



Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Bachelorarbeit

Überwachung asynchroner Web-Service- Kommunikation über einen Enterprise Service Bus

Matthias Schulz

ma.schulz@email.de

Studiengang Informatik

Matr.-Nr. 5791115

Fachsemester 8

Erstgutachter: Professor Dr. N. Ritter

Zweitgutachter: Dr. Axel Schmolitzky

Inhaltsverzeichnis

1	Einleitung	1
1.1	Zielsetzung	3
1.2	Eingrenzung	4
1.3	Gliederung der Arbeit	4
2	Grundlagen	7
2.1	Service-orientierte Architekturen	7
2.2	Verzeichnisdienste	8
2.3	XML Web Services	10
2.3.1	WSDL	11
2.4	SOAP	13
2.4.1	WS-Addressing	15
2.5	Enterprise Service Bus	16
3	Anforderungsanalyse	21
3.1	EPEJ-Beispiel	21
3.2	Protokollierung der Kommunikation	22
3.3	Asynchronität auf Anwendungsebene	24
3.4	Anforderungen an einen ESB	25
3.5	Konkretisierung des Problems	27
4	Konzept und Implementierung	29
4.1	Lösungsansatz	29
4.2	Mediatoren	30
4.2.1	LogMediator	30
4.2.2	RequestMediator	31
4.2.3	AddressingInfo	33
4.2.4	ResponseMediator	34
4.2.5	WSASendMediator	36
4.2.6	DropMediator	38
4.3	Überprüfung am Beispiel	38
5	Zusammenfassung und Ausblick	43
5.1	Zusammenfassung	43
5.2	Ausblick	44
	Literaturverzeichnis	45

Eidesstattliche Erklärung

51

1 Einleitung

Service-basierte Architekturstile sind aktuell scheinbar unumgänglich bei der Gestaltung neuer, flexibler Softwaresystem und werden für Organisationen als wichtiger Bestandteil für agile Prozesse angepriesen. “Wer nicht in SOA investiere, riskiere seine Konkurrenzfähigkeit“ ([ST07]) liest man beispielsweise öfters in einschlägiger Computerpresse. Viele dieser unpräzisen Argumente kann man sicherlich als Marketing ansehen, dennoch bieten Service-orientierte Architekturen (SOAs) auch große Vorteile bei konsequenter Umsetzung.

Sie können Prozesse und Anwendungen in Unternehmen durch Modularisierung flexibler und offener für Veränderungen gestalten und damit einen gewissen Grad an Agilität in der Umsetzung neuer Prozesse ermöglichen. Dies wird durch eine saubere Trennung von fachlicher Logik, Prozessablaufs- sowie Präsentationslogik in einer mehrschichtigen Architektur erlaubt, welche flexible Änderungen bei geschäftlichen sowie technischen Anpassungen bietet.

Besonders die Vermaschung heterogener, unternehmensübergreifender Geschäftsprozesse wird durch eine service-orientierte Architektur begünstigt. Diese zeichnet sich durch fachliche Beschreibungen von Schnittstellen, durch die Abstraktion von der Implementierung der Dienste sowie der Interoperabilität zwischen Services und Geschäftsprozessen aus.

Als Beispiel für solche Geschäftsprozesse lassen sich verschiedene Behörden mit unterschiedlichen Anforderungen und Anwendungen betrachten, die mit einem gemeinsamen Ziel zusammenarbeiten müssen und für die daher ein automatisierter Austausch von Informationen notwendig wird. Im Sinne von Webdiensten fließen hier zwei verschiedene Konzepte ein, namentlich Orchestrierung und Choreographie.

Bei der Orchestrierung von Diensten werden diese von zentraler Stelle aus kontrolliert. Orchestrierung beschreibt die Interaktion der Webdienste auf nachrichtendienstlicher Ebene (vgl. [FFO05], S. 118). Der Vorteil ist die Einfachheit, da die Ausführungsreihenfolge zentral dokumentiert ist. Allerdings hat dieses Konzept Nachteile bei organisationsübergreifenden Prozessen, da dort eine solche zentrale Stelle nicht existiert. Als Beispiel für die Anwendbarkeit von Orchestrierung kann ein verteiltes Informationssystem in einem Fertigungsbetrieb dienen, bei dem die einzelnen Systeme entlang der Fertigungsstrecke anhand des Ablaufplans einer Koordinationsstelle durch den Austausch von Nachrichten miteinander kooperieren.

Im Gegensatz zur Orchestrierung legt die Choreographie den Schwerpunkt auf die Zusammenarbeit unabhängiger Prozesse (vgl. [FFO05], S.119). Da es sich um autonome Teilprozesse handelt, wird eine einheitliche Schnittstelle benötigt, die eine Kooperation der Teilsysteme beziehungsweise einen Nachrichtenaustausch zwischen ihnen erlaubt. Die zulässigen Nachrichten zwischen den Systemen werden hierbei durch eine entspre-

chende Beschreibungssprache (WS-CDL, siehe [KBR⁺04]) definiert. Ein Beispiel für eine Choreographie können über Staatsgrenzen hinaus agierende Behörden darstellen. Bei einem Verstoß gegen geltendes Recht in größerem Maßstab kann hierbei die Zusammenarbeit von Behörden mehrerer Staaten notwendig sein. Um diese Kommunikation auf Anwendungsebene zu gewährleisten, müssen Nachrichten über standardisierte Schnittstellen ausgetauscht werden. Die Vorgänge innerhalb der Systeme der beiden Behörden bleiben dabei jeweils vor der anderen verborgen.

Der Nachteil von Choreographien besteht darin, dass es keine Möglichkeit gibt, die Korrektheit des Nachrichtenflusses zu gewährleisten. Jedoch ist es oftmals wünschenswert, sicherzustellen, dass die Kommunikation entsprechend eines verabschiedeten Protokolls gelaufen ist. Dieses Problem kann durch Nutzung einer Middleware-Ebene angegangen werden, die sich in logischer Hinsicht zwischen den Systemen befindet. Im Umfeld von service-orientierten Architekturen und Web Services kommt hierbei ein Service Bus zum Einsatz, über den alle Nachrichten auszutauschen sind ([Men07], Seite 2):

“An Enterprise Service Bus is an open standards, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services in a secure and reliable manner.”

Der Enterprise Service Bus kann als eine zentrale Instanz zur Kontrolle der Korrektheit des choreographierten Nachrichtenaustauschs verwendet werden, wenn die Nachrichten im ESB über einen Mediator an eine Protokollierungsinstanz weitergeleitet werden.

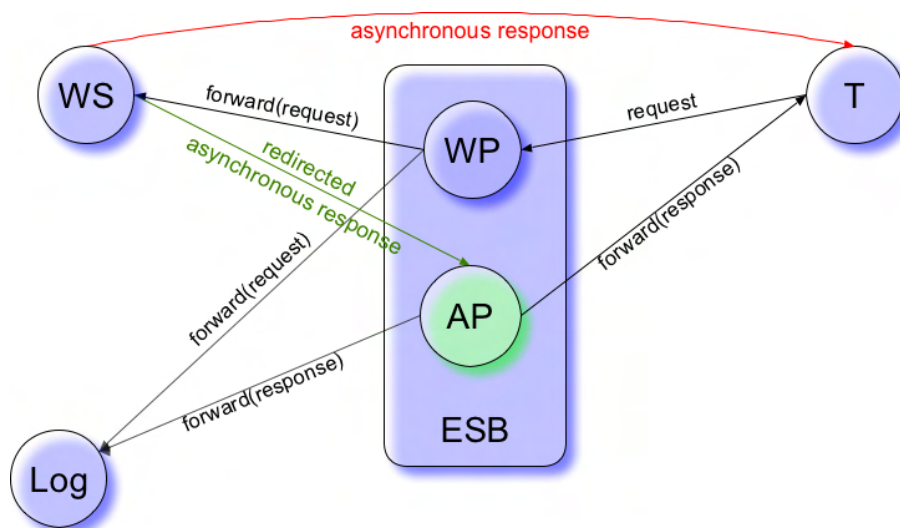


Abbildung 1.1: Schematische Darstellung der Web Services, die über den Enterprise Service Bus kommunizieren

In Abbildung 1.1 ist ein schematischer Aufbau der kooperierenden Systeme dargestellt. Der Teilnehmer (T) sendet seine Anfrage nicht direkt an den Webdienst (WS), sondern über einen Webproxy (WP) innerhalb des Enterprise Service Busses. Im Falle von syn-

chronen Nachrichten (hier nicht dargestellt) würde die Antwort des Webdienstes wieder über den ESB an den Teilnehmer gehen. Bei asynchroner Kommunikation enthält die Nachricht jedoch eine Adresse, an welche die Antwort zu senden ist. Die asynchrone Antwort darf aber nicht an den Teilnehmer gesendet werden (roter Pfeil), sondern ist über einen Antwortproxy (AP) innerhalb des ESB umzuleiten. Dies geschieht durch die Modifikation der Adressinformationen der Nachrichten, dem WS-Addressing-Header. Der Antwortproxy muss wissen, welche Empfängeradresse die umgeleitete Nachricht bekommen soll, und leitet diese dann entsprechend an den Teilnehmer weiter. Die beiden involvierten Komponenten des ESB (hier WP und AP) senden jeweils eine Kopie der Nachricht an einen Protokolldienst (Log), der Nachrichten in Form eines Ereignisses registriert. Über die gesamte Liste der empfangenen Ereignisse kann jederzeit die Konversation der Systeme nachvollzogen und deren korrekter Fluss überprüft werden.

Synchrone und asynchrone Kommunikation auf Nachrichtenebene stellen für aktuelle Service Bus-Produkte keine Schwierigkeiten dar, jedoch kann die Asynchronität auch auf die logische Anwendungsebene gehoben werden. Hierbei sendet der Teilnehmer eine Anfrage über den Service Bus und arbeitet danach an anderen Aufgaben weiter. In einem nicht näher spezifizierten Zeitraum verarbeitet der Empfänger die Nachricht und sendet irgendwann eine Antwort in Form einer neuen Anfrage an den Absender (Teilnehmer). Um festzustellen, an wen die Antwort zu senden ist, verwendet er dabei die mitgesendeten Adressinformationen, wobei die Absenderadresse dem Teilnehmer entspricht.

1.1 Zielsetzung

Das Ziel dieser Ausarbeitung ist die Erarbeitung einer Möglichkeit zur gezielten Manipulation der WS-Addressing Header-Informationen einer Nachricht sowie einer Erweiterung des Enterprise Service Busses mit der Programmiersprache Java. Die Manipulation der Nachricht zielt auf die Umleitung der asynchronen Antwort über einen Antwort-Proxy innerhalb des ESB zum Zwecke deren Protokollierung im Log-Proxy, weitergehend muss die Antwort auch den Absender der Anfrage erreichen. Hierfür muss der Enterprise Service Bus mit einem Mediator ([GHJV04], S.273 ff.) ausgestattet werden, der es ermöglicht, beim Empfang einer asynchronen Nachricht einen solchen Antwort-Proxy vorzubereiten. Entweder muss für jede Nachricht ein eigener Antwort-Proxy erzeugt werden, dessen Ziel die Antwortadresse der anfragenden Webanwendung ist. Die zweite Möglichkeit ist ein generischer Antwort-Proxy, der die Möglichkeit bietet, sich mittels einer Schnittstelle zum Mediator für eingehende Nachrichten parametrisieren zu lassen.

Am Ende der Arbeit soll sowohl ein Mediator zur Erweiterung des ESB als auch ein Antwortproxy für den ESB entstehen, die gemeinsam einen allgemeinen Ansatz zur Integration asynchroner Nachrichten in den Enterprise Service Bus zur Überprüfung von Choreographien von Diensten heterogener Organisation bieten.

1.2 Eingrenzung

Da die Themen um asynchrone Webdienste und den Enterprise Service Bus sehr weitreichend sind, ergibt sich die Notwendigkeit einer Eingrenzung der Fragestellung.

Die Arbeit wird sich ausschließlich mit der Verwendung der SOA Erweiterung WS-Addressing zur Manipulation der Header-Informationen der Nachrichten befassen, entsprechend der aktuellen Version von 2006. Zur Manipulation können auch andere Technologien dienen, wie sie zum Beispiel Suns *Java API for XML-based remote procedure calls*¹ (JAX-RPC) bietet, auch eine direkte Manipulation im Enterprise Service Bus ist möglich. Es wird vorausgesetzt, dass alle Teilnehmer (siehe T und WS in Abbildung 1.1) die entsprechenden Erweiterungen bieten, um mit Nachrichten zu arbeiten, die WS-Addressing Header aufweisen.

Die Modifikationen an den Nachrichten erfordern es, dass diese im Klartext vorliegen. Eine Verschlüsselung mit XML-Encrypt² kommt daher nicht in Frage, auch eine Signierung der Header-Informationen mit XML Signature (vgl. [WCL⁺05], S. 277) ist nicht Gegenstand dieser Arbeit. Weitere sicherheitsrelevante Erweiterungen von SOA (WS-Security³ und WS-SecureConversation⁴) gehören auch nicht zum Thema.

Eine grundsätzliche Auseinandersetzung mit der Zuverlässigkeit der Zustellung von Nachrichten über das Transportprotokoll HTTP, wie z.B. mit der SOA Erweiterung WS-ReliableMessaging (siehe [WCL⁺05], S. 187 ff.) möglich, ist auch nicht Inhalt der Arbeit.

Diese Ausarbeitung ist Teil eines größeren Gesamtprojektes zur Überwachung von Choreographien und Protokollierung von Nachrichten, in dessen Kontext eine Möglichkeit zur semantischen Prüfung der Inhalte der Nachrichten als Ziel steht. Auf diese Möglichkeit in dieser Arbeit nicht eingegangen. Zur Feststellung der Richtigkeit der Kommunikation dient hier lediglich eine Überprüfung auf das Vorhandensein der Nachrichten.

1.3 Gliederung der Arbeit

Im ersten Kapitel der Arbeit erfolgt eine Einleitung in das Thema mit einer Motivation für die Aufgabenstellung sowie einer Zielsetzung der Arbeit. In diesem Kontext wird ein kleines Beispiel erarbeitet, welches später für die Anforderungsermittlung und die Implementierung dient.

Das nächste Kapitel führt in die technischen Grundlagen der verwendeten Softwaretechnologien und -konzepte ein. Hierzu gehört eine Definition von service-orientierten Architekturen sowie der *Web Service Description Language*⁵ (WSDL). Im dritten Abschnitt des Grundlagenkapitels werden dann die Konzepte und Produkte rund um SOA erläu-

¹<http://java.sun.com/webservices/>

²<http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>

³http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

⁴<http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html>

⁵<http://www.w3.org/TR/wsdl20/>

tert, inklusive der für die Arbeit benötigten Erweiterung WS-Addressing⁶. Weiterhin erfolgt eine Beschreibung des Enterprise Service Busses sowie dessen Erweiterungsmöglichkeiten durch Mediatoren.

Das dritte Kapitel befasst sich mit einer Anforderungsermittlung für die eine mögliche Lösung, erfasst also die benötigte Funktionalität der im Laufe der Arbeit angefertigten Erweiterung des Enterprise Service Busses. Hierfür ist eine Konkretisierung des Problems auf technischer Ebene notwendig, die genau aufzeigt, an welchen Stellen es Probleme gibt. Der Anfang des Kapitels motiviert in diesem Sinne ein Beispiel, dass einerseits das Problem verdeutlicht und später auch für eine Lösung modifiziert wird.

