PUSH_DESTINATION_PORT.
       toString(), Integer.toString(DEVICE_PORT));
12   OutputStream os = conn.getOutputStream();
13   ByteArrayOutputStream bout = new ByteArrayOutputStream();
14   // XML Header erzeugen
15   PushDataRequestXML request = new PushDataRequestXML();
16   int pushId = random.nextInt();
17   request.setPushId( String.valueOf(pushId) );
18   request.setDeliveryMethod(DeliveryMethod.UNCONFIRMED);
19   request.setBBPin(bbID);
20   // XML-Header auf Stream schreiben
21   bout.write( request.getXMLHeader() );
22   request.setContentType(ContentType.HTML);
23   // Push-Content anfügen
24   if (gzip.getCompression() > 0) {
25     data = gzip.compress(data);
26   }
27   // Content schreiben
28   bout.write( request.getContentHeader() );
29   bout.write( data );
30   bout.write( request.getContentFooter() );
31   // Gesamte Nachricht versenden
32   os.write(bout.toByteArray());
33   os.flush();
34 [...]
35   os.close();
36 }
```

Listing 6.17: Push-Request erstellen

6.3 Prototyp und Beispielanwendungen

Der Service auf der Clientseite ist durch eine eigene Anwendung realisiert. Diese wird beim Start des BlackBerry automatisch ausgeführt und läuft im Hintergrund. Zur Kontrolle der Konfiguration existiert eine graphische Oberfläche (siehe Abbildung 6.1). Hier kann der Service notfalls auch manuell beendet und neu gestartet werden. Konfiguriert werden kann der Service nur vom Server aus, Eine Konfiguration am BlackBerry selbst ist nicht möglich.

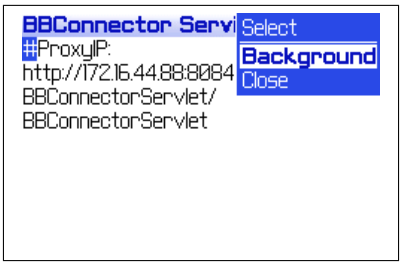


Abbildung 6.1: Anwendung zur Konfiguration der Middleware

Neben der eigentlichen Middleware wurden Prototypen zweier Beispielanwendung erstellt, die auf die Funktionalität der Middleware zurück greifen.

Prototyp 1 zur Demonstration auf der CeBIT 2007 Die Demo-Anwendung für die CeBIT 2007 besteht aus einer Anwendung auf dem BlackBerry-Gerät und einer Webseite mit Datenbankanbindung im Unternehmen. Mit ihr ist es möglich, Bestellungen abzuwickeln und diese Bestellungen in eine Datenbank schreiben zu lassen. Die Webseite dient der Darstellung aller bisher getätigten Bestellungen aus der Datenbank und kann auf Knopfdruck eine Rechnung im PDF-Format erstellen und anzeigen lassen. Alle Komponenten nutzen intern die erstellte Middleware und deren asynchrone und offlinefähige Übertragungseigenschaften. Zuerst kann der Nutzer seine persönlichen Daten wie Unternehmen, Name und E-Mail-Adresse angeben, danach wird ein weiterer Bildschirm dargestellt. In diesem werden Informationen zur gegenwärtigen Bestellung angezeigt. Dazu ist am BlackBerry eine Scanner- und Unterschriftenfeld-Kombination befestigt, die es erlaubt, Barcodes einzuscannen und Unterschriften zu leisten. Scannt der Nutzer einen Artikel, wird im internen Speicher der Anwendung nach dem entsprechenden Datensatz

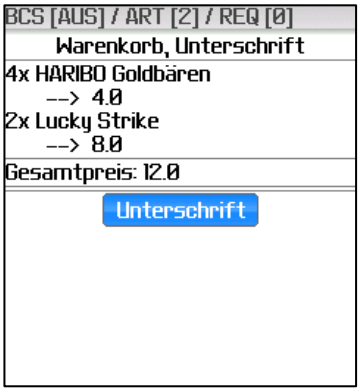
gesucht und die Informationen zu diesem angezeigt. Wird der Barcode nicht gefunden, wird eine Datenbankanfrage generiert, um diese Daten vom Server zu erhalten. Die Middleware übernimmt dann diese Anfrage und liefert die angeforderten Datenbestände vom Server an die Anwendung zurück. Hierfür ist serverseitig ein Modul integriert, das SQL-Anweisung verarbeiten kann. Sollte eine SQL-Anweisung Daten zurückliefern, so werden diese als CSV zurück geliefert.

