Einführung in Java Business Integration

Christoph Hartmann
Seminar Systemmodellierung 2006
Hasso-Plattner-Institut für Softwaresystemtechnik
christoph.hartmann@hpi.uni-potsdam.de

Abstract

In diesem Paper werden einige typische Probleme bei der Integration kurz erläutert. Dadurch können die Verbesserungen der Java Business Integration besser erkannt werden. Die Konzepte hinter JBI werden betrachtet, wobei im Besonderen auf die Architektur, Komponenten und die Kommunikation zwischen den Komponenten eingegangen wird. Für ein besseres Verständnis wird JBI an einem Beispiel angewandt und in Software für die praktische Nutzung eingeordnet. Grundkenntnisse in Service-orientierte Architektur werden dabei vorausgesetzt.

Keywords: *JBI, SOA, ESB, Web Services, XML*

1. Einführung

> There is no simple answer for enterprise integration [1]

Unternehmerische Applikationen werden heutzutage immer komplexer, umfangreicher und ständig weiterentwickelt. Meist steckt in ihnen das Wissen vieler Mannjahre. Software selbst kann nicht altern, doch ihre Umgebung ändert sich ständig. In einem Unternehmen entsteht somit die Anforderung seine Software oft auf den neuesten technischen Stand zu bringen. Gerade in aktueller Zeit ist ein Streit um die so genannte Service-orientierte Architektur (SOA) entfacht. Sie soll angeblich alle Probleme lösen und dafür sorgen, dass die Unternehmen *einfach* ihre vorhandenen Applikationen als Service bereitstellen und diese anschließend beliebig zusammen anwenden können. Nachdem Enterprise Application Integration (EAI) lange Zeit als der Hype galt, wurde mittlerweile erkannt, dass auch diese Lösung diverse Tücken hat. Die in EAI verwendete Message-orientierten Middlewares (MOM) hatte den Vorteil über einen zentralen Knoten zu kommunizieren und die Anzahl der Schnittstellen zu reduzieren. Doch je nach Hersteller war eine solchen Lösung an die herstellerbedingten proprietäre Formate gebunden. Ein Update auf ein System eines anderen Herstellers war daher nur schwer möglich und würde zu teueren Migrationskosten führen. Dass soll mittels SOA besser gelingen. Der in diesem Zuge entwickelte Enterprise Service Bus (ESB) steht für eine offene Infrastruktur. Allerdings führt ein Konzept nicht allein zu einer einheitlichen Struktur. Jeder Hersteller eines ESBs kann wiederum seine eigenen Interfaces und Schnittstellen schaffen. Damit dies nicht passiert, wurde im Java Community Process eine Spezifikation mit der Nummer 208 [9] ausgearbeitet. Diese behandelt die Integration von Anwendungen in einem ESB in der Sprache Java. Um diese Spezifikation näher beschreiben zu können, ist es allerdings notwendig, zuerst über Integration zu sprechen, sowie die Funktionsweise eines Enterprise Service Bus näher zu erläutern. Damit erlangt der Lesen einen Einblick in die Problemstellung und den späteren praktischen Nutzen.

[1] Enterprise Integration Patterns [2]

1.1. Integration

Bei einer Integration verschiedener Anwendungen treten nicht selten viele Probleme auf und die Komplexität des Themas wird meist unterschätzt. Integration darf nicht mit einer reinen verteilten Applikation verwechselt werden. Eine verteilte Applikation definierte eine klare Trennung in eine bestimmte Anzahl von Schichten. Die Kommunikation zwischen den Schichten erfolgt meist nur mit synchronen Aufrufen und anhand festgelegter Schnittstellen. Dagegen treten bei der Integration mehrerer *selbstständiger* Anwendungen wie unter anderem eines Bestellsystem und Mailsystem andere Probleme auf. Sie laufen unabhängig voneinander. Jede Applikation stellt seine benötigte Funktionalität bereit. Untereinander kommunizieren diese Applikationen meist asynchron und die gegenseitige Benutzung wird nur koordiniert. Eine direkte Benutzung durch einen menschlichen Benutzer ist dabei meist nicht vorgesehen. In der Realität verschwimmt die Unterscheidung zwischen reinen verteilten und integrierten Anwendungen zunehmend. Weiterhin erfordert die Integration klare Qualitätsanforderungen. Nach einer vom Kunden per Internet bestellte Ware darf nur

eine Mail als Bestellbestätigung abgeschickt werden. Bei diesem leicht klingenden Problem muss beachtet werden, dass die Software zur Warenbestellung meist nicht auf dem gleichen Rechner abläuft, wie das Mailsystem. Zwischen den System muss eine Netzwerkverbindung aufgebaut werden. Probleme entstehen, wenn die Netzwerkverbindung im Moment der Absendung nicht verfügbar ist. Zusätzliche Probleme bildet die Kommunikation der Systeme untereinander, wenn Sie in verschiedenen Programmiersprachen realisiert sind. Wird die Schnittstelle für das Mailsystem geändert, so muss das Bestellsystem daran angepasst werden. Die hier genannten Dinge sind nur einige Probleme von vielen, die bedacht werden müssen. Näheren Einblick bekommt der Leser in [2]. In der Vergangenheit haben sich dazu einige Lösungen herausgestellt, die in der Praxis häufiger Anwendung finden. Dazu gehören im einfachsten Fall der Dateiaustauch (mit einer klaren Spezifikation des Datenformats), eine gemeinsame Datenbank, Remote Procedure Calls oder eine Message Middleware. Aufgrund der stetig steigenden Anforderung nach simultanen Änderungen, ist eine Synchronisierung der Daten z.B. im halbstündlichen Takt meist nicht mehr möglich. Die Anwendungen sollen in Echtzeit ihre Änderungen publizieren. Daher setzen sich Implementierung von Middleware-Plattformen zur Nachrichtenübertragung immer mehr durch. Sie können meist asynchrone und synchrone Nachrichten übermitteln, speichern die Nachrichten bis sie garantiert angekommen sind und können auf Wunsch durch bestimmte Mechanismen garantieren, dass die Nachricht nur einmal versendet und empfangen wird.