Inhalt des vierten Kapitels ist das Konzept für die Implementierung der Lösung, das anhand von geeigneten Diagrammen sowie Beschreibungen vorgestellt und in Bezug auf die Anforderungen zu bewerten ist. Mit dem ausgearbeiteten Konzept soll dann eine Implementierung als Erweiterung des Enterprise Service Busses in Form eines Mediators sowie einer weiteren Lösung zum Antwortproxy erstellt werden. Anhand des im vorherigen Kapitels ausgearbeiteten Beispiels lässt sich die Umsetzung auf Funktionalität überprüfen.

Das abschließende Kapitel fasst die Arbeit zusammen und gibt einen Ausblick auf künftige Entwicklungen im Umfeld der erarbeiteten Lösung.

⁶<http://www.w3.org/TR/ws-addr-core/>

2 Grundlagen

In diesem Kapitel sollen zunächst die Grundlagen für das Konzept einer Service-orientierten Architektur vorgestellt werden. Weiterhin ist für das Thema der Arbeit eine Einführung in die Nutzung von Web Services sowie deren Beschreibungssprache WSDL notwendig. Danach wird die Kommunikationsform SOAP und deren Erweiterungen in Bezug auf Web Services vorgestellt, abschließend folgt dann der Enterprise Service Bus (ESB) im Allgemeinen sowie in der verwendeten Form. Ein wichtiges Thema hierbei spielen die Mediatoren, welche die Transformation der Anfragen innerhalb des ESB bewerkstelligen.

2.1 Service-orientierte Architekturen

Eine Service-orientierte Architektur (engl. *service-oriented architecture*) ist ein Paradigma einer abstrakten Software-Architektur. Aus der Unternehmenssicht ist dabei eine SOA erstmal ein Konzept zur Organisation von Prozessen (vgl. [ST07], S. 12):

“Eine service-orientierte Architektur (SOA) ist eine Unternehmensarchitektur, deren zentrales Konstruktionsprinzip Services (Dienste) sind. Dienste sind klar gegeneinander abgegrenzte und aus betriebswirtschaftlicher Sicht sinnvolle Funktionen. Sie werden entweder von einer Unternehmenseinheit oder durch externe Partner erbracht.“

Dienste bilden demnach die Grundlage einer Service-orientierten Architektur. Nach der Definition eines Wörterbuches ist ein Dienst die Erbringung einer Leistung für eine andere Person ([ZW88]). Im umgangssprachlichen Sinne ergibt sich dadurch einerseits die Bereitschaft, eine bestimmte Arbeit zu erbringen, als auch das Angebot, diese für andere Personen anzubieten (vgl. [MLM⁺06], S.8). Diese allgemeine Definition kann aus der Sicht der Softwaretechnik verfeinert werden (vgl. [Bar03], S. 19):

“A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services”

Ein solcher Dienst muss sich also über seine Funktionalität von anderen Diensten abgrenzen. Weiterhin besitzt er eine spezifizierte Aufgabe, die er für einen Dienstkonsumenten erledigt. Die Implementierung der Arbeitsweise bleibt hierbei vor dem Konsumenten verborgen. Ein Dienst besteht aus einer Schnittstelle, welche die Aufrufmodalitäten beschreibt, sowie einer verborgenen Implementierung.

Ein weiteres grundlegendes Merkmale einer SOA ist die lose Kopplung, also die Unabhängigkeit unter den Diensten sowie das dynamische Suchen und Einbinden jener. Verzeichnisse stellen Kataloge mit Diensten dar, die in einer maschinenlesbaren Form Beschreibungen der Dienste zur Verfügung stellen und damit die Suche ermöglichen.

Dynamisches Einbinden und Verzeichnisdienste erfordern standardisierte Schnittstellen und Beschreibungen, welche beispielsweise die *Web Service Description Language* (WSDL) bietet (siehe [CMRW07]). Die Verwendung von solchen offenen Standards begünstigt eine weite Verbreitung der Werkzeuge service-orientierter Architekturen sowie eine hohe Akzeptanz der Schnittstellen in der Industrie.

Eine weitere, auf die technischen Aspekte bezogene Definition einer SOA stellt [ST07] vor:

“A Service Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus.”

Diese Definition stellt die Teilnehmer einer SOA in den Vordergrund. Hierzu zählen Diensteanbieter (Service), der Nutzer sowie das Dienstverzeichnis. In Abbildung 2.1 sind diese schematisch dargestellt. Verzeichnisdienste nach UDDI ([CHRR04]) werden im nächsten Unterkapitel ausführlicher beschrieben. Die Sprache WSDL dient zum Austausch von Informationen eines Web Service, SOAP hingegen beschreibt die Kommunikation zwischen Nutzer und Anbieter (siehe [GHM⁺07]). WSDL und SOAP werden im Verlaufe dieses Kapitels näher beschrieben.

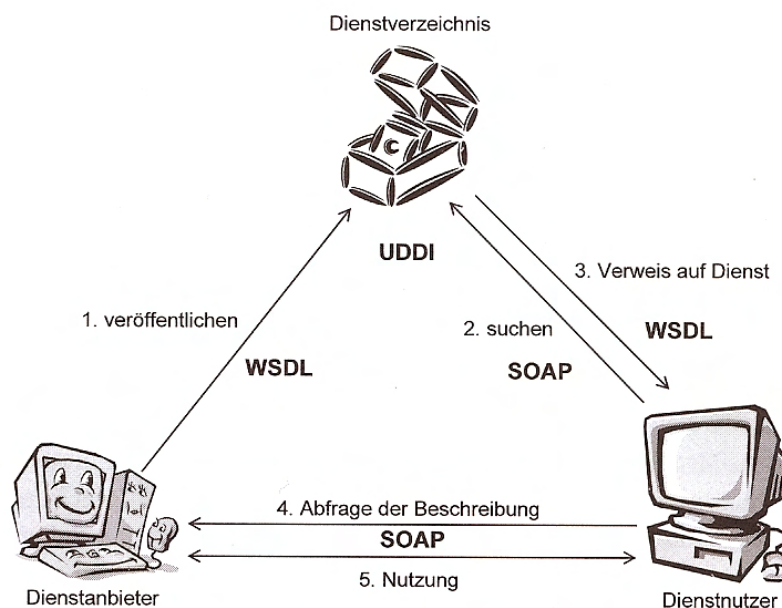


Abbildung 2.1: Web-Services-Dreieck, vgl. [Mel08], S. 56

2.2 Verzeichnisdienste

Wie in Abbildung 2.1 dargestellt, stellt ein Verzeichnisdienst eine zentrale Komponente einer SOA dar, da dieser einen sehr hohen Grad an loser Kopplung begünstigt. Ein wesentliches Merkmal der losen Kopplung stellt dabei das dynamische Suchen, Auffinden

und Einbinden von Diensten dar. Gerade in der Entwicklungszeit neuer Applikationen benötigen die Entwickler eine Anlaufstelle, bei der sie notwendige Funktionalitäten, die bereits implementiert wurden, suchen können. Ein Verzeichnisdienst agiert hier wie die gelben Seiten; geordnet nach Kategorien beschreiben die Anbieter die Leistungen ihrer Dienste.

Als Verzeichnisdienst in einer SOA sind zwei Ansätze von größerer Bedeutung - das *Universal Description, Discovery and Integration (UDDI)* -Protokoll sowie die *Web Service Inspection Language (WS-Inspection)*.

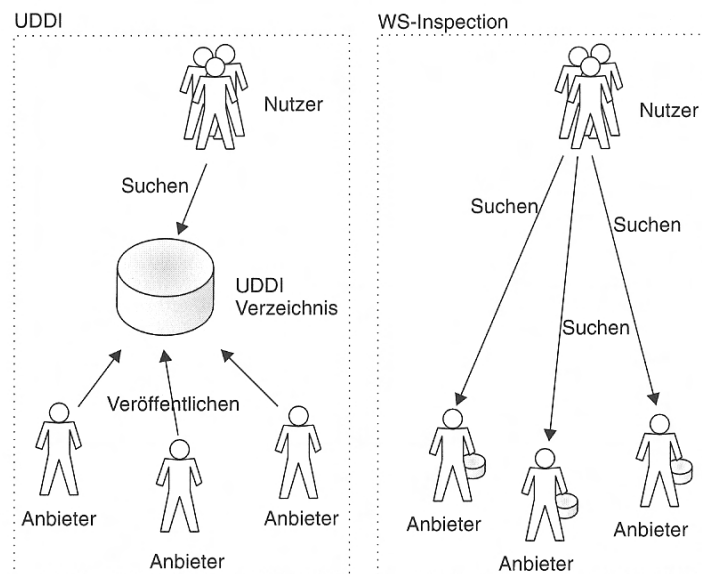


Abbildung 2.2: UDDI und WS-Inspection, vgl. [Mel08], S. 135

Die *Web Service Inspection Language* ist ein einfaches Konzept zum Beschreiben und Auffinden von Diensten, das vorrangig von Microsoft und IBM entwickelt wurde. WS-Inspection beruht auf einer standardisierten Beschreibung der Web Services in einer speziellen, XML-formattierten Datei, die jeweils immer unter dem gleichen Namen auf dem Server des Diensteanbieters auffindbar ist. Ein Client muss also die Anbieter von Diensten kennen, findet dann aber jeweils an der gleichen Stelle die Dienstbeschreibungen. Das Wissen über die Dienste ist also hier *dezentral*.

Im Kontrast zu WSIL sind Verzeichnisdienste nach UDDI *zentrale* Datenbanken, ähnlich einem Telefonbuch. Die Anbieter tragen ihre Dienste in die Datenbank ein, die Nutzer können diese dann durchsuchen und anhand ihrer Beschreibung den benötigten finden.

Ein solcher Verzeichnisdienst klärt bisher aber keine Fragen über die Qualität der angebotenen Dienste, gibt keine Möglichkeit zur Abrechnung und hat auch keine Angabe über den Verantwortlichen eines Services. Diese rechtlichen und wirtschaftlichen Fragen stehen momentan einer spontanen und dynamischen Nutzung von Web Services im Wege.

Ein UDDI-Verzeichnis enthält nach [Mel08] vier Haupttabellen. Die *White Pages* beschreiben die Diensteanbieter, die *Yellow Pages* teilen die Anbieter in Kategorien und die *Green Pages* erlauben eine Suche in den Beschreibungen der Dienste für die Anwender. Die vierte Haupttabelle stellt die *Service Type Registration* dar. Sie ist das Gegenstück zu den *Green Pages* und enthält deren Informationen in einer maschinenlesbaren Form, die von Anwendungen durchsucht werden kann. Die letzten beiden Tabellen enthalten dabei jeweils Querverweise aufeinander.

2.3 XML Web Services

Oftmals verbindet man Service-orientierten Architekturen mit Web Services, was jedoch nicht ganz richtig ist. Eine SOA stellt, wie im vorherigen Abschnitt beschrieben, lediglich ein abstraktes Konzept einer Architektur dar. Web Services hingegen sind eine mögliche konkrete Implementierung einer solchen Architektur.

Was einen solchen genau definiert diskutiert die *W3C Web Services Architecture Working Group* seit mehreren Jahren, die aktuelle Variante aus der Arbeitsgruppe ist die folgende (s. [WCL⁺05], S. 31):

“A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”

Diese Definition beschreibt die drei wichtigsten Merkmale eines Web Services. Er kommuniziert immer mit einem anderen Computersystem; in dieser Art der Kommunikation spielt der Mensch keine Rolle. Dieser tritt möglicherweise als Initiator der Interaktion auf, kann aber - bis auf die Wahl des Kommunikationspartners - keinen Einfluss darauf nehmen. In diesem Zusammenhang spricht man auch vom *application-centric web* (siehe [Cer02], S.6), da hier die Anwendungen im Vordergrund stehen.

Das zweite Merkmal ist die Beschreibung der Dienstleistungen in einem maschinenlesbaren Format, wobei sich hier die *web service description language* (WSDL) etabliert hat. Diese XML-basierte Beschreibungssprache beinhaltet alle oder ein Teil der angebotenen Dienste eines Services.

Schließlich stellt das Transportprotokoll SOAP ein drittes Merkmal der Web Services dar. Dieses beschreibt das XML-basierte Nachrichtenformat, welches zur Kommunikation genutzt wird; weiterhin definiert SOAP auch die Transportart der Nachrichten.

Die Möglichkeit zur einheitlichen Beschreibung von verteilten Funktionen in einem Netzwerk adressiert eine wichtige Anforderung innerhalb von Unternehmen. Bisherige Ansätze wie Microsoft's *Distributed Component Object Model* oder die *Common Object Request Broker Architecture* erfüllen zwar grundsätzlich die gleiche Aufgabe, liefern aber

keinen so hohen Grad an loser Kopplung. Weiterhin bieten Web Services den Unternehmen den Vorteil, Technologie mit offenen Standards zu nutzen. Das W3C¹ sowie OASIS² sind global agierende Konsortien, die sich für die Entwicklung und Ratifikation offener Standards einsetzen; weiterhin aber auch Anschluss an die Bedürfnisse der Industrie pflegen.

Die XML Web Services entstanden Ende 2000 mit der Standardisierung von *XML messaging* (SOAP und WSDL 1.1) sowie der ersten Version von des Verzeichnisdienstes UDDI und boten einen einheitlichen, von der Industrie anerkannten Standard zur Interoperation von verteilten Softwarekomponenten.

2.3.1 WSDL

Die Web Service Description Language (WSDL) beschreibt die Schnittstellen von Web Services in Form eines XML-Dokuments. Entwickelt von IBM und Microsoft liegt sie aktuell in der Form einer *W3C Recommendation* als Version 2.0 vor. Ein WSDL-Dokument beschreibt einen Dienst sowohl auf funktionaler als auch auf technischer Ebene. Die funktionale Beschreibung enthält abstrakte Informationen über den Dienst, die technische Ebene konkretisiert die Verbindung zu diesem Dienst.

Listing 2.1: Beispiel WSDL 2.0 Dokument

```
1 <description xmlns="http://www.w3.org/ns/wsd1"
2   targetNamespace="http://example.com/wsd1/HelloWorld"
3   xmlns:tnss="http://example.com/schemas/HelloWorld"
4   xmlns:soap="http://www.w3.org/ns/wsd1/soap">
5
6 <documentation> This document describes a sample service,
7 which always responses with "Hello world."</documentation>
8
9 <types>
10  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
11    targetNamespace="http://example.com/schemas/HelloWorld"
12    xmlns="http://example.com/schemas/HelloWorld">
13    <xs:element name="helloWorldResponse" type="xs:string"/>
14  </xs:schema>
15 </types>
16
17 <interface name="helloWorldInterface">
18  <operation namde="opHelloWorld"
19    pattern="http://www.w3.org/2004/03/wsd1/out-only">
20    <output messageLabel="Out" element="tnss:helloWorldResponse" />
21  </operation>
22 </interface>
```

¹World Wide Web Consortium, <http://www.w3.org>

²Organization for the Advancement of Structured Information Standards, <http://www.oasis-open.org>

```
23
24 <binding name="helloWorldSOAPBinding"
25     interface="tnss:helloWorldInterface"
26     type="http://www.w3.org/ns/wsdl/soap"
27     soap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
28     <operation ref="tnss:opHelloWorld"
29         soap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
30 </binding>
31
32 <service name="helloWorldService"
33     interface="tnss:helloWorldInterface">
34     <endpoint name="helloWorldEndpoint"
35         binding="tnss:helloWorldSOAPBinding"
36         address="http://example.com/HelloWorld" />
37 </service>
38 </description>
```

Listing 2.1 zeigt ein Beispiel für ein solches WSDL Dokument: Zeile 6 - 7, enthält eine textuelle Beschreibung der angebotenen Funktionen und ist optional. Das Kinder des Elements `types`, Zeile 9, definieren die Datentypen, die bei Ein- und Ausgabe der Nachrichten verwendet werden. In dem Beispiel gibt es nur den Typ der `helloWorldResponse` als String. Neben einfachen sind auch komplexe Typen möglich, die sich aus mehreren Feldern mit jeweils eigenen Typen zusammensetzen.