Letztlich erhält der Nutzer alle Informationen zu dem gescannten Artikel angezeigt. Er hat dann die Wahl weitere Artikel in die Bestellung aufzunehmen oder den Bestellvorgang abzuschließen. Wird der Bestellvorgang abgeschlossen, wird der Anwender aufgefordert, seine Unterschrift auf dem Unterschriftenfeld zu leisten. Ist das geschehen, wird sie am Bildschirm dargestellt. Der Ablauf siehe Abbildungen 6.2, Bildschirm 1 bis 4. Daraufhin beginnt die eigentliche Bestellung: Das vom Scannerfeld zurückgelieferte Bitmap wird zerlegt, die im Bild markierten Pixel werden seriallisiert und zusammen mit den restlichen Informationen der Bestellung in eine XML-Struktur verpackt. Diese wird durch die Middleware an den Server übermittelt, der die Daten aus der XML-Struktur extrahiert und in die Datenbank sichert. Außerdem wird eine Bestätigungs-E-Mail an den Auftraggeber generiert und versendet.

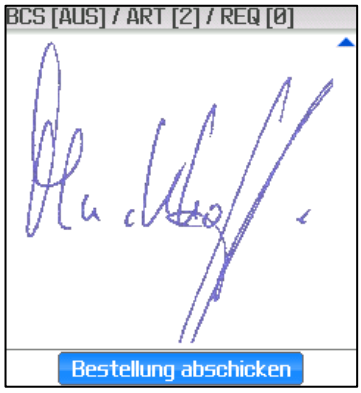
Eine Webseite auf Serverseite fragt den Inhalt der Datenbank regelmäßig ab und generiert dynamisch eine Seite, die den Inhalt darstellt. Auf dieser Seite wird zu jedem Datenbankeintrag eine Taste angezeigt, die es erlaubt, aus der Bestellung ein PDF zu generieren und anzeigen zu lassen (Abbildung 6.3). Dieses PDF enthält die bestellten Artikel samt deren Informationen und die geleistete Unterschrift.