1.2. Enterprise Service Bus

Als Weiterentwicklung der klassischen Enterprise Architecture Integration (EAI) Systemen kann der Enterprise Service Bus (ESB) [1] angesehen werden. Er arbeitet teilweise nach neuen Konzepten, profitiert aber aus den Erkenntnissen der vorhandenen EAI Systemen. EAI Systeme hatten einige typische Probleme wie die Verwendung von proprietären APIs, kein einheitliches Deploymentverhalten für Komponenten und eine Nachrichtenübermittlung über ein nicht offenem Format. Die Verwendung von herstellerspezifischen Nachrichten führte dazu, dass die Systeme nicht untereinander kompatibel waren. Die zentrale Kommunikationsstruktur (Hub-and-Spoke-Mechanismus) bei EAI Systemen führt zu erheblichen Problemen bei der Skalierung. In großen wachsenden Unternehmen muss eine wachsenden Datenmenge mit den System abbildbar sein. Sie benötigen unter anderem einen hohen Durchsatz von Daten.

Der Enterprise Service Bus ist eine Plattform, die zuverlässige Nachrichtenübermittlung, Datentransformierung und die Einbindung von Services ermöglichen soll.

Bei SOA wird davon ausgegangen, dass Services existieren, welche eine definierte Schnittstelle besitzen und über ein einheitliches Dienstverzeichnis gefunden werden können. Auf dieser Grundlage bauen die Implementierungen eines Enterprise Service Bus auf und versuchen dabei vor allem die Kommunikation zu erleichtern.

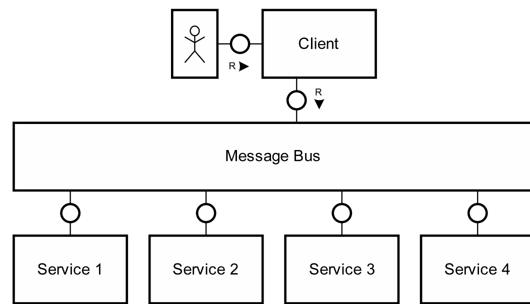

Abbildung 1. Aufbau Enterprise Serivce Bus

Der Unterschied gegenüber den EAI System liegt im Detail. Typischerweise verwenden ESBs standardisierte Schnittstellen zum Einbinden von Komponenten. Dazu gehören in Java J2CA [5] und JBI [9]. Die Services werden über klar spezifizierte Komponenten in einem ESB eingebunden und können somit nach der Einbindung in das Gesamtsystem über ein zentrales Dienstverzeichnis gefunden werden.

Konzeptionell lassen sich ESBs auch beliebig koppeln. Dies ist vergleichbar mit den heutigen Ethernet-Netzwerken. Zwei Switches für lokale Netzwerke können einfach mit einem Kabel verbunden werden und anschließend sofort miteinander kommunizieren.

Ein ESB umfasst die Funktionen

- Nachrichtenübermittlung
- Datenumwandlung (Transformation)
- Zuverlässiges Routing der Nachrichten
- Web Services [2]

und verbindet diese.

Die Qualitätsanforderungen an einen ESB sind hoch, da er zumeist in mittelständischen und großen Unternehmen Anwendung findet. Die Anforderungen umfassen daher eine garantierte Nachrichtenübermittlung und Sicherheit. Dabei soll die Übermittlung nicht irgendwann geschehen, sondern sofort und garantiert. Eine Plattform zur Integration muss daher die Qualität seiner Dienste gewährleisten können. Um dies zu können, sind unter anderem

[2]Die Benutzung von Web Services ist nicht vorgeschrieben, wird aber heutzutage meist in Verbindung mit dem Enterprise Service Bus genannt. Web Services haben den Vorteil eine programmiersprachlich unabhängige Schnittstelle in Form von WSDL [15, 17] bereitstellen. Um Probleme wie bei CORBA zu vermeiden, versucht die Industrie über die Web Service Interoperability Organization die Interoperabilität zu gewährleisten. Allerdings können auch andere Schnittstellenformate verwendet werden.

Transaktionen und die Zwischenspeicherung der Nachrichten wichtig. Die Nachrichten müssen ihren Status kennen und der ESB muss darauf geeignet reagieren können.

Eine direkte Nachrichtenübermittlung ist nicht immer möglich. Passen die Schnittstellen nicht direkt aufeinander, müssen die abgesendeten Daten vorher an das benötigte Format angepasst werden. Bei der Übertragung von XML Nachrichten geschieht dies meist auf Basis von XSLT [14]. Das Thema XML zur Nachrichtenübertragung wird im Abschnitt 11 näher erläutert.

Grundsätzlich muss klargestellt werden, dass das Konzept eines Enterprise Service Bus unabhängig von der Programmiersprache umgesetzt werden kann. Dennoch wird immer öfter mit Java geliebäugelt, da durch den Java Community Process viele wichtige notwendige Spezifikationen existieren. Für das Umsetzen eines Enterprise Service Bus existieren z.B. die Spezifikationen der Java Connector Architecture (J2CA) [5] , der Java Management Extension (JMX) [4] und der Java Business Integration (JBI) [9].

Eine zusätzliche wichtige Funktionalität ist die Komposition (Orchestration) von Services, daher die Nutzung von vorhanden Serivces in einem neuen Kontext. Dies geschieht in der Praxis häufig in Unternehmen, um die realen Betriebsprozesse z.B. mittels BPEL [11] umzusetzen.

Die Java Business Integration versucht nun die Probleme der Integration zu beseitigen. Die Spezifikation ist für die Umsetzung als ESB ausgelegt und versucht genau diese Funktionalität abzudecken. Im Folgenden wird näher auf die Java Business Integration (JBI) eingegangen.

2. Java Business Integration

JBI ist eine Spezifikation für Java, welche regelt, wie

- Komponentenaufbau (Packaging)
- Nachrichtenübermittlung (Message Routing)
- Komponentenlebenszyklus (Component Life Cycle)
- Komponentenverwaltung (Management)

auszusehen haben.

Gerade die Umsetzung dieser Punkte bildet die Möglichkeit aus einer Menge von heterogener Anwendungen eine inkrementelle Entwicklung zu ermöglichen. Es wird eine Kapselung der einzelnen Teile durchgeführt. Diese können separat entwickelt und nacheinander dem System hinzugefügt werden.

JBI wird es dabei nicht schaffen, die Probleme der Integration an sich stark zu vereinfachen, sondern eher sie verwaltbar zu machen. Eine JBI-Umsetzung stellt eine Infrastruktur für Komponenten bereit. Diese regelt, wie neue Services oder bestehenden Anwendungen in die JBI Umgebung eingebunden werden. Für die Komponenten wird klar geregelt, wie sie aufgebaut sein müssen, in das System eingebunden oder aus dem System entfernt werden können. Die Nachrichtenaustausch unter den Komponenten wird auf Basis von XML und WSDL 2.0 [17] geschehen.