Die eigentliche Funktionalität des Web Services wird in dem Element `interface`, Zeile 17, beschrieben. Dieser Abschnitt definiert alle Operationen des Dienstes einschließlich der Ein- und Ausgabeparamter und den Antworten im Fehlerfall (im Beispiel nicht angegeben). Die Beschreibung der Operationen erfolgt unabhängig vom Transportprotokoll (HTTP, SMTP, ...) sowie der Kodierung der XML-Nachrichten. Jede Operation ist mit einem *Message Exchange Pattern* (MEP) verknüpft, welches beispielsweise die Reihenfolge der Nachrichten festlegt. In Zeile 18 wird die Operation `opHelloWorld` beschrieben, die nur eine Antwort sendet (MEP ist hier *out-only*). Weitere MEP sind beispielsweise *in-only*, *in-out* und *in-optional-out* (siehe [GHM⁺07]).

Das `binding`-Element gehört zur technischen Beschreibung des Web Services. Es legt fest, über welches Transportprotokoll die vorher abstrakt definierten Funktionen aus dem `interface`-Element übertragen werden. Weiterhin sind detaillierte Informationen zu jeder Operation möglich, beispielsweise die Kodierung der Nachricht in Unicode oder zusätzliche Informationen zum Transport.

Der physikalische Speicherort des Web Services wird abschließend in dem `service`-Element definiert. Es legt für jede Schnittstelle eines Dienstes dessen Zugangspunkt (*endpoint*) fest, wobei auch mehrere Zugangspunkte für einen Dienst möglich sind.

Durch diese Angaben kann ein Nutzer nur über das WSDL-Dokument einen Dienst lokalisieren, seine Funktionalitäten in Erfahrung bringen und über die Zugangspunktdefinitionen mit dem Dienst kommunizieren. Die Informationen zu Schnittstellen und

Mechanismen zur Kontaktaufnahme sind hier in einem Dokument vereint, was gegenüber anderen Technologien (CORBA, J2EE) ein Vorteil darstellen kann. Die Beschreibungen können jedoch auch getrennt werden, sodass eine große Flexibilität bei der Anwendungsentwicklung möglich ist.

2.4 SOAP

Während Verzeichnisdienste ein grundlegendes Merkmal einer Service-orientierten Architektur darstellen und Web Services bereits eine konkrete Art der Implementierung bilden, beschreibt SOAP - unabhängig von Programmiersprache oder Plattformen - die Kommunikation der Web Services. Hierzu gehört sowohl das XML-basierte Nachrichtenformat als auch der Transport über ein fast beliebiges Protokoll, daher stand SOAP ehemals für *Simple Object Access Protocol*. Während des Standardisierungsprozesses durch das W3C wurde allerdings klar, dass SOAP weder einfach noch für Objektaufrufe geeignet war. Bis dahin hatte der Begriff SOAP jedoch bereits eine gewisse Verbreitung erreicht. Der aktuelle Bezeichner ist daher kein Akronym mehr, sondern - auch aufgrund mangelnder Alternativen - der Name der Spezifikation zur Kommunikation mit Diensten mittels Web Services. Die aktuelle Version 1.2 (siehe [GHM⁺07]) vom 27. April 2007 der Spezifikation für SOAP ist in vier verschiedene Teile gegliedert. Der erste Teil (*Part 0: Primer*) ist eine Art technische Einführung und versucht mithilfe zahlreicher Beispiele einen Leitfaden in die Spezifikation zu geben. Der zweite Teil (*Part 1: Messaging Framework*) beschreibt den Rahmen und die Elemente einer Nachricht und gibt vor, wie die Nachrichten zu übermitteln sind. Im dritten Teil (*Part 2: Adjuncts*) geht es um das Datenmodell, um Kodierungsschemata sowie entfernte Methodenaufrufe (*remote procedure calls*). Zusätzlich zeigt dieser Teil die Anbindung an ein Transportprotokoll am Beispiel des HTTP-Protokolls. Neben HTTP sind viele weitere Protokolle denkbar, unter anderem über Mails (SMTP), per FTP sowie über die Java Messaging Services (JMS).

Listing 2.2: Struktur einer SOAP-Nachricht

```
1 <env:Envelope
2   xmlns:env="http://www.w3.org/2003/05/soap-envelope">
3   <env:Header>
4     ...
5   </env:Header>
6   <env:Body>
7     ...
8   </env:Body>
9 </env:Envelope>
```

In Listing 2.2 ist der grundsätzliche Aufbau einer SOAP-Nachricht nach Teil 2 (*Messaging Framework*) der Spezifikation dargestellt. Der SOAP Envelope, sprichwörtlich der Umschlag der Nachricht, bildet den Wurzelknoten des Dokuments und enthält die Ele-

mente Header und Body. Der Parameter *xmlns* spezifiziert mit der URI-Angabe die Version des Umschlags und bindet das Schema an das Präfix *env*.

Der Header ist ein optionales Element der Nachricht, dessen Inhalt nicht durch die SOAP-Spezifikation vorgeschrieben ist; sie hält hier lediglich Platz für weitere Elemente frei. Vorstellbar sind hier Erweiterungen bezüglich der Sicherheit von Nachrichten durch Verschlüsselung oder Signaturen, Informationen über Zwischenstationen auf dem Weg zum Empfänger, Quality of Service Definitionen und andere Informationen. Der Header enthält - falls angegeben - eine oder mehrere Blöcke, die jeweils an die entsprechenden Stationen auf der Strecke zum Empfänger adressiert sein können. Analog sind auch mehr als ein Block mit unterschiedlichen Informationen für einen einzigen Empfänger erlaubt.

Die eigentlichen Nutzdaten der Nachricht befinden sich im Body. Dieser ist zwingend erforderlich und kann prinzipiell alles enthalten, was sich als XML darstellen lässt; von einer HTML-Seite über Rechnungen bis hin zu kompletten PDF-Dokumenten. Auch der Body besteht - wie der Header - aus einem oder mehreren Blöcken und kann seine eigenen Namespaces einbinden.

Im Falle eines Fehlers beim Abarbeiten der SOAP-Nachricht generiert der entsprechende Knoten eine SOAP-Fault Nachricht, die genauere Informationen über Fehlercode und Grund enthält. Der Fehlerbehandlung, der Strukturierung der Fault-Nachricht sowie der benötigten Elemente in dieser Antwort ist in der Spezifikation ein ganzes Kapitel gewidmet ([GHM⁺07], Kapitel 5.4).

Die Spezifikation gibt lediglich den Envelope sowie Header- und Body- Element vor. Die Kinderknoten letzterer sind nicht festgelegt, da sie entsprechend den Anforderungen der Applikationen entworfen werden. Allerdings definiert die SOAP-Spezifikation die Art und Weise, wie beteiligte Systeme die Nachrichten verarbeiten und wie sie im Fehlerfall reagieren sollen.

Weiterhin beschreibt das SOAP-Kommunikationsprotokoll, wie die Nachrichten formatiert sein müssen, jedoch fehlt jeglicher Mechanismus zur Koordination des Nachrichtenflusses. Die Art der Kommunikation lässt sich mit Nachrichten in einem Rohrpostsystem vergleichen, bei dem jeweils ein Rohr vom Sender zum Empfänger führt. Damit diese Nachrichten über Zwischenstationen geleitet werden können, die nicht nur eine genaue starre Verbindung zum endgültigem Empfänger aufweisen, benötigt es einer spezifischen Möglichkeit zur Adressierung.

Im Sinne der Kommunikation über das HTTP-Protokoll besteht die Adressierung bei "puren" SOAP-Nachrichten in dem Verbindungsaufbau des Senders zum Empfänger über einen Request mit der URL des Empfängers. Neben der URL ist jedoch auch die empfangende Klasse zu referenzieren, die den benötigten Dienst zur Verfügung stellt. In der Literatur wird der vollqualifizierte Klassenname als "Service-Endpoint" bezeichnet (vgl. [Mel08], S. 94), bei Java als Beispiel bestehend aus Klassennamen sowie dem vollständigen Paketnamen. Für die Klasse *Echo* im Paket *test* der als Beispiel dienenden Domain *example.com* wäre ein solcher Service-Endpoint also <http://example.com/test/Echo>.

2.4.1 WS-Addressing

Das *Hyper Text Transfer Protocol* ist gerade im Internet-Umfeld weit verbreitet und kommt auch oft bei SOAP-Nachrichten zum Einsatz, da es mit Firewalls gut harmoniert. Bei komplexeren Strukturen, beispielsweise bei der Kommunikation über Unternehmensgrenzen hinaus, sollte nicht vernachlässigt werden, dass innerhalb der Unternehmen andere Transportprotokolle zum Einsatz kommen können. Die SOAP-Nachrichten werden also immer öfters über mehrere Protokolle transferiert, bevor sie ihren Endpunkt erreichen. Wenn zwei Firmen kooperieren, kann bei der Softwareentwicklung auf einer Seite nicht bestimmt werden, welche Protokolle auf der anderen Seite zum Einsatz kommen. Bei der HTTP-Kommunikation ist die Frage der Adressierung durch die Mechanismen von HTTP sichergestellt; dieser ist aber nicht für die weitere Zustellung über andere Transportwege (JMS, SMTP oder auch reines TCP) geeignet.

Die 2006 vom W3C standardisierte Erweiterung des SOAP-Headers WS-Addressing erfüllt genau diesen Zweck. Sie definiert eine universelle Darstellung der Absender- und Empfängerinformationen, die unabhängig vom Protokoll in die Nachricht eingebettet werden können (vgl. [ST07], S. 494).

Ein weiterer, aus Anwendungssicht möglicherweise als Nachteil empfundener Grund liegt in der Natur der Web Services, die keinen Zustand erlauben. Wenn sich während einer Konversation mehrere Nachrichten aufeinander beziehen sollen, kann dies entweder anwendungsseitig realisiert werden, oder aber in einer Erweiterung der Header-Elemente der SOAP-Nachrichten. Letzteres erreicht WS-Addressing durch eine eindeutige Nachrichtenennung sowie die Kennung der Nachricht, auf welche die aktuelle Bezug nimmt.

Listing 2.3: WS-Addressing

```
1 <env:Envelope xmlns:env="..."
2   xmlns:wsa="http://schemas.xml.org/ws/2005/08/addressing">
3   <env:Header>
4     <wsa:MessageID>
5       uuid:aaaabbbb-cccc-dddd-eeee-ffffffffffffff
6     </wsa:MessageID>
7     <wsa:To>
8       http://example.com/sampleRequestAddress
9     </wsa:To>
10    <wsa:ReplyTo>
11      <wsa:Address>
12        http://example.com/sampleReplyAddress
13      </wsa:Address>
14    </wsa:ReplyTo>
15    <wsa:Action>
16      http://example.com/sampleOperation
17    </wsa:Action>
```

```
18 </env:Header>
19 <env:Body> ... </env:Body>
20 </env:Envelope>
```

Das Listing 2.3 zeigt eine mögliche Anwendung von WS-Addressing Informationen in einer beliebigen SOAP-Nachricht. Das im Umschlag oder im Header definierte Namespace-Element *wsa* bezieht sich hier die aktuelle Verabschiedung der W3C von 2006. Das Element *MessageID* enthält die Nachrichtenennung, auf die sich Antworten mittels des Elements *RelatesTo* beziehen können. Die Elemente *To* und *ReplyTo* spezifizieren die Adresse, an welche die Nachricht einerseits versandt wird (*To*) sowie den Empfänger der Antwort (*ReplyTo*).

Bei den Elementen, die sich auf Adressen beziehen, handelt es sich um Endpunktreferenzen. Diese können optional neben der Adresse weitere Daten wie Parameter und Metainformationen enthalten. Die Parameter können Informationen enthalten, die bei der Verarbeitung am Endpunkt relevant sind. In den Metainformationen kann beispielsweise das Verhalten des Endpunktes beschrieben werden, weiterhin sind hier auch Erweiterungen möglich (beispielsweise Policies).

Die Intention der Nachricht wird durch das Element *Action* angezeigt, wenn diese nicht aus der Nachricht selbst ersichtlich ist. Ein Beispiel hierfür wäre ein einfaches Bestellsystem, wobei im Nutzlastteil der Nachricht die Warenkorb-Identifikation übertragen wird. Je nach aufgerufener Aktion ist mit der selben Nutzlast (die ID) der Warenkorb entweder abzusenden oder zu stornieren.

2.5 Enterprise Service Bus

Der Enterprise Service Bus ist ein Middleware-Produkt, der eine vermittelnde Rolle zwischen den kommunizierenden Systemen einnimmt. Zu den wichtigsten Merkmalen gehören die Transformation von Nachrichten, die Integration von bestehenden Anwendungen als auch das Routing von Nachrichten zwischen den angebundenen Systemen.

Bei der Kommunikation möglicherweise verschiedener Plattformen mit unterschiedlichen Programmiersprachen kann es zu Problemen durch inkompatible Datentypen kommen. Dies bezieht sich sowohl auf einfache Differenzen wie unterschiedliche Zahlendarstellungen (in 32- und 64 Bit) sowie verschiedenen Datentypen bis hin zur kompletten Umformung von ganzen Schemata, beispielsweise Adressdaten. Die Aufgabe des ESB ist hier die Transformation der Anfragen in eine Form, die das Zielsystem versteht. Dieses muss nicht notwendigerweise auch ein Web Service sein, sondern es kann - im Zuge der Umsetzung alter Systeme auf neue, Service-basierte Systeme - sich auch um eine proprietäre Anwendung mit entsprechenden Konnektoren an den ESB handeln.

Diese Integration bestehender Anwendungen wird also auch durch den auf Datenfluss ausgelegten Bus ermöglicht. Sie bietet den IT Managern einer Organisation eine hohe Flexibilität bei der Auswahl der zu verwendenden Softwaresysteme, um eine mög-

lichst optimale Lösung für ein bestimmtes Problem zu erarbeiten. Dieses Aufgabenfeld der Integration verschiedener Softwaresysteme - auch Enterprise Application Integration genannt - bietet auch Vorteile, wenn externe Lösungen hinzugezogen werden sollen.

Ein im Sinne von loser Kopplung und Service-orientierten Architekturen sehr wichtiger Punkt am Service Bus ist die Entkopplung der logischen Nachrichten von der Art und Weise, wie und wohin sie übertragen werden. Führt man in einem bestehenden oder einem neuen System eine SOA ein, so erhält man - vereinfachend ausgedrückt - erstmal eine Menge von Services. Die Kommunikation der Services wird durch Konfigurationsdateien an den einzelnen Endpunkten festgelegt, jeder Service kennt also seine Gesprächspartner.

In diesem Szenario ist nun vorstellbar, dass die Dienste über ein Cluster von Systemen angeboten werden, welches jedoch durch einen Hardwaredefekt unbrauchbar wird. Die Services werden nun auf ein neues System umgezogen, jedoch müssen auf einer möglicherweise sehr großen Zahl von Klienten Konfigurationsdateien ausgetauscht werden. Die lose Kopplung der Services verhindert an diesem Punkt zumindest ein Rollout einer neuen Softwareversion. Ein Enterprise Service Bus stellt hier eine große Erleichterung dar, wenn die Systeme nur noch auf Anwendungsebene Nachrichten austauschen. Die Applikationen werden nicht mehr gegen spezifische Endpunkte gebunden, sondern senden ihre Nachrichten an einen logischen Endpunkt; die Infrastruktur kümmert sich dann um die physische Zustellung der Nachrichten. Eine solche Infrastruktur entspricht dann einer Bus-Architektur, wie sie ein ESB bietet (vgl. [ST07], S. 486).