1. Aufnehmen der Kundeninformatioenen

2. Scannen einzelner Artikel



3. Warenkorb kontrollieren und Unterschrift leisten



4. Bestellung abschicken

Abbildung 6.2: Cebit-Demo: Bestellvorgang auf dem BlackBerry

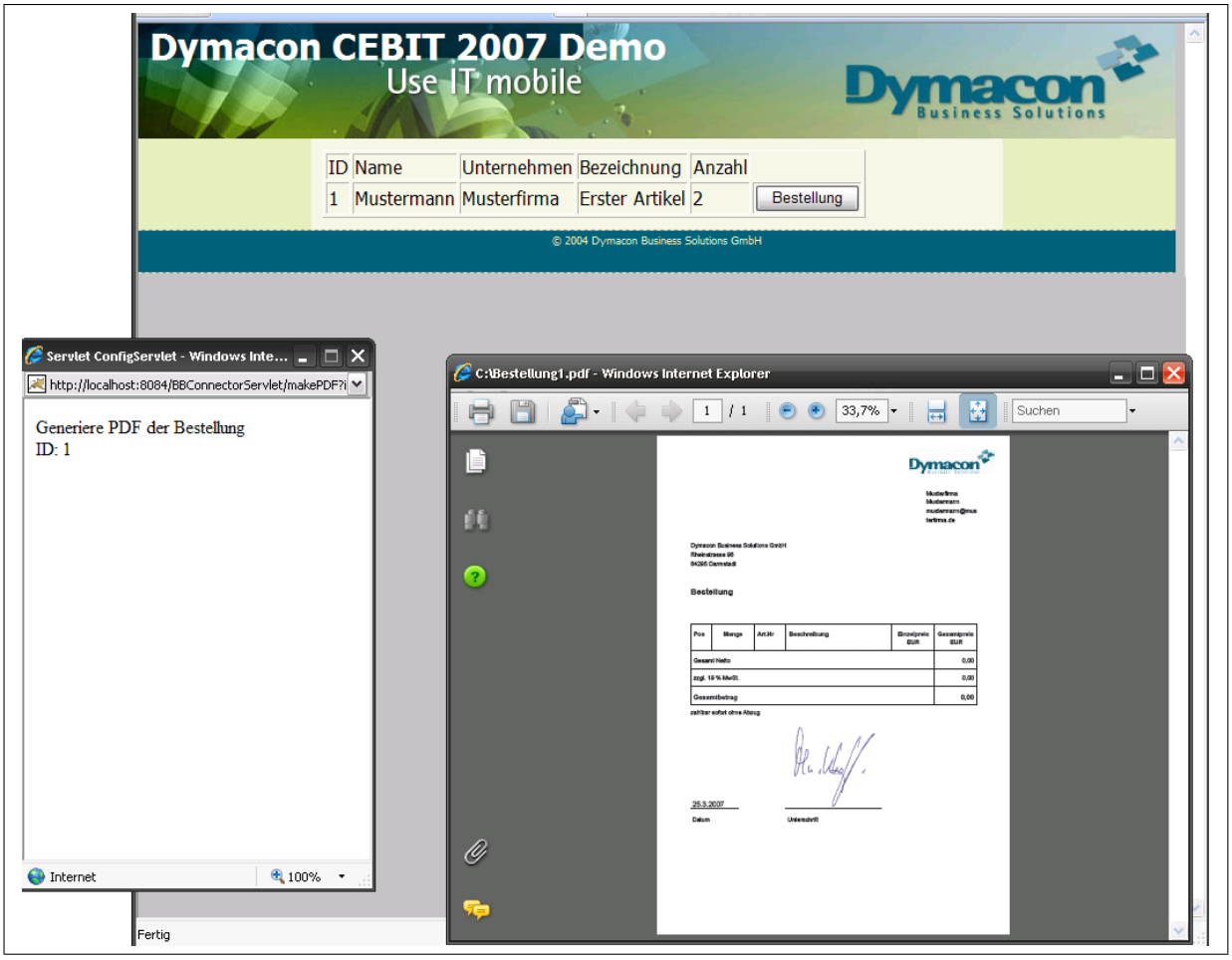


Abbildung 6.3: Webseite mit Informationen zu allen Bestellungen

Prototyp 2 als Kundenanwendung Die Kundenanwendung basiert ebenfalls auf der Middleware. Sie unterstützt den Kunden beim Lagerverkauf seiner Artikel. Mit einem Barcode-Scanner kann der Kunde seine Artikel scannen und bekommt die Artikeldaten auf dem BlackBerry angezeigt (siehe Abbildung 6.4).

Abbildung 6.4: Anzeigen eines vorher gescannten Artikels

Die Anwendung holt sich die Artikeldaten vom Server. Hierzu existiert serverseitig ein Modul, dass die entsprechenden Daten aus der Datenbank des Warenwirtschaftssystems abfragt, diese entsprechend aufbereitet und zum Client schickt. Die abgefragten Artikel werden zusätzlich auf dem Gerät gespeichert. Der Kunde kann wählen ob bei neuen Anfragen erst die lokale Datenbank auf dem Gerät abgefragt wird, oder die aktuellen Daten vom Server geholt werden sollen. Zusätzlich kann die lokale Datenbank komplett mit dem aktuellen Datenbestand synchronisiert werden. Abbildung 6.5 zeigt das entsprechende Kontextmenü hierzu.