Die wohl häufigsten Fragen in Verbindung mit JBI sind: Wozu JBI benötigt wird, wenn Web Services verwendet werden? Warum sollte noch einmal ein Java-Wrapper um die WSDL-Schnittstelle gebaut werden? Die Fragen müssen nach und nach beantwortet werden. Zuerst ist JBI eine Java-Spezifikation für die Umsetzung eines ESB in der Java-Welt. Dabei wird allerdings nicht vorausgesetzt, dass die bereitgestellten Services selbst in Java realisiert sind. Sie können in jeder beliebigen Programmiersprache umgesetzt werden. JBI selbst ist kein Wrapper, da die im XML erlangten Daten nicht noch einmal umgewandelt werden, sondern direkt als Payload in der normalisierten Nachricht gespeichert werden können. Bei der Anwendung von Web Services, können diese durch ihre vorliegende WSDL-Beschreibung einfach verwendet werden. Dazu regelt JBI hauptsächlich das Problem der einheitlichen Einbindung von zusätzlichen Services in das Gesamtsystem und die Registrierung an die JBI Implementierung. JBI regelt in diesem Fall nur die

- Einbindung an das System (hier meist Enterprise Service Bus),
- Verwaltet den Lebenszyklus
- Mapping der internen auf externen Endpunkte

Diese konkreten Nachrichten werden als XML Dokumente weitergegeben. Durch die Verwendung von WSDL als Schnittstellenbeschreibung empfiehlt es sich sogar, als Datenformat zum Austausch XML zu verwenden. Die eigentlichen Nachrichten müssen daher nicht umständlich umkonvertiert werden. Die Vor- und Nachteile von XML als Datenformat für den Nachrichtenaustausch werden im Abschnitt 11 näher erläutert.

Grundsätzlich macht es immer Sinn für eine JBI Komponente vorher ein Web Service zu erstellen, der die benötigten Informationen bereitstellt. Allerdings kann dies auch innerhalb der JBI Komponente realisiert werden, in dem Sie selbst die WSDL Beschreibung und die Implementierung mitliefert. Das macht hauptsächlich Sinn im Zusammenhang mit Legacy-Systemen.

3. JBI Komponenten

Im Folgenden sollen die in JBI verwendeten Komponentenarten näher erläutert werden. Eine JBI Komponente kann als ein Teil einer ganzen JBI Applikation angesehen werden, die ein klar definiertes Problem löst. Erst die JBI Applikation im Gesamten bildet dann die eigentliche

Anwendung. Diese Begrifflichkeiten sind für die Erstellung von Komponenten relevant, da sich je nach Anwendungsart auch die Art der Komponenten unterscheidet.

JBI unterscheidet grundlegend zwei Typen von JBI Komponenten

- Binding Component (BC)
- Service Engine (SE)

Zusätzlich zu den elementaren Komponentenarten gibt es noch zwei weitere:

- Shared Library
- Service Assembly

3.1. Binding Component (BC)

Eine Binding Component verbindet einen externen Service mit der JBI Infrastruktur. Sie konvertiert ein Nachrichtenprotokoll und benutzt somit externe Funktionalität. Eine Binding Component stellt selbst keine Applikationslogik zur Verfügung, welche über das Maß der Protokollumsetzung hinausgeht. Die Einhaltung dies Paradigmas liegt allerdings beim Entwickler und wird durch eine JBI Implementierung nicht kontrolliert.

Binding Components sind von der Anwendbarkeit vergleichbar mit J2CA [5]. Diese Protokolladapter müssen sich zusätzlich zum Protokoll auch mit Quality of Services (siehe 10)auseinander setzen. Typische Protokollumsetzungen sind SOAP, HTTP, FTP, SMTP, REST und ebMS. Allerdings sind hier noch viele weitere Protokolle denkbar.

3.2. Service Engine (SE)

Die Service Engine stellt im Gegensatz zur Binding Component selbst Anwendungslogik zur Verfügung. Diese Implementierungen sind meist allgemein gehalten und stellen Ausführungsumgebungen bereit. Typische Vertreter in diesem Bereich der Service Engine sind die XML Transformation (XSLT) oder die Business Process Execution Language (BPEL). Service Engines haben zur Binding Component die Möglichkeit sogenannte Service Units (SU) einzubinden (delpoyment). Service Units sind im Falle von BPEL die konkreten BPEL-XML-Dateien und bei XSLT die Transformations-Dateien. Das hat den Vorteil, dass die benötigte Logik nur einmal geschrieben werden muss. Die Service Units müssen dann nur noch die jeweiligen Konfigurationsparameter enthalten.

3.3. Shared Library

Diese Art von Komponenten stellt Funktionen für mehrere JBI Komponenten zur Verfügung. Dies ist nützlich bei großen Komponenten, in denen über mehrere Binding Components oder Service Engines die gleichen Libaries benötigt werden.

3.4. Service Assembly (SA)

Ein Service Assembly ist die Zusammenfassung von mehreren Service Engines, Binding Components und Service Libraries. Es handelt sich um eine komposite Komponente. Nicht alle Komponentenarten müssen in ihr vorkommen. Die enthaltenen Komponenten stellen die benötigten Funktionalität für eine JBI Applikation bereit. Damit wird die Verteilung vor allem für Benutzer oder Administratoren einer gesamten Applikation erleichtert. In einigen Fällen kann es vorkommen, dass zwangsläufig eine Reihenfolge bei der Installation zu beachten ist. Die kann in einer Service Assembly festgehalten werden.

3.5. Komponentenverteilung

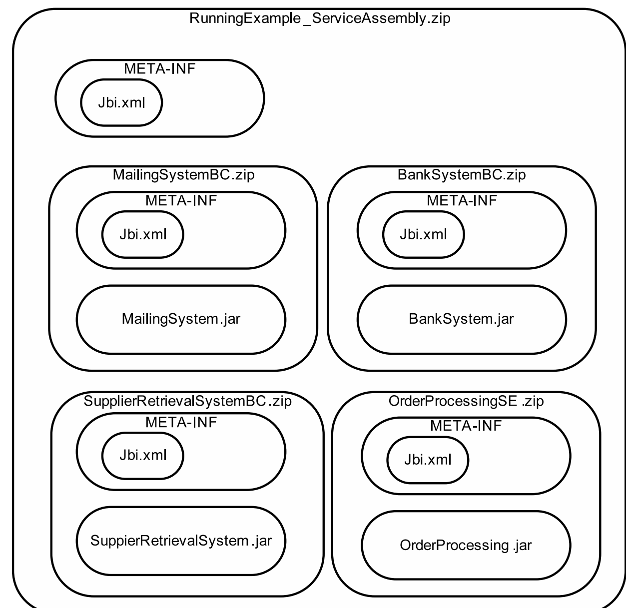

Abbildung 2. Komponentenpacketierung

Allein die Spezifikation von zu implementierenden Java-Klassen ermöglicht keine Verwendung in verschiedenen Implementierungen einer JBI Umgebung. Zusätzlich muss vorgegeben werden, in welchem Dateiformat die Distribution der Komponenten zu erfolgen hat. Dabei verzichtet JBI auf eine eigene Dateiendung und nimmt sich, wie in Java üblich, den zip-gepackten Komponenten an. Dabei wird die Endung .zip beibehalten. Innerhalb des Archivs muss ein Ordner META-INF existieren, in welchem die *jbi.xml* enthalten sein muss. Innerhalb dieses Deskriptors wird definiert, um welche Komponentenart es sich handelt, was die