Die Routingfunktion des Enterprise Service Busses bezeichnet die Möglichkeit, virtuelle Kommunikationskanäle zwischen einzelnen Anwendungen zu schaffen. Dies kann innerhalb einer Organisation oder sogar über deren Grenzen hinaus geschehen, sodass verschiedene Firmen über einen oder mehrere Busse Nachrichten austauschen können. Diese Komponente ähnelt stark der bisherigen nachrichtenorientierten Middleware (*Message-oriented Middleware*), jedoch wird die starre Schnittstelle durch eine flexible ersetzt. Wie ein Netzwerkschwitch oder das E-Mailsystem ist der ESB in der Lage, Nachrichten aufgrund bestimmter Merkmale intelligent an die entsprechenden Empfänger weiterzuleiten, ohne andere Teilsysteme zu beeinflussen. Bei einfachen *Remote Procedure Call*-artigen Systemen ist für jeden Knoten jeweils eine Punkt-zu-Punkt Verbindung zu den anderen Knoten notwendig, also im ungünstigsten Fall benötigt man $n(n - 1)/2$ Kanäle. Mit einem Enterprise Service Bus reduziert sich die Anzahl auf n , da jeder Knoten nur mit dem ESB verbunden ist (vgl. [Mel08], S. 24).

Es gibt mittlerweile eine Vielzahl an verschiedenen Service Bus-Produkten, sowohl von namhaften Herstellern für viel Geld als auch als Open-Source-Lösungen. In dieser Arbeit wird der Service Bus von WSO2³ verwendet, der durch das laufende EoS (*Enforcement of steps - supervising task execution*⁴) der Universität Hamburg bereits vor Beginn der Arbeit ausgewählt wurde.

³<http://wso2.org>

⁴<http://vsis-www.informatik.uni-hamburg.de/projects/eos/>

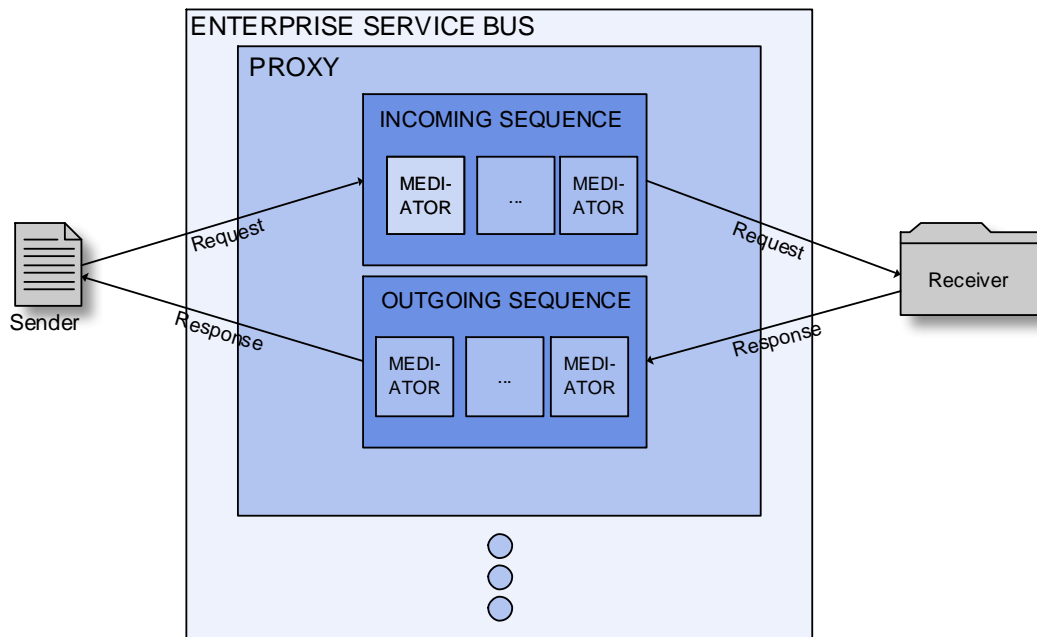


Abbildung 2.3: Schematische Darstellung des Enterprise Service Busses (WSO2)

Der verwendete WSO2 Enterprise Service Bus basiert auf dem Apache Synapse Enterprise Service Bus⁵ und lässt sich über die *Synapse Configuration Language*⁶ konfigurieren. Diese deskriptive Konfiguration besteht aus Endpunkten, Sequenzen, Aufgaben und Proxy Services.

Abbildung 2.3 stellt die Komponenten des ESB schematisch dar. Ein Proxy Service definiert einen virtuellen Service, der Anforderungen (*requests*) akzeptiert, vermittelt und an den adressierten Endpunkt übergibt. Beim Vermitteln kann der Proxy Service eine Nachricht transformieren und so mit einer anderen Schnittstelle versehen.

Diese Transformation geschieht über die Sequenzen. Eine Sequenz besteht dabei aus einem oder mehreren Mediatoren und einem Endpunkt. Ein Endpunkt (*endpoint reference*) beschreibt eine spezifische Zieladresse für eine Nachricht. Dies kann entweder als URI, als WSDL Endpunkt sowie als Failover- oder Loadbalance Gruppe geschehen.

Die Mediatoren eines Enterprise Service Busses schließlich stellen die Werkzeuge zur Transformation und Vermittlung von Nachrichten dar.

“Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently” (vgl. [GHJV04], S. 273)

Die angegebene Definition beschreibt die allgemeine Funktion eines Mediators. Durch die Trennung der kommunizierenden Instanzen ermöglicht er ein hohes Maß an looser Kopplung. Im Zusammenhang mit dem Enterprise Service Bus jedoch fungieren die Proxy Services als trennende Instanz, während die Mediatoren Komponenten dieser Dien-

⁵<http://synapse.apache.org>

⁶http://synapse.apache.org/Synapse_Configuration_Language.html

ste sind.

Listing 2.4: Grundgerüst eines Mediators in Java

```
1 import org.apache.synapse.ManagedLifecycle;
2 import org.apache.synapse.MessageContext;
3 import org.apache.synapse.core.SynapseEnvironment;
4 import org.apache.synapse.mediators.AbstractMediator;
5
6 public class SimpleMediator
7     extends AbstractMediator
8     implements ManagedLifecycle {
9     public void destroy() {
10    }
11
12    public void init(SynapseEnvironment arg0) {
13    }
14
15    public boolean mediate(MessageContext arg0) {
16        return true;
17    }
18 }
```

Ein leerer Mediator ist in Listing 2.4 gezeigt. Jeder Mediator muss dabei das Interface *ManagesLifecycle* implementieren und von der abstrakten Klasse *AbstractMediator* erben, beide liegen im Paket *org.apache.synapse*. Der *AbstractMediator* bietet außerdem Funktionen zum Tracing und Logging an.

Die Funktion *init(...)* wird beim Starten des Enterprise Service Busses aufgerufen und erhält als Argument eine Instanz der *SynapseEnvironment*-Klasse, welche Zugriff auf die ausführende SOAP-Engine erlaubt. Der Zugriff ist unter anderem für das Erstellen und Senden von Nachrichten über Synapse zuständig. Analog ruft der ESB die *destroy(...)*-Methode beim Herunterfahren auf, sodass die Mediatoren offene Ressourcen geordnet schließen können.

Die *mediate(...)*-Funktion bildet den Kern des Mediators. Diese Operation muss implizit durch die Schnittstelle *Mediator* implementiert werden, welches durch die Basisklasse eingebunden wird. Als Argument erhält sie ein Objekt vom Typ *MessageContext*, welches unter anderem Zugriff auf die eigentliche SOAP-Nachricht, auf Informationen zum Routing (Absender und Empfänger) als auch auf weitere Eigenschaften der Nachricht gewährt.

Die Mediatoren spielen im Laufe der Arbeit eine wichtige Rolle. Einerseits sind sie für die Überwachung der Nachrichten durch den im ersten Kapitel angesprochen Protokollierungsservice verantwortlich, andererseits erlauben sie eine Modifikation der Routingfunktion.

3 Anforderungsanalyse

In diesem Kapitel sollen die Anforderungen für die Überwachung asynchronen Nachrichtenaustauschs über den Enterprise Service Bus ermittelt werden.

Der erste Abschnitt führt ein Beispiel ein und erläutert dessen Kontext. An diesem lässt sich danach die Problemstellung der Arbeit verdeutlichen. Weiterhin ist eine Betrachtung des Begriffs Asynchronität in Bezug auf Nachrichten notwendig, eine Zusammenfassung der Eigenschaften eines Service Busses sowie eine technische Konkretisierung der Anforderungen schließen das Kapitel ab.

3.1 EPEJ-Beispiel

Das im Folgenden motivierte Beispiel stammt aus der Fallstudie WP3-CS2 des R4eGov¹-Projektes ([EECB07]) der Europäischen Union (nachfolgend EPEJ-Beispiel genannt), an dem auch die Universität Hamburg beteiligt ist. Diese Fallstudie behandelt die sichere Verbindung der IT-Systeme von Europol und Eurojust, den Eckpfeilern der europäischen Justiz - und Rechtsdurchsetzung. Auch wenn ihre Arbeitsweisen stark divergieren, besteht ein großer Bedarf an einer Kollaboration der beiden Behörden. Trotz der geklärten rechtlichen Lage sowie der Einrichtung einer gesicherten Kommunikationsverbindung zwischen beiden ist es bisher aufgrund verschiedener Implementierungen von Verfahren und Richtlinien der 27 Mitgliedsstaaten nur zu einer geringen Anzahl von gemeinsamen Fällen gekommen. Aus technischer Sicht gibt es bereits die beiden Computergestützte Systeme OASIS (Europol, *Overall Analysis System for Intelligence and Support*) sowie CMS (Eurojust, *Case Management System*), jedoch arbeiten diese Computersysteme getrennt. Das R4eGov-Projekt erarbeitet eine verbindende Architektur für internationale und nationale Behörden. Diese Fallstudie beschränkt sich auf die Interaktionen zwischen Europol, Eurojust und den betroffenen Behörden in Belgien.

Das EPEJ-Beispiel beschreibt einen Fall, bei dem der Zoll im spanischen Hafen bei der Routineüberprüfung in einer Ladung Kaffee eingeschmuggeltes Kokain findet. Der Container stammt aus Venezuela und ist ab Spanien zum Transport per LKW nach Belgien vorgesehen. Der spanische Staatsanwalt erachtet den Drogenhandel als von europäischem Ausmaße und schaltet daher die Eurojust ein. Hierfür schreibt er ein Ersuchen an die spanische Eurojust-Stelle (EJNMa - Eurojust National Member, Land "a"), die einen Vorgang (Arbeitsdatei) in ihrem CMS erstellt (siehe Abbildung 3.1, Punkt 1) und ein Treffen einberuft. Dies Treffen findet zwischen den Eurojust-Zweigstellen der involvierten Nationen statt. Neben weiteren Punkten wird beschlossen, zusätzliche Informationen über den Absender durch einen Verbindungsoffizier bei Europol (*Europol Liaison Officer*,

¹<http://www.r4egov.eu>

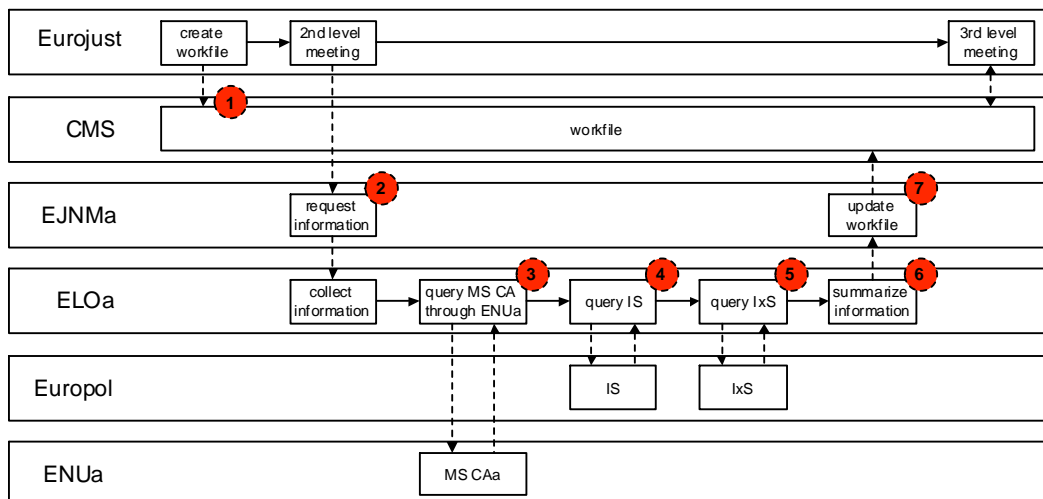


Abbildung 3.1: Schematischer Ablauf des EPEJ-Beispiels

ELO) einzuholen (Abbildung 3.1, Punkt 2). Die Informationen sind in einem nationalen System gespeichert, dem *Member State's Competent Authority system* (MS CAa), der Zugriff erfolgt über die entsprechende nationale Europol Einheit (ENU, *Europol National Unit*, Abbildung 3.1, Punkt 3). Nach Abruf der Informationen aus dem MS CA System erhält sie der Verbindungsoffizier; weiterhin sucht er in den Informations- und Indexsystemen bei Europol (Abbildung 3.1, Punkte 4 und 5). Die Informations- und Indexsysteme sind Europol-eigenen Datenbanksysteme, werden über das InfoEx-System abgefragt und sie sind im Gegensatz zu den MS CA-Systemen international. Die gesammelten Daten fasst der Verbindungsoffizier zusammen (Abbildung 3.1, Punkt 6) und leitet sie an die anfragende Zweigstelle von Eurojust weiter. Der EJMNa aktualisiert mit diesen Daten anschließend die in Abbildung 3.1, Punkt 1 erstellte Arbeitsdatei im CMS (Abbildung 3.1, Punkt 7) und gibt allen an der Untersuchung beteiligten EJMNs Zugriff darauf. Abschließend gibt es ein weiteres Treffen, um die Informationen auszuwerten und weitere Schritte in der Strafsache einzuleiten.

3.2 Protokollierung der Kommunikation

Um festzustellen, dass jeder Schritt des oben beschriebenen Ablaufs durchgeführt wurde und um den gesamten Ablauf nachzuvollziehen, bedarf es einer Überwachung der Kommunikation. Hierfür gibt es generell verschiedene Ansätze. Geht die Kommunikation jedoch über die Grenzen der jeweiligen Organisationen hinaus, so muss das Monitoring extern erfolgen, da eine Modifikation innerhalb der Organisationen aus Sicherheitsgründen nicht möglich ist (siehe [RR09]). Der Grund hierfür ist die Autonomie der involvierten Teilnehmer. Es ist zwar möglich, ihnen ein Kommunikationsprotokoll vorzuschreiben, jedoch kann nicht bestimmt werden, dass sie hierfür alle Nachrichten an einen bestimmten Punkt senden müssen. Die Überwachung muss über das von aussen

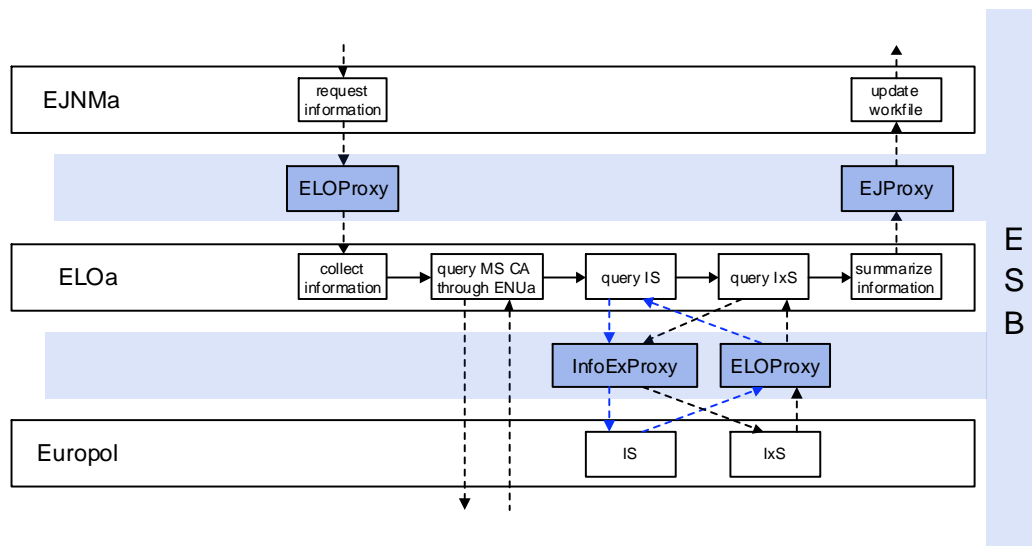


Abbildung 3.2: Kommunikation im EPEJ-Beispiel über einen ESB (Ausschnitt)

beobachtbare Verhalten geschehen, also mittels der durch das Internet ausgetauschten Nachrichten.