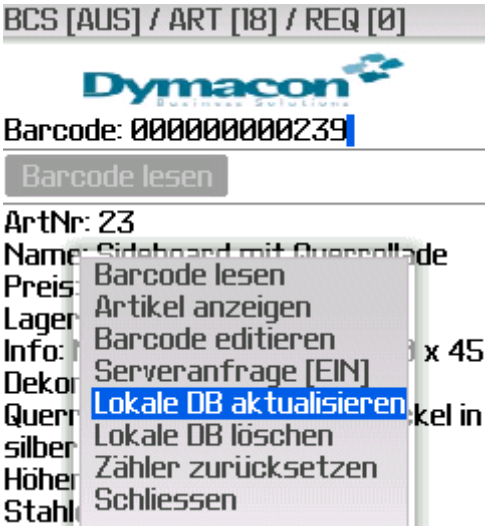


Abbildung 6.5: Kontextmenu zum Bedienen der Anwendung

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die vorliegende Arbeit beschreibt die Realisierung einer Middleware, um mobilen Anwendungen einen asynchronen Zugriff auf Informationen und Funktionen von Webanwendungen zu ermöglichen. Zuerst wurden Grundlagen für mobile Anwendungen erläutert, dann die Anforderungen an die Middleware spezifiziert: Die Möglichkeit, Nachrichten zum BlackBerry zu schicken, Offlinefähigkeit, modularer Aufbau, sichere Datenübertragung, OTA-Konfigurierbarkeit und geringes Datenvolumen. Als Plattform diente die BlackBerry-Umgebung des Unternehmens RIM.

In der Machbarkeitsstudie wurden verschiedene Lösungsansätze und -wege zur Realisierung der Middleware vorgestellt. Hierbei wurden aktuelle Komponentensysteme verglichen. Es wurde die Entscheidung getroffen, einen eigenständigen Service auf dem Gerät sowie eine auf Java EE Servlets basierende Serverlösung zu entwickeln. Anwendungen, die die Funktionen der Middleware nutzen wollen, kommunizieren über Schnittstellen mit dieser. Dazu implementieren diese ein Adapter, der das Senden und Empfangen von Nachrichten ermöglicht.

Um die Komplexität der Middleware zu reduzieren, wurde sie in zwei Schichten aufgeteilt, die getrennt voneinander entwickelt wurden: Eine Serviceschicht, die den Kontakt zu den Anwendungen bereitstellt, und eine Kommunikationsschicht, die für die Übertragung der Daten zwischen den BlackBerry-Mobilfunkgeräten und dem Server zuständig ist. Die Serviceschicht verwaltet alle angemeldeten Anwendungen und stellt diesen die Nachrichten zu, beziehungsweise nimmt Nachrichten von diesen entgegen. Die Nachrichten werden über ein auf XML-basierendes Protokoll verschickt. Die Kommunikationsschicht speichert die Sendeaufträge in einer Wartesschlange und verschickt diese, sobald sich der

BlackBerry im Empfangsbereich eines Mobilfunk-Betreibers befindet. Ein Thread innerhalb dieser Schicht lauscht ständig auf eintreffende Nachrichten und leitet diese an die Serviceschicht weiter. Auf Serverseite nimmt ein Servlet alle Nachrichten entgegen und verarbeitet sie. Nachrichtem vom Server werden durch das proprietäre Push-Protokoll von RIM an das entsprechende Empfangsgerät versendet. Zur korrekten Adressierung des Empfängers wird die eindeutige BlackBerry-ID verwendet.

Abschließend wurden zwei Anwendungen entwickelt, welche die Funktionalität der Middleware nutzen.

7.2 Bewertung der Arbeit

Die beiden erstellten Anwendungen rund um die Middleware, wurden auf der CeBIT 2007 vorgestellt. Die Unternehmen T-Online und Research in Motion stellten beide Produkte an ihren Ständen vor. Eine Anwendung befindet sich bereits im Kundeneinsatz. Die Konzeption der Middleware wurde im Rahmen einer Mobilfunktagung in Osnabrück als Fachartbeit eingereicht und als wissenschaftlicher Beitrag angenommen.