Komponente ausführt, wie sie benannt ist sowie viele weitere Details je nach Komponentenart. Bei den Komponentenarten wird analog zu der schon genannten Einteilung verfahren: *component* , *service-assembly, shared-library, services* . Bei dieser Unterteilung mag den Leser die Unterscheidung zwischen *component* und *service* wundern. *component* wird für die Service Engine und die Binding Component verwendet. Bei einem *service* handelt es sich um die Service Unit (z.B. BPEL-XML-Dateien) welche in eine passende Service Engine hineindeployed werden kann. Für nähere Details zum Deskriptor sei auf Kapitel 6.3 der Spezifikation [9] verwiesen.

4. JBI Architektur

Die im vorherigen Abschnitt beschriebenen JBI Komponenten werden nun in das Gesamtbild einer JBI Implementierung eingeordnet. Der im Bild dargestellte Aufbau verdeutlicht den Zusammenhang. Dabei werden im Weiteren neue Begriffe eingeführt, die hauptsächlich aus dem Bereich des Enterprise Service Bus kommen und simultan bei JBI verwendet werden.

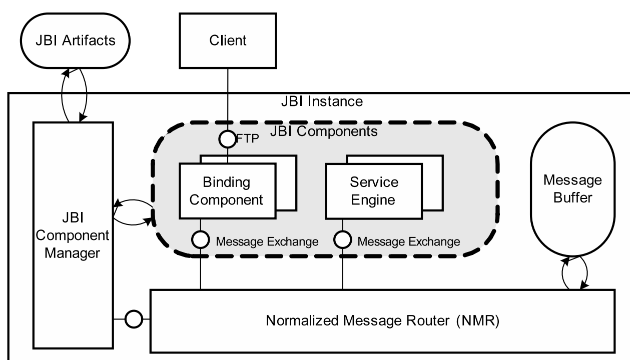

Abbildung 3. JBI Architektur

Die Unterscheidung zwischen *Service Consumer* (z.B. Client) und *Service Provider* (z.B. Service Engine) wird zum besseren Verständnis bei der Darstellung von Anfragen und dem Routing verwendet. Der *Service Consumer* initiiert eine Anfrage an das System und erwartet eine Antwort. Der *Service Provider* stellt die benötigte Funktionalität bereit und bearbeitet die durch den *Service Consumer* gestellte Anfrage. Um Anfragen senden oder empfangen zu können, benötigt jede Komponente einen *Delivery Channel*. Dabei handelt es sich um einen bidirektionalen Komunikationskanal, welcher zur Kommunikation der JBI Komponenten mit dem Normalized Message Router benutzt. Mit Hilfe des *Delivery Channel* lassen sich unter anderem auch benötigte Services per Servicename, Endpunktname oder WSDL-Beschreibung finden und ein Message Exchange auf dem *Delivery Channel* ablegen.

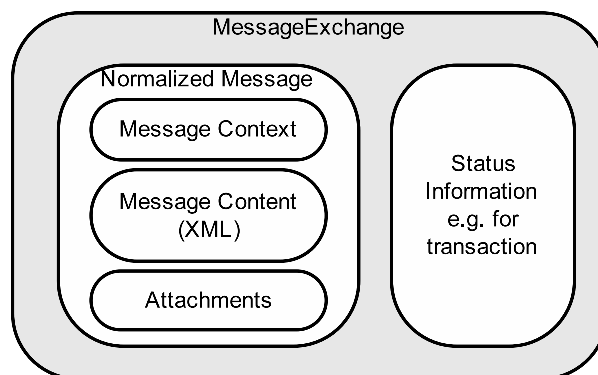

Abbildung 4. MessageExchange

Der *Message Exchange* ist der für JBI benötigte Java-Container für die normalisierte Nachrichten. Dieser enthält je nach Message Exchange Pattern (siehe Abschnitt 5) die gestellte normalisierte Anfrage und Antwort. Zusätzlich zu den eigentlichen Nachrichten enthält er Fehlermeldungen, Status, Endpunktinformationen sowie den Transaktionskontext. *Normalized Message* ist der JBI Begriff für eine normalisierte Nachricht und besteht aus 3 Teilen: Payload, Properties und Attachments. Der Payload enthält die eigentliche Nachricht im XML Format, währenddessen die Properties die Kontextinformationen halten. Dazu zählen z.B. Sicherheitseinstellungen wie Access Control Lists. Die Anhänge (Attachments) dienen der Übertragung von nicht XML Daten und müssen innerhalb der normalisierten Nachricht nach den Standards WS-I Attachments Profle 1.0 [18] oder W3C XML-binary Optimized Packaging (XOP) [16] eingebunden werden. Eine nicht normalisierte Nachricht wird als *Bound Message* (z.B. FTP Nachricht) bezeichnet. Sie liegt im jeweiligen Protokollformat vor. Die Umwandlung zwischen einer *Bound Message* und der *Normalized Message* erfolgt innerhalb der Binding Components. Der Zusammenhang zwischen den Begriffen *Normalized Message* und *Message Exchange* wird im Bild 4 verdeutlicht.

5. Message Exchange Patterns

Als grundlegendes Element setzt JBI auf die WSDL (Web Service Description Language) Schnittstellenbeschreibung auf. Sie dient der Abstraktion und der einfachen Integration von Web Services. JBI agiert nach der noch in Entwicklung befindlichen Spezifikation 2.0 [17]. Sie definiert vier verschiedene Nachrichtenaustauschmuster. Im Nachhinein wurde aus Kompatiblitätsgründen ein Mapping für WSDL 1.1 [15] bereitgestellt.