Im Projekt EoS entschied man sich für das Monitoring mittels eines Mediators für den Enterprise Service Bus. Der Bus stellt für jeden Dienst einen Proxy bereit, über den die Kommunikation dann stattfindet. Für einen Ausschnitt des Beispiels ist dies in Abbildung 3.2 dargestellt. Innerhalb der Proxy-Dienste protokolliert der *wslog*-Mediator die Nachrichten. Damit auch bei vielen nebenläufigen Aktionen die verschiedenen Sequenzen Konversationen zuweisbar sind, enthalten die SOAP-Nachrichten eine Kennung für die einzelnen Rollen der Dienste als auch für die Konversation selbst. Die entsprechende Erweiterung der SOAP-Nachricht zeigt das Code-Beispiel 3.1.

Listing 3.1: Erweiterung des SOAP-Headers durch das ChorMon-Framework

```

1 <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
2   <s:Header>
3     <vsis:chormon xmlns:vsis="http://vsis.de/choreography/monitoring">
4       <fromRole>ENURoleType</fromRole>
5       <toRole>ELORoleType</toRole>
6       <chorIdentifier>
7         874408ed-4cbf-46c5-8c73-31c7d1c9b97e
8       </chorIdentifier>
9     </vsis:chormon>
10  </s:Header>
11  <s:Body> ... </s:Body>
12 </s:Envelope>

```

Über die eindeutige Kennung der Choreographie *chorIdentifier* lässt sich eine Nachricht eindeutig einer bestimmten Kommunikation zuweisen, ausführliche Informationen über das ChorMon-Framework gibt [RR09].

3.3 Asynchronität auf Anwendungsebene

Die meisten Anfragen innerhalb des Beispiels erhalten sofort eine Antwort, nicht so jedoch die Anfrage des EJNM an den ELO. Bei der Modellierung des Systems bietet sich daher an dieser Stelle die Verwendung asynchroner Kommunikationsstrukturen an.

Der Begriff "asynchrone Kommunikation" beim Austausch von Nachrichten kann auf zwei verschiedenen Ebenen betrachtet werden: Sowohl auf der Transportebene als auch auf der Anwendungsebene.

Asynchronität auf der Transportebene bezieht sich auf die Kommunikation über einen virtuellen Nachrichtenkanal mittels Paketen des zugrunde liegenden Transportprotokolls (beispielsweise TCP/IP). Bei synchroner Kommunikation erfolgt der Austausch von Anfrage und Antwort dabei über denselben Kanal, in diesem Sinne beziehen sich alle Pakete aufeinander. Dies lässt sich mit einem Telefonat vergleichen, bei dem der Anrufende von seinem Gesprächspartner fast unverzüglich die Antwort auf seine Frage erhält. Bei der asynchronen Kommunikation verhält es sich anders. Es werden zwei Nachrichtenkanäle verwendet - im übertragenen Sinne übermittelt der Anrufende also eine Rückrufnummer und wartet danach vor dem Telefon auf den Rückruf.

Die typische Kommunikation zwischen einem Browser und einem Webserver geschieht in der Regel in einem Anfrage / Antwort - Muster, der Client erwartet also eine Antwort. Bei Web Services über SOAP sind hingegen auch Interaktionen vorstellbar, wo der Client lediglich eine Anfrage sendet oder nur eine Antwort ohne vorherige Anfrage abrufen. Dies Muster wird durch die bereits erwähnten *Message Exchange Patterns* (MEPs) spezifiziert.

Für asynchrone SOAP-Nachrichten kommen die angesprochenen WS-Addressing Headers zum Einsatz. Die Nachrichten erhalten eine Absender-, Antwort- sowie Empfängeradresse und werden entsprechend behandelt. Die Großzahl der Enterprise Service Bus-Produkte unterstützt die asynchrone Kommunikation auf Transportebene, sowohl zwischen Client und ESB als auch zwischen ESB und Empfänger. Bei asynchroner Kommunikation zwischen Client und ESB, aber synchroner Kommunikation zwischen ESB und Empfänger entfernt der Bus die WS-Addressing Header und fügt sie auf dem Rückweg wieder ein. Bei asynchroner Kommunikation zwischen ESB und Empfänger werden die WS-Addressing Header ausgetauscht (oder hinzugefügt), entsprechend bei der Zustellung der Antwort ein weiteres Mal.

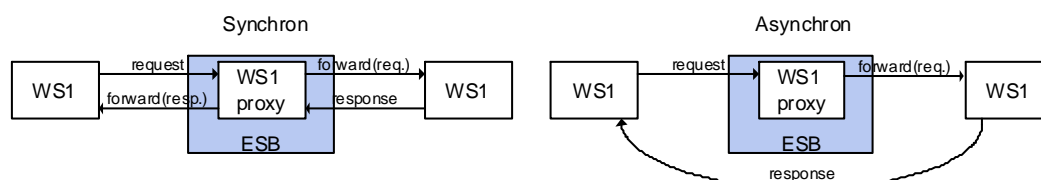


Abbildung 3.3: Synchroner und asynchroner Kommunikation über den ESB

Die zweite Ebene ist die der Anwendung. Vergleichbar mit dem System der Deutschen Post sendet der Klient eine Nachricht (einen Brief) und "vergisst" den Vorgang erstmal.

Hat der Empfänger den Brief empfangen, kann er die Nachricht verarbeiten, weitere Nachrichten an andere verschicken, Informationen zusammenstellen und irgendwann einmal auf den Brief antworten. Der Vorgang wiederholt sich daraufhin mit vertauschten Rollen. Die beiden Parteien stellen hierbei eine generische Schnittstelle zum Empfang von Nachrichten bereit (Briefkasten) und müssen nicht - wie bei der asynchronen Kommunikation auf Transportebene - eine spezielle Endstelle für diese eine Nachricht offenhalten. Bezogen auf SOAP werden hier also jeweils nur Anfragen ausgetauscht, die Antwort ist eine neue Anfrage.

Die Abbildung 3.3 zeigt den Unterschied zwischen der synchronen und asynchronen Kommunikation bei Verwendung eines Enterprise Service Busses.

Listing 3.2: Falsche ReplyTo-Adresse im WS-Addressing Header

```
1 <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
2   <s:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
3     ...
4     <wsa:To>http://server:8080/axis2/services/ELOBehaviorService</wsa:To>
5     <wsa:ReplyTo>
6       <wsa:Address>http://www.w3.org/2005/08/addressing/none</wsa:Address>
7     </wsa:ReplyTo>
8     <wsa:MessageID>
9       urn:uuid:9D1B1A973C1A730B9112417121777596190332041283930
10    </wsa:MessageID>
11    <wsa:Action>
12      http://vsis.de/choreography/monitoring/example/requestInformation
13    </wsa:Action>
14  </s:Header>
15  <s:Body> ... </s:Body>
16 </s:Envelope>
```

Bei dieser Art des Nachrichtenaustausches stellt der ESB ein Problem dar, da er auf dem Weg zum Empfänger die WS-Addressing Header austauscht oder entfernt. Der Empfänger kann also nicht auf die Nachricht antworten, da er entweder keine Nachricht mit Adressinformationen empfängt oder aber eine anonyme Antwortadresse des ESB (siehe Listing 3.2, Zeile 6).

3.4 Anforderungen an einen ESB

Die bisher beobachteten Probleme sollen nun mit den bekannten Anforderungen aus [DKS07] an einen Enterprise Service Bus verglichen werden. Diese sind in der linken Spalte von Tabelle 3.1 gelistet. Die rechte Spalte zeigt, welche der Eigenschaften im Beispiel zugesichert werden.

Lose Kopplung, Mediation sowie Schematransformation hatte bereits Kapitel 2 erläutert. Bei der Serviceaggregation handelt es sich um die Fähigkeit, bei einer eingehenden

Tabelle 3.1: Eigenschaften eines Enterprise Service Busses (vgl. [DKS07])

Lose Kopplung	✓
Ortstransparenz	✗
Meditation	✓
Schematransformation	✓
Serviceaggregation	✓
Lastverteilung	✓
Sicherheit	✓
Monitoring	✓
Konfigurierbarkeit	✓

Anfrage eine ganze Reihe von Diensten aufzurufen, um die Anfrage zu beantworten. Die Lastverteilung ermöglicht es dem ESB, je nach Bedarf und Konfiguration Anfragen auf mehrere Dienstanbieter aufzuteilen. Die Sicherheitsthematik rund um den ESB behandelt hauptsächlich das Erzwingen bestimmter Sicherheitsvoraussetzungen für Anfragen sowie die Einhaltung von definierten Richtlinien. Hierzu kann beispielsweise die Verwendung von WS-Security gehören. Monitoring hingegen befasst sich mit der Leistungsanalyse sowie der Sicherstellung der Verfügbarkeit der angesprochenen Dienstanbieter. Kann ein bestimmter Dienst beispielsweise nicht in der vorgegebenen Zeit antworten, besteht im ESB die Möglichkeit, einem Administrator eine Nachricht zu senden. Moderne Systeme sollten weiterhin nicht fest programmiert sein, sondern einen gewissen Flexibilitätsgrad durch Konfigurationsdateien oder aufweisen. Im verwendeten WSO2 ESB wird dies durch die *Synapse Configuration Language* gewährleistet.

Die im Beispiel verletzte Anforderung stellt die Ortstransparenz dar. Ortstransparenz soll die physikalische Endstelle des verwendeten Dienstes vor dem Nutzer verbergen. Idealerweise versendet ein Nutzer nur eine Nachricht an die ESB Infrastruktur ohne den eigentlichen Empfänger der Nachricht zu kennen. Die Trennung vom eigentlichen Endpunkt ermöglicht es beispielsweise, Dienstanbieter nach Bedarf hinzuzufügen oder zu entfernen. Fällt beispielsweise ein System aus, so müssen nicht alle Klienten neu kompiliert werden.

Im Beispiel ist die Ortstransparenz nicht gegeben. Die Anfrage des EJM an den ELOa enthält den physikalischen Endpunkt des EJM als Antwortadresse, wodurch der Anbieter nicht mehr vom Nutzer getrennt ist. Sendet der ELOa dann seine Antwort an den EJM, geht diese am ESB vorbei und kann folglich nicht protokolliert werden. In der Praxis führt dieser Umstand allerdings dazu, dass die Nachricht keinen Empfänger erreicht. Dies liegt an der Art und Weise, wie der ESB mit den WS-Addressing-Headern umgeht. Eingehende Nachrichten werden hierbei je nach Einstellung entweder komplett vom WS-Addressing-Header befreit oder der ESB ersetzt diesen durch seinen eigenen Header. Da er hierbei von synchronem Austausch auf der Transportebene ausgeht, er-

setzt er die Antwortadresse durch einen anonymen Endpunkt, also durch eine nicht vorhandene Adresse. Der ELOa-Klient kann die Antwort also nicht abschicken und generiert eine Fehlermeldung in seinem System.

3.5 Konkretisierung des Problems

Die Abbildung 3.4 zeigt die zwei Möglichkeiten, wie die Antwort nach Ersetzung der WS-Addressing Headers im ESB fehlgeleitet werden kann. Der erste Web-Service (WS1) sendet seine Anfrage an den Proxy für WS2 im ESB und gibt sich dabei selbst als Antwortadresse an. Im ersten Fall (Abbildung 3.4, Punkt 1) geht der ESB von synchroner Kommunikation auf Transportebene aus und ersetzt sowohl Ziel als auch Antwortadresse, letztere durch eine anonyme Adresse. Der zweite Web-Service arbeitet nun seine Anfragen ab und möchte später eine Antwort senden. Da er die anonyme Adresse als Ziel verwendet, kommt die Antwort nirgends an.

Im zweiten Fall (Abbildung 3.4, Punkt 2) gibt der ESB seine eigene Adresse (also den Proxy für WS2) für die Antwort an, er erwartet daraufhin eine ganz bestimmte Antwort. Da WS2 aber später eine neue Anfrage als Antwort sendet, leitet er diese Nachricht wieder an WS2 um. Hierdurch kommt es zu einem Fehler im Proxy-Dienst für WS2, da er die angeforderte Operation nicht unterstützt - denn diese bietet WS1.

Bei asynchroner Kommunikation entspricht die Antwortadresse einem speziellen Endpunkt im Proxy für WS2, der nur für die Antwort offen gehalten wird. Dieser erwartet eine Nachricht, die eine bestimmte Struktur aufweist. In der WSDL-Beschreibung von WS2 sind hierfür das *Message Exchange Pattern* sowie die definierten Rückgabedatentypen vorgesehen. Da die Antwort aber nicht entsprechend aussieht, kommt es zu einem Fehler bei der Abarbeitung der Nachricht im ESB.

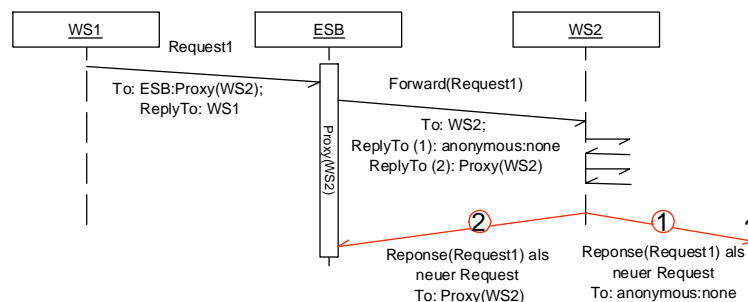


Abbildung 3.4: Fehlgeschlagene Anfrage über den ESB

Im aktuellen Entwicklungszweig vom Synapse Service Bus, der die Grundlage für den verwendeten WSO2 ESB bildet, wird an einer Option gearbeitet, um die WS-Addressing Header möglichst unangetastet zu lassen. Abbildung 3.5 zeigt diese Kommunikation. Die von WS2 empfangene Nachricht enthält hierbei weiterhin die Referenz von WS1 als Antwortadresse. Sendet WS2 später seine Nachricht, geht diese am ESB vorbei und kann folglich nicht protokolliert werden.