7.3 Ausblick

Für eine erweiterte Funktionalität der Middleware können weitere Eigenschaften realisiert werden:

- Die Funktionalität kann durch weitere Module erhöht werden. Z.B. könnte neben der vorhandenen Möglichkeit, Datenbanken zu beschreiben und abzufragen, auch der Zugriff auf ERP-Systeme ermöglicht werden.
- Die Funktionalität der Module kann in Java Enterprise Beans ausgelagert werden. So können diese auch relativ einfach von anderen Java EE Anwendungen benutzt werden.
- Die Datensicherheit kann durch die Nutzung von HTTPS erhöht werden.

- Möglichkeit des synchronen Datenaustauschs. Zur Übertragung großer Datenmengen könnte die geöffnete HTTP-Verbindung mehr, als nur die Statuscodes zurück liefern.
- Verbesserung der Fehlersemantik auf At-Most-Once. Der Server könnte Aufträge bzw. deren Checksummen speichern und Duplikate löschen. Die Zwischenspeicherung solcher Checksummen ist nur eine begrenzte Zeit nötig.

Literaturverzeichnis

[Apel u. Plack 2003] Apel, Sven ; Plack, Marco: *Überblick und Vergleich von Technologien zur Realisierung einer Middleware für mobile Informationssysteme.* Seminararbeit / Ausarbeitung, 2003. – Institut für Angewandte Informatik, Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg

[Balzert 1999] Balzert, Heide: *Lehrbuch der Objektmodellierung: Analyse und Entwurf.* Heidelberg / Berlin : Spektrum Akademischer Verlag GmbH, 1999

[BITKOM 2006] BITKOM: Daten zur Informationsgesellschaft 2006 / Bundesverband Informationswirtschaft Telekommunikation und neue Medien e.V. 2006. – Forschungsbericht

[Borchert 2003] Borchert, Dr. A.: Objektorientierte Datenbankanwendungen / Universität Ulm. 2003. – Forschungsbericht

[CMVP 2007] CMVP, Cryptographic Module Validation P. (Hrsg.): *Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules.* Version: 2007. http://csrc.nist.gov/cryptval/140-1/140val-all.htm. – [Online seit <unbekannt>; zuletzt besucht am 17.05.2007]

[Grothmann u. Rudat 2004] Grothmann, Heiko ; Rudat, Arsten: *Mobile Middleware.* Seminararbeit / Ausarbeitung, 2004. – Institut für Informatik LS X der Universität Dortmund

[Jeckle u. a. 2004] Jeckle, Mario ; Rupp, Chris ; Hahn, Jürgen ; Zengler, Barbara ; Queins, Stefan: *UML 2 glasklar.* München : Carl Hanser Verlag, 2004

[Kollmann 2007] Kollmann, Tobias: *E-Business.* Wiesbaden : Gabler Verlag/GWV Fachverlage GmbH, 2007

[Mandl 2005] Mandl, Prof. Dr. P.: Verteilte Systeme / Fachbereich Informatik/Mathematik an der Universität München. 2005. – Forschungsbericht

[Massoth 2006] Massoth, Prof. Dr. M.: Telekommunikation / Fachbereich Informatik der Hochschule Darmstadt. 2006. – Forschungsbericht

[Monson-Haefel 2004] Monson-Haefel, Richard: *J2EE Web Services*. Addison Wesley, 2004

[Otto 2003] Otto, Alexander: *Internet-Sicherheit für Einsteiger*. 1. Galileo Computing, 2003

[Research in Motion 2007] Research in Motion (Hrsg.): *BlackBerry-Sicherheit für Daten*. Version: 2007. http://www.blackberry.com/de/products/enterprisesolution/security/data.shtml. – [Online seit <unbekannt>; zuletzt besucht am 17.05.2007]

[RFC 1996] RFC: GZIP file format specification version 4.3 / Network Working Group. 1996. – Forschungsbericht