Message Exchange Patterns stehen für die möglichen

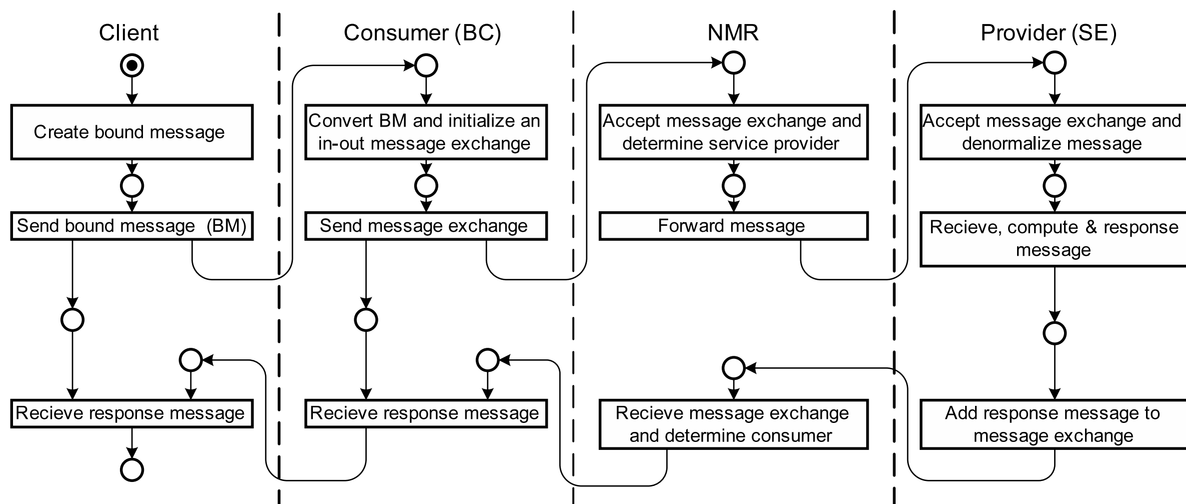

Abbildung 5. Request-Response

Kommunikationsmuster. Dabei wird auf den in WSDL 2.0 spezifizierten Muster aufgebaut. In WSDL werden 4 Arten unterschieden. In den Klammern werden die dazugehörigen Begriffe innerhalb von JBI benannt.

- One-Way (In-Only)
- Reliable-One-Way (Robust In-Only)
- Request-Response (In-Out)
- Request-Optional Response (In-Optional Out)

Diese Arten können von jeder JBI Implementierung erwartet werden. Allerdings können zusätzlich neue weitere Arten in jeder JBI Umsetzung implementiert worden sein. Dabei muss davon ausgegangen werden, dass diese nur in der jeweiligen Implementierung nutzbar sind und eine Austauschbarkeit der JBI Komponenten bei der gleichen Implementierung nicht mehr möglich ist.

Die Kommunikationsmuster sind besonders relevant für den Normalized Message Router. Er muss von vornherein wissen, wie die Kommunikation zwischen zwei Endpunkten abläuft. Nur so kann der Router geeignet reagieren und wissen, ob er auf eine Antwort warten muss. Das Beispiel eines Ablaufes für Request-Response wird in Abbildung 5 dargestellt. Er beinhaltet die Aktionen, die dabei mindestens ablaufen.

6. Normalized Message Routing

Der Normalized Message Router nimmt alle normalisierten Nachrichten in Form eines Message Exchange von den angemeldeten JBI Komponenten über den *Delivery Channel* entgegen und leitet sie an das gewünschte Ziel weiter. Der Router entkoppelt *Service Provider* von deren *Service Consumer*.

Die Funktionsweise kann am Besten anhand eines Beispiels verdeutlicht werden. Zuerst wird der Ablauf abstrakt beschrieben und im Weitern noch einmal anhand einer Beispielanwendung verdeutlicht.

Der folgende Fall bezieht sich auf das Kommunikationsmuster request-response wie in Abbildung 5 beschrieben. Die anderen Muster erfolgen nach einem ähnlichen Ablauf.

1. Client sendet Bound Message an BC
2. Binding Component normailisiert die Bound Message und erzeugt daraus ein Message Exchange mit dem jeweiligen Nachrichtenmuster, der normalisierten Nachricht (Normalized Message) und den Zielendpointinformationen. Anschließend übergibt die Binding Component den MessageExchange über den DeliveryChannel an den Normalized Message Router
3. der Normalized Message Router entnimmt Endpointinformationen aus dem Message Exchange und sendet diesen weiter an den angegebenen Endpunkt
4. die Service Engine akzeptiert den Message Exchange am DeliveryChannel. Anschließend wird die Nachricht extrahiert und bearbeitet
5. Service Engine erzeugt eine Antwort als normalisierte Nachricht und hinterlegt diese im Message Exchange. Dieser wird dann an den Normalized Message Router zurück gesendet
6. Normalized Message Router leitet den Message Exchange an den DeliveryChannel der Binding Component