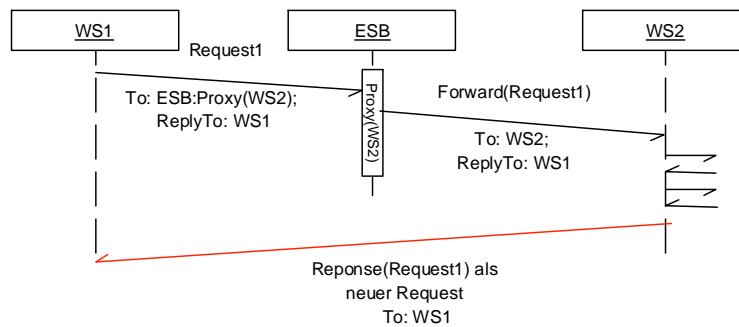


Abbildung 3.5: Antwort geht am ESB vorbei, wenn die Header erhalten bleiben

Im vierten Kapitel soll nun eine mögliche Lösung für das Problem vorgestellt werden, welche die Ortstransparenz wiederherstellt und die Protokollierung der Nachrichten gewährleistet. Dafür werden die folgenden Punkte benötigt:

- Sicherstellung der Ortstransparenz, dafür
 - ist eine Modifikation der Header notwendig, damit jeder Teilnehmer nur den ESB als Kommunikationspartner sieht
 - sowohl eingehender Nachrichten an das Ziel eines Proxy-Dienstes als auch die Antworten an den anfragenden Dienst
- Gewährleistung der Überwachung
 - Erstellung eines generischen Proxy-Dienstes, der die Antworten annimmt und protokolliert
 - Umleitung der Antworten aller eingehenden Nachrichten über diesen Antwort-Proxy

4 Konzept und Implementierung

Dieses Kapitel stellt eine Möglichkeit zur Behebung der in der Anforderungsanalyse aufgeführten Probleme vor. Der erste Teil des Kapitels gibt eine Übersicht über die Lösung, der nächste Teil erklärt die einzelnen Komponenten und skizziert ihre Funktionsweise. Abschließend zeigt das Beispiel die praktische Arbeitsweise der Lösung, wobei auch die erforderlichen Änderungen in der Konfiguration des ESB erläutert werden.

4.1 Lösungsansatz

Die Abbildung 4.1 zeigt die Anordnung der Mediatoren in den Sequenzen der jeweiligen Proxy-Dienste.

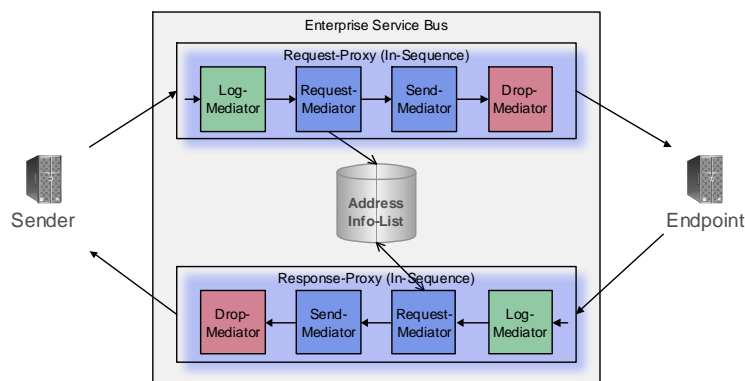


Abbildung 4.1: Übersicht über die Anordnung der Mediatoren

Die grundlegende Idee der Lösung ist die Speicherung der Adressinformationen in einer geeigneten Datenstruktur sowie der Umleitung über einen Proxy-Dienst. Weitere Möglichkeiten wären die Speicherung der Adressinformationen innerhalb der Header oder der Nutzlast der Nachricht. Im Header können diese allerdings nicht ohne Anpassung der Dienste bei den Teilnehmern gespeichert werden, da bei der Erstellung der neuen Anfrage diese Headerinformationen eingefügt werden müssen. Ähnlich verhält es sich mit der Speicherung in der Nutzlast, denn auch hier muss dann der Teilnehmer diese Informationen wieder in die Nachricht einfügen, wenn er die Antwort sendet. Die einzige Möglichkeit, die keinen Einfluss auf die involvierten Endpunkte hat, ist also die Speicherung der Adressinformationen innerhalb des ESB.

Der Antwort-Dienst ist als generischer Proxy angelegt, der keinen definierten Endpunkt aufweist. Er leitet die eingehenden Nachrichten entsprechend der Adressinformationen zum ursprünglich im ReplyTo-Feld angegebenen Endpunkt. Als weitere Alternative zu einem generischen Proxy ist es denkbar, für jede eingehende Anfrage einen neuen Antwort-Proxy zu erstellen, der nur für diese eine Nachricht zuständig ist und diese

dann an einen über die Synapse-Proxy-Konfiguration definierten Endpunkt leitet. Dieses Vorgehen würde zwar spezialisierte Mediatoren im Antwort-Proxy vermeiden, hat allerdings zwei schwerwiegende Nachteile. Einerseits müsste dazu relativ tief in die Struktur von Synapse eingegriffen werden, damit für jede Anfrage ein neuer Proxy dynamisch erstellt werden kann. Diese Funktionalität ist der Administrationsoberfläche vorbehalten und normalerweise nicht innerhalb eines Mediators ausführbar. Hierzu wäre also eine angepasste Version von Synapse erforderlich. Den zweiten Nachteil stellt die Performance dar. Bei einer hohen Anzahl paralleler Anfragen würde auch ein hohes Aufkommen an Proxy-Dienstes zur Abarbeitung der Antworten erforderlich sein. Dem könnte man nur entgegenwirken, wenn man zur Laufzeit eine Liste aller Proxy-Dienste führt und diese nicht für jede Antwort neu erstellt, sondern nur für die, die an bisher noch nicht geführte Endpunktreferenzen gehen. Dieser Aufwand ist aber höher als eine geführte Liste aller Adressinformationen in Verbindung mit einem generischen Antwortproxy. Die Unterscheidung der Nachrichten in der Liste der Adressinformationen geschieht über die in der Anfrage verwendete Nachrichtenennung, auf welche die Antwort Bezug nimmt.

Für diesen Ansatz sind also drei verschiedene Mediatoren notwendig. Der erste Mediator kümmert sich um die Speicherung der Adressinformationen sowie der Ersetzung der korrekten Zieladresse (RequestMediator). Der nächste Mediator greift in den Sendemechanismus des ESB ein und verhindert die Ersetzung der WS-Addressing Informationen durch die des ESB (WSASendMediator). Der dritte Mediator muss dann auf dem Rückweg den Endpunkt der Antwort wieder auf die im ReplyTo-Feld der Anfrage stehende Adresse ändern (ResponseMediator).

4.2 Mediatoren

Der folgende Abschnitt beschreibt die in Abbildung 4.1 gezeigten Mediatoren im Einzelnen. Der bereits vorhandene Mediator für Logging sowie der von Synapse gelieferte DropMediator werden dabei nur kurz erläutert.

4.2.1 LogMediator

Der LogMediator stammt aus dem Choreographie-Monitor des EoS-Projektes. Dieser Mediator sendet eine Kopie aller Anfragen an einen als Parameter spezifizierten Endpunkt, der diese protokolliert, also zur späteren Verifikation speichert. Hierfür bereitet er eine neue Nachricht vor, die alle Header-Elemente sowie den eigentlichen Envelope der eingehenden Nachricht im Payload (Body) enthält. Weitere Informationen zum Monitor-Prinzip liefert [RR09].

4.2.2 RequestMediator

Der RequestMediator bildet zusammen mit dem später vorgestellten ResponseMediator den wichtigsten Teil zur Lösung der aufgezeigten Probleme. Er hat die Aufgabe, die Information aus dem ReplyTo-Feld der WS-Addressing Header zu sichern und für die spätere Ersetzung im ResponseMediator bereitzustellen. Diese Adressinformationen werden aus dem Nachrichtenkontext ausgelesen und in der Klasse *AddressingInfoSet* gespeichert. Der RequestMediator befindet sich im Paket *de.matrixteam.bsc.ws.mediators*.

In der Konfiguration des Mediators sind zwei Parameter notwendig: Zieladresse (*targetEPR*) sowie Adresse des Antwortdienstes (*responseEPR*). Die Zieladresse ist notwendig, da dem WSASendMediator die Endpunktreferenz der Konfiguration nicht zugänglich ist, diese wird dann obsolet. Die Adresse des Antwortdienstes trägt er dann als ReplyTo-Information in die abgehende Nachricht ein.

Der RequestMediator ist als Mediator mit eigenem Namespace in Synapse eingetragen, damit er in der Konfiguration parametrisiert werden kann. Hierfür sind die zwei Helfersklassen *RequestMediatorFactory* und *RequestMediatorSerializer* erforderlich, die im gleichen Paket liegen. Die Factory hat die Aufgabe, die Konfiguration des Mediators aus der XML-Konfigurationsdatei (*synapse.xml*) auszulesen und den Mediator zu initialisieren. Im Gegensatz dazu ist der Serializer für das Schreiben der Konfiguration verantwortlich, wenn beispielsweise über das Web-Frontend vom Enterprise Service Bus Änderungen am Mediator vorgenommen werden.

Listing 4.1: Ausschnitt aus der *mediate(...)*-Funktion des RequestMediators

```
1 public boolean mediate( MessageContext arg0 )
2 {
3     boolean returnValue = true;
4     EndpointReference localePR = arg0.getTo();
5     // Adressinformationen erstellen...
6     AddressingInfo ai = new AddressingInfo(
7         arg0.getMessageID(),
8         arg0.getReplyTo().getAddress() );
9     // ...und in der Liste speichern
10    AddressingInfoSet.getInstance().add( ai );
11    // Adressinformationen im Nachrichtenkontext setzen
12    arg0.setReplyTo( new EndpointReference( _responseEPR ) );
13    arg0.setFrom( localePR );
14    arg0.setTo( new EndpointReference( _targetEPR ) );
15    // Adressinformationen im Header tauschen
16    SOAPEnvelope env = arg0.getEnvelope();
17    if ( env != null ) {
18        SOAPHeader header = env.getHeader();
19        if ( header != null ) {
20            returnValue &=
21                AddressingWorker.modifyWSAHeaderInformation(
```

```

22         header, "To", _targetEPR ) &&
23     AddressingWorker.modifyWSAHeaderInformation(
24         header, "ReplyTo", _responseEPR ) &&
25     AddressingWorker.modifyWSAHeaderInformation(
26         header, "From", localEPR.getAddress( ) );
27     }
28 }
29 return returnValue;
30 }

```

Das Listing 4.1 zeigt eine gekürzte Fassung des Requestmediators. Zu Beginn speichert er die benötigten WS-Addressing Informationen in einer später vorgestellten Datenstruktur (Zeile 6-8). Hierzu gehören Nachrichtenennung sowie die ReplyTo-Adresse für die Antwort. Diese Datenstruktur fügt er in eine Liste ein (Zeile 10), auf die der Antwortproxy später zugreifen kann.

In Zeile 12-14 werden die Adressinformationen im Nachrichtenkontext gesetzt. Die Zieladresse ist für den WSASendMediator von besonderer Bedeutung, da er wie angesprochen nicht auf die eigentliche Endpunkt-Referenz zugreifen kann. Die Absender- sowie Antwortadresse dienen im Nachrichtenkontext lediglich für Tracing- und Debuginformationen anderer Module, da ein Zugriff über den eigentlichen SOAP-Envelope umständlich ist. Der Mediator tauscht die Absenderadresse durch seine eigene Adresse aus, die er in Zeile 4 aus der Zieladresse der eingehenden Nachricht zieht.

Die Informationen, die der Ziel-Webdienst später benötigt, müssen im SOAP-Envelope selbst stehen. In Zeile 16-19 wird sichergestellt, dass Envelope und Header entsprechend gesetzt sind, in Zeile 21 - 26 werden sie dann entsprechend modifiziert. Dies geschieht über die Helferklasse *AddressingWorker*, die auszugsweise in Listing 4.2 gezeigt ist. Die Ersetzung ist einerseits notwendig, damit der Client weiß, wohin die Nachricht zu senden ist, andererseits aber auch um die in Kapitel 3 angesprochene Ortstransparenz zu gewährleisten. In der Nachricht soll nach Abarbeitung keine Referenz mehr auf den ursprünglichen Absender zeigen.

Listing 4.2: Ersetzung der WS-Addressing Informationen

```

1 static final String __wsaNS = "http://www.w3.org/2005/08/addressing";
2 public static boolean modifyWSAHeaderInformation(
3     SOAPHeader header, String localName, String newValue )
4     {
5         QName searchFor = new QName( __wsaNS, localName );
6         Iterator it = header.getChildrenWithName( searchFor );
7
8         if ( it.hasNext() ) {
9             OMElement el = ( OMElement ) it.next();
10            if ( localName.equals( "To" ) ) {
11                el.setText( newValue );
12            }

```

```

13     }
14 }

```

Über den Namespace von WS-Addressing und den lokalen Namen des zu ersetzenden Objektes (beispielsweise "To") sucht die Funktion das entsprechende Element im Header über die Methode `getChildrenWithName(QName)` (siehe Listing 4.2, Zeile 6). Diese liefert als Ergebnis einen Iterator; das erste gefundene Objekt entspricht dem gesuchten. Je nach dem Element, das modifiziert werden soll, steht der Wert direkt im Element (bei "To") oder ist ein Kind des Elementes (z.B. "ReplyTo", hier nicht gezeigt). Über die Methode `setText(...)` kann dann ein neuer Wert gesetzt werden.

Sind die WS-Addressing Informationen ersetzt, ist die Arbeit des Mediators abgeschlossen. Wird ein Element nicht gefunden, bricht die Verarbeitung der eingehenden Nachricht im ESB ab, da der Mediator dann den Wert `false` zurückgibt.

Die angesprochene Konfiguration des Mediators zeigt das Listing 4.3. Die beiden Parameter sind Unterknoten des Mediators, die Factory liest diese später aus und übergibt sie dem Mediator nach der Initialisierung durch Aufruf der entsprechend benannten Setter (`setTargetEPR(String)`, `setResponseEPR(String)`).

Listing 4.3: Mediatorkonfiguration für den RequestProxy

```

1 <rp:requestProxy xmlns:rp="http://ws.apache.org/ns/synapse">
2   <rp:responseEPR>http://server/ResponseProxy</rp:responseEPR>
3   <rp:targetEPR>http://server/Target</rp:targetEPR>
4 </rp:requestProxy>

```

4.2.3 AddressingInfo

Die beiden Klassen `AddressingInfo` und `AddressingInfoList` stellen die Datenstrukturen zur Speicherung der Adressinformationen bereit und sind im Paket `de.matrixteam.bsc.ws.addressing.data` zu finden. Ein Objekt vom Typ `AddressingInfo` enthält immer eine Endpunktreferenz für die Zieladresse sowie die damit verbundene Nachrichten-ID. Diese beiden Eigenschaften werden im Konstruktor gesetzt und sind über entsprechende Getter-Methoden abrufbar. Die Klasse `AddressInfoSet` stellt eine Liste mit den Adressinformationen bereit. Sie ist nach dem Singleton-Pattern (siehe [GHJV04], S. 127) implementiert, gibt also immer nur Zugriff auf eine Instanz der Klasse frei. Der Konstruktor ist privat und kann nur von der Klassenmethode `getInstance()` aufgerufen werden, der dies genau einmal tut. Hierdurch wird sichergestellt, dass es nur eine Liste mit Adressinformationen gibt. Die Instanz der Klasse stellt Methoden bereit, um Objekte des Typs `AddressInfo` zu speichern, abzurufen und um sie aus der Liste zu entfernen. Der Zugriff auf die speichernde Datenstruktur geschieht hierbei synchronisiert, um Probleme beim gleichzeitigen Zugriff zu verhindern.

4.2.4 ResponseMediator

Der ResponseMediator stellt das Gegenstück zum RequestMediator dar und soll die Adressinformationen wiederherstellen. Er findet sich auch im Paket *de.matrixteam.bsc.ws.mediators*. Im Gegensatz zum RequestMediator ist er als einfacher Klassen-Mediator in Synapse konfiguriert, da er keine weiteren Parameter benötigt. Da er aber auf die gleiche Liste von Adressinformationen zugreifen soll, ist es notwendig, den ResponseMediator auf dem gleichen ESB wie den RequestMediator einzurichten.