[RFC u. a. 1999] RFC ; Fielding, R. ; Irvine, UC ; Gettys, J. ; Compaq ; W3C ; MIT ; uva.: Hypertext Transfer Protocol – HTTP/1.1 / Network Working Group. 1999. – Forschungsbericht

[Schmatz 2004] Schmatz, Klaus-Dieter: *Java 2 Micro Edition*. 1. Dpunkt Verlag, 2004

[Schütte 2006a] Schütte, Prof. Dr. A.: Verteilte Systeme: CORBA / Fachbereich Informatik der Hochschule Darmstadt. 2006. – Forschungsbericht

[Schütte 2006b] Schütte, Prof. Dr. A.: Verteilte Systeme: Kommunikationsmodelle / Fachbereich Informatik der Hochschule Darmstadt. 2006. – Forschungsbericht

[Sichting 2004] Sichting, Helge: *Middleware-Architektur für mobile Informationssysteme*, Otto-von-Guericke-Universität Magdeburg, Diplomarbeit, 2004. – Institut für Technische und Betriebliche Informationssysteme, Fakultät für Informatik

[SUN 2007] SUN (Hrsg.): *Java ME Technology*. Version: 2007. http://java.sun.com/javame/technology/index.jsp. – [Online seit <unbekannt>; zuletzt besucht am 12.04.2007]

[Tanenbaum u. van Steen 2003] Tanenbaum, Andrew ; Steen, Marten van: *Verteilte Systeme*. Pearson Studium, 2003

[Teichmann u. Lehner 2002] Teichmann, Rene ; Lehner, Franz: *Mobile Commerce. Strategien, Geschäftsmodelle, Fallstudien*. Berlin : Springer, 2002

[Wichmann u. Stiehler 2004] Wichmann, Dr. T. ; Stiehler, Dr. A.: Prozesse Optimieren mit Mobile Solutions / Berlecon Research GmbH, Berlin. 2004. – Forschungsbericht

[Wikipedia Foundation 2007a] Wikipedia Foundation (Hrsg.): *BlackBerry*. Version: 2007. http://de.wikipedia.org/wiki/BlackBerry. – [Online seit 17.05.2007; zuletzt besucht am 17.05.2007]

[Wikipedia Foundation 2007b] Wikipedia Foundation (Hrsg.): *CSV*. Version: 2007. http://de.wikipedia.org/wiki/CSV-Datei. – [Online seit 10.05.2007; zuletzt besucht am 17.05.2007]

[Wikipedia Foundation 2007c] Wikipedia Foundation (Hrsg.): *FIPS 140-2*. Version: 2007. http://en.wikipedia.org/wiki/FIPS_140-2. – [Online seit 23.02.2007; zuletzt besucht am 17.05.2007]

[World Wide Web Consortium 2007] World Wide Web Consortium (Hrsg.): *XML*. Version: 2007. http://www.w3.org/. – [Online seit <unbekannt>; zuletzt besucht am 14.04.2007]

Abkürzungsverzeichnis

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
B2B	Business-to-Business
B2C	Business-to-Consumer
BES	BlackBerry Enterprise Server
CCM	CORBA Components Model
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
EDGE	Enhanced Data Rates for GSM Evolution
ERP	Enterprise Resource Planning
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HSDPA	High Speed Downlink Packet Access
Java EE	Java Platform, Enterprise Edition
Java ME	Java Platform, Micro Edition
Java SE	Java Platform, Standard Edition
JVM	Java Virtual Machine
KVM	Kilobyte Virtual Machine
MDS	BlackBerry Mobile Data System(TM)
MIDP	Mobile Information Device Profile
MMS	Multimedia Message System
PDA	Personal Digital Assistent
PIM	Personal Information Management

RFC ..... Request for Comments
RIM ..... Research in Motion
RMI ..... Remote Method Invocation
SMS ..... Short Message System
SOAP ..... Simple Object Access Protocoll
SUN ..... Sun Microsystems GmbH
UDDI ..... Universal Description, Discovery and Integration
UMTS ..... Universal Mobile Telecommunications System
URI ..... Unique Ressource Identifier
WSDL ..... Webservice Description Language