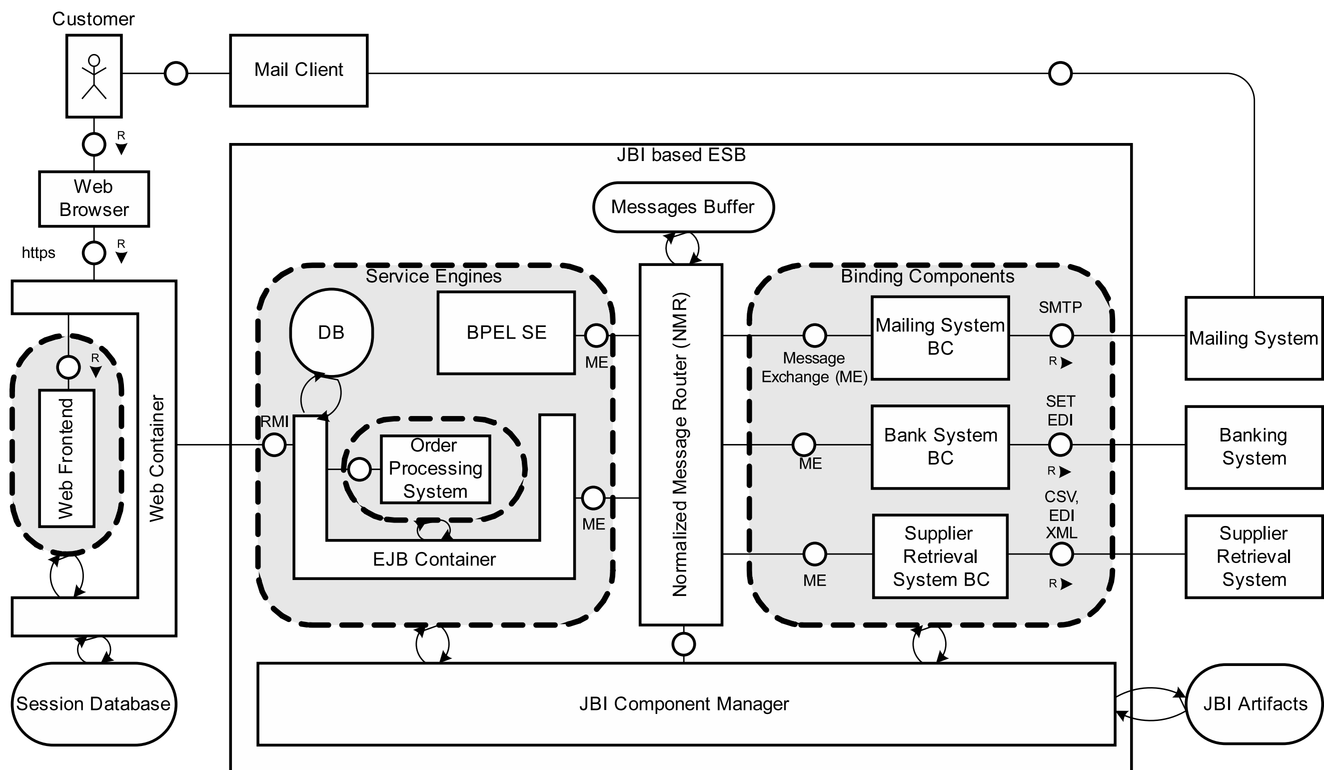

Abbildung 6. Running Example

7. Binding Component denormalisiert die Nachricht und übergibt die Antwort an den Client

8. Client empfängt BoundMessage von der BC

Erwähnenswert ist noch die Unterscheidung zwischen internen und externen Endpunkten. Externe Endpunkte sind z.B. die URLs der eingebundenen Web Services. Allerdings werden diese URLs nicht direkt an den Normalized Message Router weitergeleitet, sondern nur in der jeweiligen Komponente gehalten. Die Komponente nimmt dann dir Regisiterung am Router vor. Dabei wird ein interner Endpunkt erstellt und ein Mapping erfolgt.

Interessant ist auch die Abfage der aus dem Service Repository bekannten Daten. Ein Endpunkt kann über die WSDL-Schnittstelle, die direkte angebabe des internen Endpunkts oder der Servicenamen gefunden werden.

Die Funktionsweise des Routers wird in der Spezifikation nicht näher erläutert und wurde absichtlich offengehalten. Allein der grobe Ablauf nach den Message Exchange Patterns (siehe 4) wird abgesteckt. Somit bleibt für die JBI Implementierenden ein gewisser Spielraum, sich durch eigene kreative Ideen abzusetzen und dennoch eine spezifikationskonforme Kommunikation zu ermöglichen.

7. Beispielanwendung

In der Abbildung 6 wird eine Beispielanwendung skizziert. Dabei kann der Kunde Bestellungen über eine Webseite eingeben und abschicken. In diesem Beispiel werden Sie nicht sofort überprüft, sondern vorerst angenommen. Die eingegebenen Daten werden an das Ordering Processing System weitergeleitet. Dieses System muss anschließend über das extern angebundene Suupplier Retireval System den passenden Zulieferer finden und bei der Bank die Liquidität des Kunden abfragen. Diese Teilsysteme sind als Binding Component in das System integriert und machen die benötigte Protokollumsetzung. Sobald die Teilaufgaben durch das Ordering Processing System erfolgreich abgeschlossen sind, wird eine Bestätigung oder Absage über das Mailing System abgesandt.

Der Vorteil an dieser Architektur sind vor allem die übersichtlichen Verknüpfungen des Systems. Das Mailing System wird nur einmal eingebunden und kann anschließend durch alle Komponenten gefunden werden. Sie müssen keine SMTP Umsetzung mehr implementieren. Wird zusätzlich eine Nachverfolgung von Bestellungen benötigt, so muss nur Ordering Processing System erweitert werden. Eine weitere Möglichkeit ist, dass bestimmte Kunden anstelle einer Mail einen Brief bekommen. Dazu kann einmal

ein System geschrieben werden, welches die Nachrichten mit zusätzlicher Anschrift entgegen nimmt. Hier wird klar, dass ein Postkartensystem nicht zwangsweise die gleiche Schnittstelle implementiert wie das Mailing System. Das Problem der Schnittstellen wird durch JBI nicht gelöst. Hier treten semantische Probleme hervor, die wohl in näherer Zukunft nicht für den produktiven Einsatz gelöst werden können.

8. Component LifeCylce

Es werden grundsätzlich vier Zustände für Komponenten unterschieden. Der Startzustand wird eingenommen, sobald die JBI Komponente der JBI mgebung bekannt ist. Bevor eine Komponente gestartet werden kann, nachdem sie installiert wurde, muss sie initialisiert werden. Bei der Initialisierung hat die Komponente Zugriff auf den *ComponentContext*, mit dem Endpunkte am Normalized Message Router angemeldet, abgemeldet und ein Delivery Channel geöffnet werden kann. Nach der Initialisierung hat eine JBI Komponente alle benötigten Informationen erhalten um seine Funktionalität anzubieten oder den Bus Anfragen zu senden. Eventuell aufgebaute Resourcen beim initialisieren und starten der Komponente können durch das explizite beenden und herunterfahren wieder aufgelöst werden. Der Standardlebenszyklus ist in Abbildung 7 ersichtlich.

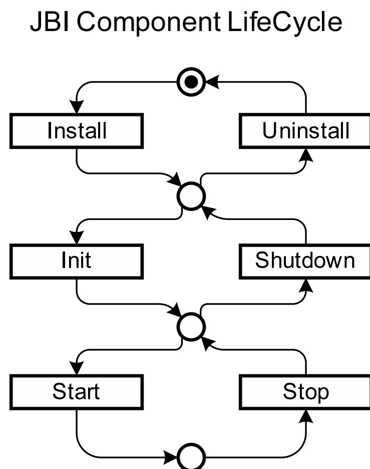

Abbildung 7. Komponentenlebenszyklus

JBI selbst lässt sich durch weitere Zustände erweitern. Das ist allerdings nur für den laufenden Zustand nach einem Start der Komponente möglich.