Um die Nachrichten an die richtige Adresse umzuleiten, muss der Mediator wissen, auf welche ursprüngliche Anfrage die aktuell zu verarbeitende Nachricht sich bezieht. Hierfür verwendet er die *RelatesTo*-Information im WS-Addressing Header, die dafür entsprechend vom Client gesetzt werden muss. Danach ersetzt der Mediator wie der RequestMediator die Adressinformationen der Nachricht, damit die Ortstransparenz gewährleistet wird.

Listing 4.4: Ausschnitt aus der *mediate(...)*-Funktion des ResponseMediators

```
1 public boolean mediate( MessageContext arg0 )
2 {
3     boolean returnValue = true;
4     boolean hasCtxReplyTo = false;
5     EndpointReference localePR = arg0.getTo();
6     // prüfen, ob Nachricht Antwortadresse enthaelt
7     if ( ( arg0.getReplyTo() != null ) &&
8         ( !( ( arg0.getReplyTo().hasAnonymousAddress() ) ||
9           ( arg0.getReplyTo().hasNoneAddress() ) ) ) )
10    {
11        hasCtxReplyTo = true;
12    }
13    // Adressinformationen abrufen und in Liste loeschen
14    String relatesTo = arg0.getRelatesTo().getValue();
15    AddressingInfo ai = AddressingInfoSet.getInstance().get( relatesTo );
16    String endpoint = ai.getEndpoint();
17    AddressingInfoSet.getInstance().remove( ai );
18    // wenn Antwortadresse, dann muss diese gespeichert werden
19    if ( hasCtxReplyTo ) {
20        AddressingInfo replyTo = new AddressingInfo(
21            arg0.getMessageID(),
22            arg0.getReplyTo().getAddress() );
23
24        arg0.setReplyTo( localePR );
25        AddressingInfoSet.getInstance().add( replyTo );
26    }
27    // setze Zielpunkt fuer Sendevorgang
28    arg0.setTo( new EndpointReference( endpoint ) );
29    arg0.setFrom( new EndpointReference( localePR.getAddress() ) );
```

```
30 // ersetze WS-Addressing Informationen im Header
31 SOAPEnvelope env = arg0.getEnvelope();
32 if ( env != null ) {
33     SOAPHeader header = env.getHeader();
34     if ( header != null ) {
35         returnValue &=
36             AddressingWorker.modifyWSAHeaderInformation(
37                 header, "To", endpoint ) &&
38             AddressingWorker.modifyWSAHeaderInformation(
39                 header, "From", localEPR.getAddress() );
40
41         if ( hasCtxReplyTo ) {
42             returnValue &= AddressingWorker.modifyWSAHeaderInformation(
43                 header, "ReplyTo", localEPR.getAddress() );
44         }
45     }
46 }
47 return returnValue;
48 }
```

Das Listing 4.4 zeigt einen Ausschnitt aus der Implementierung des ResponseMediators. In Zeile 4 wird die Variable *hasCtxReplyTo* definiert, die dann den Wert `true` annimmt, wenn die Antwort auf die ursprüngliche Anfrage selbst auch wieder eine Antwortadresse enthält. In diesem Fall muss der Mediator wie der RequestMediator für die Umleitung auf sich selbst sorgen und die Adressinformationen in der Klasse *AddressInfoSet* speichern. Eine solche Antwort liegt dann vor, wenn der Knoten `ReplyTo` im WS-Addressing Header existiert (also nicht den Wert `null` hat), keine anonyme Adresse aufweist und nicht leer ist. Diese Bedingungen werden in Zeile 7-9 geprüft.

In der Zeile 14 wird die Kennung der Nachricht, auf die sich die aktuelle Anfrage bezieht, über die Funktion *getRelatesTo()* extrahiert. Liefert diese Funktion kein Ergebnis oder ist die Nachrichtenennung in der Liste nicht enthalten, so muss die Verarbeitung an dieser Stelle abgebrochen werden (im Listing nicht gezeigt). Andernfalls speichert der Mediator die Zieladresse in einem String zwischen und löscht die aktuell bearbeitete Adressinformation aus der Liste (Zeile 17).

Ist eine Antwortadresse im Umschlag gesetzt (*hasCtxReplyTo*), so wird in den Zeilen 20-25 wie im RequestMediator ein *AddressInfo*-Objekt erzeugt und in der Liste abgelegt. Die eigene Adresse für die Antwort nimmt der Mediator dabei in Zeile 5 aus dem Ziel der eingehenden Nachricht, welches der ResponseMediator ist.

Analog zum RequestMediator werden zum Wahren der Ortstransparenz sowie zur korrekten Weiterleitung der Nachricht in den Zeilen 28 - 39 die entsprechenden WS-Addressing Header modifiziert und die Adressierungseigenschaften im *MessageContext* gesetzt. Für den Fall einer vorhandenen Antwortadresse muss zudem die Rücksendeadresse geändert werden, dies erfüllt der Aufruf in Zeile 42.

Nach erfolgreicher Abarbeitung kann die Nachricht durch den nachfolgenden WSA-SendMediator versendet werden, andernfalls wird die Abarbeitung durch einen Rückgabewert `false` abgebrochen. Sind in der Nachricht keine Headerinformationen auffindbar, bricht der Code mit einer Ausnahme vom Typ *SynapseException* ab und führt zu einer Fehlernachricht an den Absender (hier nicht gezeigt).

4.2.5 WSASendMediator

Der implementierte WSASendMediator ersetzt den vom ESB am Ende der Sequenz vorgesehenen, eingebauten SendMediator. Der Hauptunterschied der beiden Mediatoren liegt in der Behandlung der WS-Addressing Header, welche vom eingebauten ersetzt werden. Abbildung 4.2 zeigt die Aufrufkette des originalen Send-Mediatoren in Synapse. Die unterste ist die *mediate(...)*-Methode dieses Send-Mediatoren, nach oben gehen die Aufrufe immer tiefer ins System.

Abbildung 4.2: Aufruf-Hierarchie des Send-Mediatoren in Synapse

```

send(EndpointDefinition, MessageContext) org.apache.synapse.core.axis2.Axis2FlexibleMEPClient
└─ sendOn(EndpointDefinition, MessageContext) org.apache.synapse.core.axis2.Axis2Sender
    └─ send(EndpointDefinition, MessageContext) org.apache.synapse.core.axis2.Axis2SynapseEnvironment
        └─ send(MessageContext) org.apache.synapse.endpoints.AddressEndpoint
            └─ mediate(MessageContext) org.apache.synapse.mediators.builtin.SendMediator
  
```

Die oberste Methode *send(...)* kommt dabei aus der Klasse *Axis2FlexibleMEPClient*, welche die Nachricht schließlich an einen Axis2-Client zur Zustellung weitergibt. In dieser Methode findet auch das Herauslösen der WS-Addressing Informationen statt. Diese ersetzt der Axis2-Client dann je nach Konfiguration beim Senden durch die Daten des ESB oder lässt sie ganz weg (siehe 3.4).

Listing 4.5: Auszug aus den Methodensignaturen des *Axis2FlexibleMEPClient*s

```

public static void send(EndpointDefinition endpoint,
    org.apache.synapse.MessageContext synapseOutMessageContext);
private static MessageContext cloneForSend(MessageContext ori);
public static SOAPEnvelope removeAddressingHeaders(
    MessageContext axisMsgCtx);
private static void detachAddressingInformation(
    ArrayList headerInformation);
  
```

Am Sendevorgang der Nachricht sowie bei der Entfernung der entsprechenden Header sind die Methoden der Klasse *Axis2FlexibleMEPClient* beteiligt, die das Listing 4.5 zeigt. Die statische Methode *send(...)* ist dabei der Einstiegspunkt, der letztendlich von der *mediate(...)*-Funktion des SendMediators aufgerufen wird. Nach der Feststellung diverser Optionen, welche die zu sendende Nachricht erfordert (Sicherheit, Reliable Messaging, ...), kloniert die Methode *cloneForSend(...)* den MessageContext. Dies ist erforderlich, damit

bei manipulativen Operationen durch diese sowie aufgerufenen Methoden keine Veränderung der Nachricht stattfindet, um diese beispielsweise im nächsten Mediator noch an weitere Endpunkte mit anderen Optionen zu senden. Die Funktion, welche die Nachricht kloniert, entfernt auch die WS-Addressing Header. Dies geschieht, damit die Kommunikation auch mit Endpunkten funktioniert, die WS-Addressing nicht unterstützen. Die hier aufgerufene Methode `removeAddressingHeaders(...)` sammelt alle WS-Addressing Elemente und übergibt sie der vierten Methode `detachAddressingInformation(...)`, welche an diesen Header-Elementen dann jeweils die Methode `detach(...)` zum Ablösen aufruft.

Da im EoS-Projekt von WS-Addressing grundsätzlich Gebrauch gemacht wird und alle angesprochenen Clients dies auch unterstützen, ist eine solche Trennung nicht nur unnötig, sondern in Anbetracht der oben angesprochenen Probleme sogar hinderlich. Die Methoden der Klasse sind statisch (Klassenmethoden) und lassen sich zwar überschreiben, jedoch kann in einer erbbenden Klasse nicht einfach nur die Methode zum Klonen überschrieben werden. Ruft eine Klasse in Java eine statische Methode einer anderen Klasse auf, die diese von ihrer Basisklasse erbt, so geht der Aufruf an die Basisklasse selbst. Diese dort aufgerufenen Methode kann nicht auf die statischen Methoden der erbbenden Klasse zugreifen.

Listing 4.6: Aufruf von statischen Methoden in Basisklassen

```

1 public class A {
2     public static void writeA() { System.out.println(getA()); }
3     public static String getA() { return "A"; } }
4 public class B extends A {
5     public static String getA() { return "anderesA"; } }
6
7 public class Main {
8     public static void main(...) { B.writeA(); } }
9
10 // Ausgabe: A

```

Das Code-Beispiel 4.6 verdeutlicht diesen Umstand. Die Klasse `B` versucht, eine statische Methode der Klasse `A` zu überschreiben. Beim Aufruf in der Klasse `Main` wird jedoch - da `B` nicht selbst die Methode anbietet - `writeA()` der Basisklasse aufgerufen und damit auch die eigenen Implementierung von `getA()` verwendet. Wollte Klasse `B` in der Methode `getA()` auf die Implementierung der Basisklasse zugreifen wollen, ginge dies auch nur mit dem direkten Aufruf (`A.getA()`), nicht aber `super.getA()`).

Dies führt dazu, dass in der Lösung die komplette Klasse `Axis2FlexibleMEPClient` kopiert werden musste, wobei in der Methode zum Klonen der Nachricht der Aufruf an die Methode, welche die Header entfernt, gelöscht wurde. Die neue Klasse heißt `WSAFlexibleMEPClient` und findet sich im Paket `de.matrixteam.bsc.ws.send.core`

Die Implementierung der `WSASendMediator`-Klasse ist entsprechend kurz, da diese nur noch die Nachricht versendet. Es wird eine Endpunktreferenz definiert, für welche die Nutzung von WS-Addressing aktiviert ist. Die `mediate(...)`-Funktion setzt das Ziel und

rufft danach die *send(...)*-Methode des *WSAFlexibleMEPClient*s auf. Gegenüber der Originalimplementierung hat der *WSASendMediator* keine Informationen über den Endpunkt der Nachricht, sondern nutzt die Zieladresse aus dem *MessageContext* der Nachricht. Der *SendMediator* von Synapse arbeitet mit einer in der Konfiguration definierten Endpunktreferenz.

Für die Konfiguration kann der *WSASendMediator* analog zum *ResponseMediator* als Klassenmediator definiert werden.

4.2.6 DropMediator

Der letzte Mediator in der Sequenz für eingehende Nachrichten ist der *DropMediator*. Dieser gehört zu den eingebauten Basismediators von Synapse und befindet sich im Paket *org.apache.synapse.mediators.builtin*. Er bricht die weitere Verarbeitung ab, indem er die Zieladresse der Nachricht auf *null* setzt und den Wert *false* zurückgibt. Der restliche Körper der Methode *mediate(...)* besteht nur noch aus Nachrichten für den Debug- oder Trace-Modus.

Der Abbruch der Sequenz an dieser Stelle ist erforderlich, damit der ESB die Nachricht nicht ein zweites Mal sendet. Der zweite Sende-Vorgang über eine definierte Endpoint-Reference des Proxy-Dienstes würde zu dem in Abbildung 3.4 gezeigten Fehler führen, außerdem könnte das Zielsystem durch die zweimalige Zustellung der Nachricht irritiert sein.

4.3 Überprüfung am Beispiel

Die bisher vorgestellten Mediatoren werden nun zur Überprüfung der Anforderungen aus dem dritten Kapitel am Beispiel gezeigt. Das in Kapitel 3 vorgestellte Beispiel enthält eine Abfrage, die der Client asynchron ausführt und deren Beantwortung von der korrekten Ersetzung der Adressinformationen abhängt. Das Listing 3.2 auf Seite 25 zeigt dabei die falsche Antwort, die der ESB an den Empfänger weiterleitet.

Listing 4.7: In-Sequence des ELOProxy-Dienstes im ESB

```
1 <syn:inSequence>
2   <wslog1:wslog ... />
3   <rp:requestProxy xmlns:rp="http://ws.apache.org/ns/synapse">
4     <rp:responseEPR>
5       http://localhost:8280/soap/ResponseProxy
6     </rp:responseEPR>
7     <rp:targetEPR>
8       http://localhost:8080/axis2/services/ELOBehaviorService
9     </rp:targetEPR>
10  </rp:requestProxy>
11  <syn:class name="de.matrixteam.bsc.ws.mediators.WSASendMediator"/>
12  <syn:drop/>
```

```
13 </syn:inSequence>
```

Zur Einrichtung der neuen Mediatoren ist eine Modifikation der Konfigurationsdatei des Enterprise Service Busses erforderlich. Im Detail ist eine Anpassung der Sequenz für eingehende Anfragen des ELOProxy-Dienstes sowie die Einrichtung des ResponseProxy-Dienstes notwendig. Das Listing 4.7 zeigt die neue In-Sequence für den ELOProxy, der Logging-Mediator wurde dabei abgekürzt. Der ESB läuft hier beispielsweise auf dem gleichen Rechner wie der Dienstanbieter im Axis2-Server. Axis2 ist über Port 8080 erreichbar, der ESB über Port 8280. Zeile 3 - 10 von Listing 4.7 zeigt die Definition des RequestProxy, wie bereits in Kapitel 4.2.2 besprochen mit den beiden Parametern für den Antwortproxy sowie der Endpunktreferenz des eigentlichen Dienstes. In Zeile 11 wird der Klassenmediator WSASendMediator eingebunden, der für das Verschicken der Nachricht verantwortlich ist, durch den DropMediator in Zeile 12 wird die weitere Verarbeitung abgebrochen, damit es nicht zum mehrfachen Versenden kommt. Die Out-Sequence sowie die Endpoint-Reference Knoten in der Konfiguration für den ELOProxy müssen nicht angepasst sind nicht anzupassen; sie könnten auch gelöscht werden.

Listing 4.8: In-Sequence des ResponseProxy-Dienstes im ESB

```
1 <syn:inSequence>
2   <wslog2:wslog ... />
3   <syn:class name="de.matrixteam.bsc.ws.mediators.ResponseMediator"/>
4   <syn:class name="de.matrixteam.bsc.ws.mediators.WSASendMediator"/>
5   <syn:drop/>
6 </syn:inSequence>
```

Das Listing 4.8 zeigt die Sequenz für eingehende Nachrichten des ResponseProxy, der in der Konfiguration einzurichten ist. Hat man den leeren Mediator in der Administrationsoberfläche angelegt, kann man entweder die In-Sequence von Hand in die Konfiguration eintragen oder man fügt die einzelnen Elemente über die GUI in die Sequenz ein. Der ResponseProxy ist analog zum WSASendMediator als unparametrisierter Klassenmediator einzufügen, für Drop- und WSASendMediator gelten die gleichen Eigenschaften wie im ELOProxy.

Listing 4.9: WS-Addressing Header nach der Ersetzung durch den RequestMediator

```
1 <s:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
2   <wsa:To>
3     http://localhost:8080/axis2/services/ELOBehaviorService
4   </wsa:To>
5   <wsa:ReplyTo>
6     <wsa:Address>http://localhost:8280/soap/ResponseProxy</wsa:Address>
7   </wsa:ReplyTo>
8   <wsa:From>
9     <wsa:Address>http://localhost:8280/soap/ELOProxy</wsa:Address>
10  </wsa:From>
```

```

11 <wsa:MessageID>
12     urn:uuid:16266B2A57A4721BB71242149820531
13 </wsa:MessageID>
14 <wsa:Action>
15     http://vsis.de/choreography/monitoring/example/requestInformation
16 </wsa:Action>
17 </s:Header>

```

Führt man das Beispiel nun erneut aus, so kann man am ELOBehaviorService die WS-Addressing Header wie in Listing 4.9 abfangen. Wie in Zeile 6 und 9 ersichtlich wird, hat der Mediator die Absenderadresse durch die des ELOProxy-Dienstes sowie die Antwortadresse durch die des Antwortproxy-Dienstes getauscht. Der ELOBehaviorService sieht keine Adressen mehr ausser die des ESB, die Ortstransparenz ist also gewährleistet. Durch die Umleitung der Nachricht über den Antwortproxy kann der ESB die Nachricht zudem protokollieren, also ist auch die zweite Anforderung erfüllt (Gewährleistung der Protokollierbarkeit).

Listing 4.10: Header der Antwort nach Ersetzung durch den ResponseMediator

```

1 <s:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
2   <wsa:To>
3     http://localhost:8080/axis2/services/EJBehaviorService
4   </wsa:To>
5   <wsa:ReplyTo>
6     <wsa:Address>
7       http://www.w3.org/2005/08/addressing/none
8     </wsa:Address>
9   </wsa:ReplyTo>
10  <wsa:From>
11    <wsa:Address>
12      http://avatar.local:8280/soap/ResponseProxy
13    </wsa:Address>
14  </wsa:From>
15  <wsa:MessageID>
16    urn:uuid:89C9E78D5CDF6CDBA41242149821378
17  </wsa:MessageID>
18  <wsa:Action>
19    http://vsis.de/choreography/monitoring/example/submitInformation
20  </wsa:Action>
21  <wsa:RelatesTo>
22    urn:uuid:16266B2A57A4721BB71242149820531
23  </wsa:RelatesTo>
24 </s:Header>

```

Das Listing 4.10 zeigt abschließend die Antwort des ELOs an den EJNM, die nun korrekterweise im ESB protokolliert wird. Die Antwort wie dargestellt ist die, welche der EJBehaviorService sieht. Die Antwortadresse hat der ResponseMediator im Antwortproxy

ersetzt, sodass der Empfänger auch wieder nur den ESB sieht. Also ist auch auf dieser Seite die Ortstransparenz des Enterprise Service Busses gegeben.

Abschließend zeigt das Beispiel, dass die in Kapitel 3 gestellten Anforderungen an die in dieser Arbeit erbrachten Lösung im vollen Umfang erfüllt sind. Durch die Ersetzung der Adressen bleibt die Ortstransparenz des ESB auch bei asynchronen Kommunikationsmustern erhalten. Weiterhin können asynchron versendete Nachrichten durch die Umleitung über den Antwortproxy protokolliert werden.

Eine genaue Anleitung zur Einrichtung des Beispiels sowie zur Verwendung der erarbeiteten Lösung findet sich auf dem der Arbeit beigelegten Datenträger in der Datei *Readme.txt*. Weiterhin gibt es dort vorbereitete Versionen der verwendeten Softwareprodukte sowie den kompletten Quellcode.

5 Zusammenfassung und Ausblick

Das letzte Kapitel fasst die in der Arbeit gewonnen Erkenntnisse zusammen und soll einen Ausblick auf zukünftige Entwicklungen sowie Erweiterungsmöglichkeiten der erarbeiteten Lösung geben.

5.1 Zusammenfassung

Service-orientierte Architekturen stellen einen Ansatz dar, der bei korrekter Umsetzung Agilität und Flexibilität in Organisationen verspricht. Aktuelle Trends, allen voran die Buzzwords Web 2.0 und Cloud Computing, setzen auf Dienstkonsumenten und Dienstanbieter, also Services und den damit verbundenen SOAs. Bei Prozessen über Unternehmensgrenzen hinweg stoßen geregelte Workflows durch Orchestrierung an ihre Grenzen, da sie eine zentrale Instanz zur Vermittlung erfordern. Choreographien bieten sich zur hingegen bieten keinerlei Möglichkeit zur Überprüfung der Kommunikation.

Die Überwachung von Nachrichten zur Einhaltung eines vorher definierten Protokolls ist jedoch ein wichtiger Aspekt beispielsweise bei der Vernetzung verschiedener Behörden. Das im Projekt Enforcements of Steps (EoS) des Fachbereichs VSIS der Universität Hamburg erarbeitete Frameworks zum Monitoring bietet genau diese Funktionalität. Wie aber im dritten Kapitel der Arbeit gezeigt, gab es bisher Probleme bei der Überwachung asynchroner Kommunikationsmuster auf Anwendungsebene, die im Rahmen dieser Arbeit erörtert wurden. Weiterhin gab es bei der Nutzung von asynchronen Nachrichten eine Verletzung der Eigenschaften, die ein ESB in der Praxis bieten sollte.

In der Anforderungsanalyse haben sich die folgenden Punkte als die

- Sicherstellung der Ortstransparenz
- Gewährleistung der Protokollierbarkeit

Das Kapitel *Konzept und Implementierung* zeigte hier eine Möglichkeit auf, wie diese Probleme lösbar sind. Ohne Eingriffe in den verwendeten Enterprise Service Bus von WSO2 konnte eine Möglichkeit zusammengestellt werden, einerseits die verletzte Ortstransparenz wiederherzustellen als auch die Gesamtheit aller Nachrichten zu protokollieren. Dies erreichte die vorgestellte Lösung durch eine Umleitung der Nachrichten über einen Antwortproxy, der diese an eine Protokollinstanz weiterleiten kann. Durch die Ersetzung der Absender- sowie Antwortadressen sehen die Clients keine anderen Teilnehmer als den ESB, also gilt die Ortstransparenz auch für asynchrone Nachrichten. Da der Antwortproxy generisch gestaltet wurde, also für *alle* Antworten zuständig ist, konnte möglichen Performance-Problemen vorgebeugt werden.

Ein aktuelles Beispiel aus einer Fallstudie des R4eGov-Projekt der europäischen Union konnte die Funktionalität der erbrachten Lösung zeigen. Diese benötigte lediglich eine Modifikation der Konfiguration des Enterprise Service Busses sowie die Installation der notwendigen Mediatoren.

5.2 Ausblick

Die im vierten Kapitel vorgeführte Lösung stellt keine Möglichkeit bereit, die Antworten über einen anderen Enterprise Service Bus zu leiten. Dies liegt an der fehlenden Unterstützung einer verteilten Adressinformations-Liste, die bisher nur innerhalb einer einzigen Java Virtual Machine existiert. Denkbar wäre eine Erweiterung durch die Übergabe der Adressinformationen an einen anderen ESB über Web Services, also statt der Speicherung in der lokalen Liste könnte die Information auch in einer Nachricht an einen zweiten ESB gesendet werden. Dieser kann die enthaltenen Adressen aus den Nachrichten nutzen und als Antwortproxy agieren.

Andererseits hat diese Arbeit nur die Ersetzung von Adressen aus den WS-Addressing Headern betrachtet. Jedoch nutzen SOAP-Erweiterungen wie WS-Coordination auch die Nutzlast der Nachricht zum Austausch von Web Service Adressen. WS-Coordination legt den Schwerpunkt auf Aktivitäten und bietet den Empfängern von Nachrichten auch sinnverwandte Web Services zu einer Aufgabe an (siehe [WCL⁺05], S. 228). Ein interessanter Punkt für den Ausbau der Implementierung stellt also das Parsen der Nutzlast nach weiteren Endpunktreferenzen dar, die dann auch entsprechend ersetzt und über den Enterprise Service Bus geleitet werden könnten.

Ein weiterer Punkt stellt aktuelle Entwicklungen im Synapse Bus dar. Der Wunsch, die WS-Addressing Header ungeändert weiter zu verschicken, besteht nicht nur in dem Rahmen des EoS Projektes. Aus diesem Grund arbeitet die Entwicklungsgemeinde momentan an einer Lösung, in der die Entwickler von Mediatoren durch eine Option im Nachrichtenkontext entscheiden können, ob die Header beibehalten werden sollen. Der Entwicklungszweig zum Zeitpunkt dieser Arbeit enthielt nur den Ansatz einer solchen Lösung. Wenn diese Option in der nächsten Version des Synapse Busses vollständig implementiert ist und auch in den WSO2 ESB übergegangen ist, wird allerdings nur der WSASendMediator dieser Arbeit überflüssig. Die Umleitung von Nachrichten, die auf der Anwendungsebene asynchron sind, ist weiterhin nicht vorgesehen.

Literaturverzeichnis

- [Bar03] BARRY, Douglas K.: *Web Services and Service-Oriented Architecture*. Morgan Kaufmann, 2003. – ISBN 1558609067
- [Cer02] CERAMI, Ethan: *Web Services Essentials*. O'Reilly, 2002. – ISBN 0596002246
- [CHRR04] CLEMENT, Luc ; HATELY, Andrew ; RIEGEN, Claus von ; ROGERS, Tony: UDDI Version 3. In: *UDDI Spec Technical Committee Draft, OASIS Open*, 2004. – http://uddi.org/pubs/uddi_v3.htm
- [CMRW07] CHINNICI, Roberto ; MOREAU, Jean-Jacques ; RYMAN, Arthur ; WEERAWARANA, Sanjiva: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. In: *W3C Recommendation*, 2007. – <http://www.w3.org/TR/2007/REC-wsdl20-20070626/>
- [DKS07] DAVIES, Jeff ; KRISHNA, Ashish ; SCHOROW, David: *The Definitive Guide to SOA: BEA Aqualogic Service Bus*. Apress, 2007. – ISBN 1590597974
- [EECB07] EUROPOL ; EUROJUST ; CANGH, Thomas V. ; BOUJRAF, Abdelkrim: The Eurojust-Europol Case Study (WP3-CS2). In: *R4eGov Case Studies*, 2007. – http://www.r4egov.eu/resources/details.php?Id_taxonomy=6
- [FFO05] FISCHER, Herbert ; FLEISCHMANN, Albert ; OBERMEIER, Stefan: *Geschäftsprozesse realisieren: Ein praxisorientierter Leitfaden von der Strategie bis zur Implementierung*. Vieweg+Teubner Verlag, 2005. – ISBN 3834800538
- [GHJV04] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-oriented Software*. 2nd. Addison-Wesley, 2004. – ISBN 9780582844421
- [GHM⁺07] GUDGIN, M. ; HADLEY, M. ; MENDELSON, N. ; MOREAU, J.-J. ; NIELSEN, H. F. ; KARMARKAR, A. ; LAFON, Y.: SOAP Version 1.2 Part 1: Messaging Framework W3C Recommendation. W3C - World Wide Web Consortium, 2007. – <http://www.w3.org/TR/soap12-part1/>
- [KBR⁺04] KAVANTZAS, Nickolas ; BURDETT, David ; RITZINGER, Gregory ; FLETCHER, Tony ; LAFON, Yves: Web Services Choreography Description Language Version 1.0. In: *W3C Working Draft*, 2004. – <http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/>
- [Mel08] MELZER, Ingo: *Service-orientierte Architekturen mit Web Services*. Spektrum Akademischer Verlag, 2008. – ISBN 9783827419934
-

- [Men07] MENGE, Falko: Enterprise Service Bus. In: *Free And Open Source Software Conference, 2007*. – <http://programm.froscon.de/2007/events/66.en.html>
- [MLM⁺06] MACKENZIE, C. M. ; LASKEY, Ken ; MCCABE, Francis ; BROWN, Peter F. ; METZ, Rebekah: Reference Model for Service Oriented Architecture 1.0. In: *OASIS Standard, 2006*. – <http://docs.oasis-open.org/soa-rm/v1.0/>
- [RR09] RIEGEN, Michael von ; RITTER, Norbert: Reliable Monitoring for Runtime Validation of Choreographies. In: *The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, IEEE Computer Society, 5 2009
- [ST07] STARKE, Gernot ; TILKOV, Stefan: *SOA Expertenwissen*. Dpunkt.Verlag GmbH, 2007. – ISBN 3898644375
- [WCL⁺05] WEERAWARANA, Sanjiva ; CURBERA, Francisco ; LEYMANN, Frank ; STOREY, Tony ; FERGUSON, Donald F.: *Web Services Platform Architecture*. 2nd. Prentice Hall, 2005. – ISBN 0131488740
- [ZW88] ZWAHR, Annette ; WECK, Helga: *BI Universallexikon [in 5 Bd.]*. 2. Auflage. VEB Bibliographisches Institut Leipzig, 1988. – ISBN 3323002067
-

Abbildungsverzeichnis

1.1	Schematische Darstellung der Web Services, die über den Enterprise Service Bus kommunizieren	2
2.1	Web-Services-Dreieck, vgl. [Mel08], S. 56	8
2.2	UDDI und WS-Inspection, vgl. [Mel08], S. 135	9
2.3	Schematische Darstellung des Enterprise Service Busses (WSO2)	18
3.1	Schematischer Ablauf des EPEJ-Beispiels	22
3.2	Kommunikation im EPEJ-Beispiel über einen ESB (Ausschnitt)	23
3.3	Synchrone und asynchrone Kommunikation über den ESB	24
3.4	Fehlgeschlagene Anfrage über den ESB	27
3.5	Antwort geht am ESB vorbei, wenn die Header erhalten bleiben	28
4.1	Übersicht über die Anordnung der Mediatoren	29
4.2	Aufruf-Hierarchie des Send-Mediators in Synapse	36

Listings

2.1	Beispiel WSDL 2.0 Dokument	11
2.2	Struktur einer SOAP-Nachricht	13
2.3	WS-Addressing	15
2.4	Grundgerüst eines Mediators in Java	19
3.1	Erweiterung des SOAP-Headers durch das ChorMon-Framework	23
3.2	Falsche ReplyTo-Adresse im WS-Addressing Header	25
4.1	Ausschnitt aus der <i>mediate(...)</i> -Funktion des RequestMediators	31
4.2	Ersetzung der WS-Addressing Informationen	32
4.3	Mediatorkonfiguration für den RequestProxy	33
4.4	Ausschnitt aus der <i>mediate(...)</i> -Funktion des ResponseMediators	34
4.5	Auszug aus den Methodensignaturen des <i>Axis2FlexibleMEPClient</i> s	36
4.6	Aufruf von statischen Methoden in Basisklassen	37
4.7	In-Sequence des ELOProxy-Dienstes im ESB	38
4.8	In-Sequence des ResponseProxy-Dienstes im ESB	39
4.9	WS-Addressing Header nach der Ersetzung durch den RequestMediator	39
4.10	Header der Antwort nach Ersetzung durch den ResponseMediator	40

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____