9. Management von JBI Komponenten

Eine JBI Implementierung bietet die Möglichkeit, seine Komponente zu administrieren. Dazu gibt die Spezifikation für die Komponenten JMX [4] Schnittstellen vor, die implementiert werden können. Dazu gehören unter anderem die ManagementBeans *InstallationServiceMBean*, *InstallerMBean*, *DeploymentServiceMBean*, *LifeCycleMBean*, *ComponentLifeCycleMBean*. Mit Hilfe dieser Schnittstellen kann eine Oberfläche dem Administrator die Verwaltung von Komponenten erheblich erleichtern. Insgesamt umfassen die Managementfunktionen in JBI den Umfang:

- Installation of JBI Komponenten
- Manipulation des Lebenszykluses
- durch Komponenten selbst definierte weitere Managementfunktionen

Wie schon angedeutet werden alle Managementaufgaben mittels JMX realisiert, was innerhalb der Java-Umgebung sehr verbreitet hat. Die Bereitstellung eigener Managementfunktionen ist allerdings fragwürdig, da eine JBI Implementierung nicht generisch die Semantik hinter den zusätzlichen Funktionen erkennen kann.

10. Quality of Services

Die Spezifikation von JBI definiert, wie Services gefunden, die benötigten Daten in einen Normalized Message Router eingespeist und die Komponenten verwaltet werden können. Dies allein reicht jedoch für Anwendungen im unternehmerischen Bereich nicht aus. Neben der performanteren Verarbeitung spielt es in diesem Umfeld eine große Rolle, dass asynchrone Nachrichten übertragen werden. Zusätzlich ist das Zielsystem einer Nachricht evtl. kurzfristig aufgrund eines Neustarts nicht erreichbar. Der Normalized Message Router darf für die Qualifizierung im unternehmerischen Umfeld die Daten nicht nur einfach annehmen und dann ungesichert weiterleiten. Die Qualität der Übertragung ist sehr wichtig. Eine einmal abgesandte Nachricht darf nicht mehrfach ihr Ziel erreichen. Wird beispielsweise in einem Bestellsystem eine Bestellungen mehrmals eingespeist, wird die Bestellung nicht korrekt verarbeitet. Im schlimmsten Fall erfolgt gar keine Bestellung oder im anderen Fall einer Mehrfachbestellung. In jedem Fall ist der Kunde sehr verärgert und wird den Anbieter wechseln. Die Wichtigkeit solcher Qualitätskritierien sollte klar sein und wird fälschlicherweise von vielen als gegeben vorausgesetzt.

JBI selbst definiert nur lokale Transaktionen. Diese finden sich in diversen Klassenstrukturen (Message Exchange) wieder. Die gerade interessante Form von verteilten Transaktionen lässt sich JBI Spezifikation für zukünftige Versionen offen. Dieser Part ist daher bisher nicht spezifiziert und wird den jeweiligen Implementierenden überlassen.

Implizit erwartet der Benutzer einer JBI Implementierung auch, dass bei Transaktionen die Nachrichten intern abgespeichert werden. Diese müssen daher z.B. auf einer Festplatte zwischengespeichert werden, bis ein Nachrichtenaustausch vollständig abgewickelt wurde. Stürzt das System ab, so muss der Zustand vom Zwischenspeicher wieder eingelesen werden können. Aus sicherheitstechnischen und datenschutzrechtlichen Gründen dürfen Nachrichten allerdings nicht unverschlüsselt abgespeichert werden. Alle hier angesprochenen Anforderungen werden durch die Spezifikation nicht definiert. Es werden Beispiele gegeben, wie diese Anforderungen realisiert werden könnten. Eine Pflicht gibt es dagegen nicht. Diese ist allerdings nicht unbedingt zur Einbindung von Komponenten relevant, da viele Anforderungen intern durch den Normalized Message Router durchgeführt werden können ohne dass die einzelnen Komponenten davon Kenntnis haben müssen.

Viel interessanter für die jeweiligen Komponenten sind Sicherheitsrichtlinien. Hier wären zum Beispiel die so genannten Access Control Lists (ACLs) anzusiedeln. Sie ließen sich in einer normalisierten Nachricht im Kontext (Message Context) unterbringen. Auch hierzu macht die Spezifikation keine genauen Angaben und verweist auf Implementierungsdetails der jeweiligen JBI Komponenten. Die Verlagerung der Bearbeitung von Sicherheitsfunktionalität in die JBI Komponent ist allerdings kritisch zu sehen. Dadurch entstehen wiederum Inkompatibilitäten, die eine Benutzbarkeit von Services erschwert. Passen die Implementierungen der Sicherheitsrichtlinien nicht zusammen, kann im Notfall der Service nicht in Anspruch genommen werden, obwohl die Schnittstellen zueinander passen. Hier erscheint das Problem ähnlich wie bei Web Services. Sicherheitsaspekte wurde auch dort erst durch weitere Spezifikationen wie Web Service Security [12] behandelt.

11. XML Datentypen als Nachrichtenformat

Im Zusammenhang mit dem Enterprise Service Bus hat sich das Datenformat XML so gut wie durchgesetzt. Das bedeutet, dass die Nachrichten im Bus über XML Datentypen versendet werden. XML Datentypen bilden die Grundlage für JBI und sind daher besonders interessant für eine genauere Betrachtung. XML hat den Vorteil, dass es meist menschenfreundlich verfasst ist. Gerade die Metadaten geben meist Aufschluss für den Inhalt, auch wenn dieser nicht immer auf den ersten Blick erkenntlich ist. Gegenüber reinen Textdateien ohne Metadaten ein klarer Vorteil. Zusätzlich bietet XML eine relativ losgelöste Beziehung zwischen dem XML Dokument und dem zugehörigem Datenschema. Dagegen werden in einer Datenbank die Daten direkt zur Tabelle gespeichert. Für diese Tabelle ist klar geregelt, welche Datenstruktur darunter liegt. Sie zu ändern bedeutet meist hohen Aufwand, da alle Daten angepasst werden müssen. Dies ist bei XML dagegen relativ leicht möglich. Allerdings gilt auch hier, dass dies nur funktioniert solange das Schema erweitert wird. Dann können auch alte Implementierungen mit dem Datenformat noch etwas anfangen, da die XML Parser die zusätzlichen Daten einfach ignorieren. Ein weiterhin nicht zu unterschätzender Faktor sind die diversen Spezifikationen um XML herum. Hier sollen nur XQuery und XSLT genannt werden. Die Transformation von Daten wird in Bezug auf die Services immer wichtiger, da die Services meist nicht die identischen Datenformate an den Schnittstellen verwenden. XSLT bietet dafür eine geeignete Basis um die Daten umzuwandeln. Weiterhin sind XML Parser (DOM Parser und SAX Parser) schon weit verbreitet und über einige Jahre etabliert. Ein Mapping von XML auf die jeweilige Programmiersprache ist bei den heutigen Middleware-Plattformen möglich. In Java existiert dazu z.B. JAX-B [7]. Von dieser Spezifikation macht auch JAX-WS 2.0 [6] für die Java Enterprise Edition Gebrauch, um Java Schnittstellen als Web Service bereitzustellen. Bei Web Services liegen die Schnittstellenbeschreibung in Form von WSDL vor. Bei den übertragenen Daten handelt es sich dabei um XML Datentypen. Wird XML innerhalb eines ESB verwendet, entfällt der Konvertierungsaufwand.

Dagegen nimmt man den Aufwand bei Legacy-Systemen in Kauf. Meist wird zum Ansprechen ein proprietäres Protokoll verwendet, bei dem eine Konvertierung der Daten in jedem Fall notwendig ist. Bei der Verwendung von XML muss klar sein, dass das Parsen von XML-Dokumenten einen ziemlich hohen Aufwand bedeutet. Systeme können bei einem hohen Durchsatz dadurch schnell ausgelastet werden. Dies nimmt man in Kauf und versucht das durch Funktionalität zum Verknüpfen von mehreren ESBs auszugleichen.

12. Schlussfolgerungen

Die Entwicklung von JBI stellt einen großen Schritt voraus für die Entwicklung im Bereich des Enterprise Service Bus dar. JBI kann als Rahmeninfrastruktur für die Einbindung vieler verschiedener Anwendungen gesehen werden und zieht Nutzen aus den neuen technologischen Trends um XML und den Web Services. Da abzusehen ist, dass sich das Konzept des Enterprise Service Bus weiter verbreitet, wird JBI vermutlich nach und nach in bekannte Middleware-Plattformen Einzug halten. In der ersten Version der Java Enterprise Edition 5 (JEE5) liefert Sun sogar gleich einen eigenen auf JBI basierenden ESB mit. Die für den Java Bereich wichtigen Firmen wie IBM und BEA haben sich von der Teilnahme an der Spezifikation zurückgezogen und arbeiten an einer eigenen Lösung. Trotz vieler Vorteile von JBI bleibt abzuwarten, ob allein eine gute konkrete Idee zur Umsetzung reicht. IBM und BEA sind sicherlich nicht daran interessiert, dass alle Welt Komponenten für jeden En-

terprise Service Bus bauen kann.

Da JBI für die Microsoft-Fraktion ungeeignet ist, bleibt auch das Verhalten der Redmonder abzuwarten. Eventuell bieten Sie mit der Microsoft Windows Communication Foundation ein geeignetes Pendant.

Grundsätzlich wird JBI und der ESB nicht alle Probleme beheben. Es wird einige der aus EAI bekannten Probleme minimieren und neue schaffen. JBI löst das Problem von syntaktisch identischen Nachrichten und ermöglicht eine Implementierung einer umfassenden Message-Middleware, welche sich zum normalen Nachrichtenaustausch auch um die Qualitätssicherung kümmern kann. Die einzelnen Anwendungen melden sich nur an einem Bus an und können alle registrierten Services nutzen. Sobald ein Service ausfällt, kann der Normalized Message Router die Daten zwischenspeichern bis der benötigte Service wieder angebunden ist. Auf den ersten Blick erscheint daher durch JBI alles übersichtlicher zu werden. Allerdings ist in diesem Zuge zu erwarten, dass neue Probleme auftreten. Es wird eine Vielzahl von Services geben, welche ein ähnlichen Problem anders lösen. Es wird schwer werden in der schieren Menge der Services den Überblick über die angebotenen Dienste zu erlangen. Die Lösung der semantischen Probleme steckt allerdings erst in seinen Kinderschuhen und wird noch Jahre bis zum Produktiveinsatz brauchen. Bis dahin sollen XSLT-Engines die Konvertierung der Daten ermöglichen, da der Customer aus Service 1 nicht automatisch zu Customer aus Service 2 passt. Das erschwerte eine Verknüpfung von Services erheblich.

Dennoch ist die hinter JBI steckende Idee ist mit einem positiven Blickwinkel zu betrachten, da unterschiedliche Schnittstellen und inkompatible Datenkommunikation innerhalb eines ESB vermieden werden soll. Vielleicht bringen die folgenden Versionen noch die benötigten Verbesserungen damit sich JBI wie von selbst durchsetzt.

Literatur

[1] D. A. Chappel. *Enterprise Service Bus. Theory in Practice.* O'Reilly, 2004.

[2] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: designing, building, and deploying messaging*. Addison Wesley, 2004.

[3] F. Menge. *Service-orientierte Architektur (SOA) verglichen mit Komponentenmiddleware und Enterprise Application Integration (EAI).* Seminar Systemmodellierung 2006, 2006.

[4] S. Mircosystems. Java management extensions (jmx) specification. http://www.jcp.org/en/jsr/detail?id=3, 2002.

[5] S. Mircosystems. J2ee connector architecture 1.5. http://-www.jcp.org/en/jsr/detail?id=112, 2003.

[6] S. Mircosystems. Java api for xml-based web services (jax-ws) 2.0. http://www.jcp.org/en/jsr/detail?id=224, 2006.

[7] S. Mircosystems. Java architecture for xml binding (jaxb) 2.0. http://www.jcp.org/en/jsr/detail?id=222, 2006.

[8] S. Mircosystems. Java enterprise edition 5. http://java.sun.-com/javaee, 2006.

[9] S. Mircosystems. Jbi specification. http://www.jcp.org-/en/jsr/detail?id=208, 2006.

[10] S. Mircosystems. Sun service orientated business integration. http://java.sun.com/integration, 2006.

[11] OASIS-OPEN. Ws-bpel. http://www.oasis-open.org/-committees/wwwtc_home.php?wg_abbrev=wsbpel, 2005.

[12] OASIS-OPEN. Web service security. "http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, 2006.

[13] E. Pulier and H. Taylor. *Understanding Enterprise SOA*. Manning Publications, 2006.

[14] W3C. Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 1999.

[15] W3C. Web services description language (wsdl) 1.1. http://-www.w3.orgTR/wsdl, 2001.

[16] W3C. W3c xml-binary optimized packaging (xop). http://www.w3.org/TR/xop10/, 2005.

[17] W3C. Web services description language (wsdl) 2.0. http://-www.w3.orgTR/2006/CR-wsdl20-primer-20060327/, 2006.

[18] WS-I. Ws-i attachments profle 1.0. http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html, 2006.