TUHH

Technische Universität Hamburg-Harburg

Entwurfskonzepte und Implementierungsstrategien für das WS-BusinessActivity Framework

vorgelegt von

Simon Zambrovski
simon@zambrovski.org

im Rahmen einer Studienarbeit

Hamburg, 31. Juli 2004

0.1 Vorwort

Diese Arbeit wurde im Rahmen einer Studienarbeit an der Technischen Universität Hamburg-Harburg entwickelt. An dieser Stelle möchte ich mein Tutor M.Sc. Muhammad Fahrat Kaleem, Professor Dr. Friedrich Vogt, Dr. Markus Venzke und Boris Gruschko für ihre Hilfe danken.

0.2 Struktur der Arbeit

Jedes Kapitel fängt mit einer kurzen Einleitung an und endet mit einer Zusammenfassung.
Das Kapitel 1 gibt eine kurze Einleitung und definiert die Fragestellung. Der Systemaufbau wird im Kapitel 2 erläutert. Das dritte Kapitel gibt einen Überblick über die verwendeten Technologien und Produkte. Im letzen Kapitel ist eine Zusammenfassung der Arbeit gegeben. Im Anhang sind Listings und ein Literaturverzeichnis zu finden.

Inhaltsverzeichnis

0.1 Vorwort i

0.2 Struktur der Arbeit i

1 Einleitung **1**

1.1 Motivation 1

1.2 Zielsetzung 2

1.2.1 Konzeptuelle Anforderungen 2

1.3 Die Spezifikationen 2

1.4 Die Protokolle 3

1.4.1 Business Activity With Participant Completion 3

1.4.2 Business Activity With Coordinator Completion 4

1.5 Kommunikationsmodelle 4

1.5.1 Asynchrone Kommunikation 4

1.5.2 Kommunikation mit Bestätigungen 4

2 Systemaufbau **7**

2.1 Erweiterung der Spezifikationen 7

2.1.1 Initialisierung und Terminierung 8

2.1.2 Registrierung 8

2.1.3 Übermittlung der Entscheidungen des Initiators 9

2.1.4 Koordinationsprotokolle 10

2.1.5 Zusammenfassung 11

2.2 Der Proxy 11

2.2.1 Proxy Client 11

2.2.2 Proxy Dienst 12

2.2.3 Zusammenfassung 14

2.3 Webservice Seite 14

2.3.1 Vorbedingungen 14

2.3.2 Participant-Dienst 15

2.3.3 Zusammenfassung 17

2.4 Client Seite 17

2.4.1 Einsatzszenarien 17

2.4.2 Zusammenfassung . . . 18
2.5 Middleware Dienst . . . 18
2.5.1 Transaktionsdienst . . . 19
2.5.2 Activation Komponenten . . . 20
2.5.3 Registration Komponenten . . . 20
2.5.4 Koordinationsprotokolllogik und Komponenten . . . 20
2.5.5 Zusammenfassung . . . 21
2.6 Monitoring und Logging . . . 21
2.6.1 Monitoring Komponenten . . . 21
2.6.2 Logging . . . 21
2.6.3 Zusammenfassung . . . 22

3 Implementierung **25**
3.1 Implementierungsumgebung . . . 25
3.2 Verwendete Technologien und Systeme . . . 25
3.2.1 Apache AXIS . . . 25
3.2.2 De-/Serializierung . . . 26
3.2.3 JBoss Applikationsserver . . . 26
3.3 Beispielszenario . . . 27
3.3.1 Travel Agency System . . . 27

4 Zusammenfassung **31**
4.1 Die Spezifikationen . . . 31
4.2 Die Implementierung . . . 31
4.3 Stellung im Gesamtkonzept . . . 31
4.4 Ausblick . . . 32

A Listings **33**
A.1 Proxy Dienst WSDL . . . 33
A.2 Transaction Dienst WSDL . . . 34
A.3 Transaction Dienst XML-Schema . . . 37
A.4 Airline Dienst WSDL . . . 39
A.5 Airline Dienst XML-Schema . . . 42
A.6 Hotel Dienst WSDL . . . 45
A.7 Hotel Dienst XML-Schema . . . 48

B Literaturverzeichnis **52**

Abbildungsverzeichnis

1.1 Gliederung der Webservices Spezifikationen 1
1.2 Business Activity With Participant Completion 3
1.3 Business Activity With Coordinator Completion 4

2.1 Systemübersicht 7
2.2 Inaktiver Proxy Client ist transparent 12
2.3 Aktiver Proxy Client lenkt den Nachrichtenfluß um 13
2.4 Participant-Dienst Komponente 15
2.5 Client 17
2.6 Middleware-Dienst Komponente 18
2.7 Benutzeroberfläche des WS-BA Monitoring Clients 22
2.8 Zustände der Protokollinstanz 23

3.1 Kommunikation bei einer Bestellung und einer anschließenden Reklamation 29

Listings

2.1 Register Nachricht 9
2.2 RegisterResponse Nachricht 10
2.3 Completed-Nachricht 11
2.4 Proxy Response Header 12
2.5 Proxy Header 13
2.6 confirm-Nachricht 19
2.7 cancel-Nachricht 20
A.1 Proxy Dienst WSDL 33
A.2 Transaction Dienst WSDL 34
A.3 Transaction Dienst XML-Schema 37
A.4 Airline Dienst WSDL 39
A.5 Airline Dienst XML-Schema 42
A.6 Hotel Dienst WSDL 45
A.7 Hotel Dienst XML-Schema 48

Kapitel 1

Einleitung

1.1 Motivation

Die durch SOAP[6] und WSDL[10] beschriebenen Webservices stellen die Basisschicht einer verteilter Umgebung zur Verfügung. Mit der Entwicklung weiterer Standards werden zusätzliche Funktionalitäten bereitgestellt, die für den Produktiveinsatz der verteilten Systeme benötigt werden und den Umstieg auf die Webservice-Technologie überhaupt erst möglich machen. Eine ganze Palette dieser Standards wurde in der letzten Zeit aufgebaut. Die folgende Abbildung schildert die Gliederung der Spezifikationen[1].

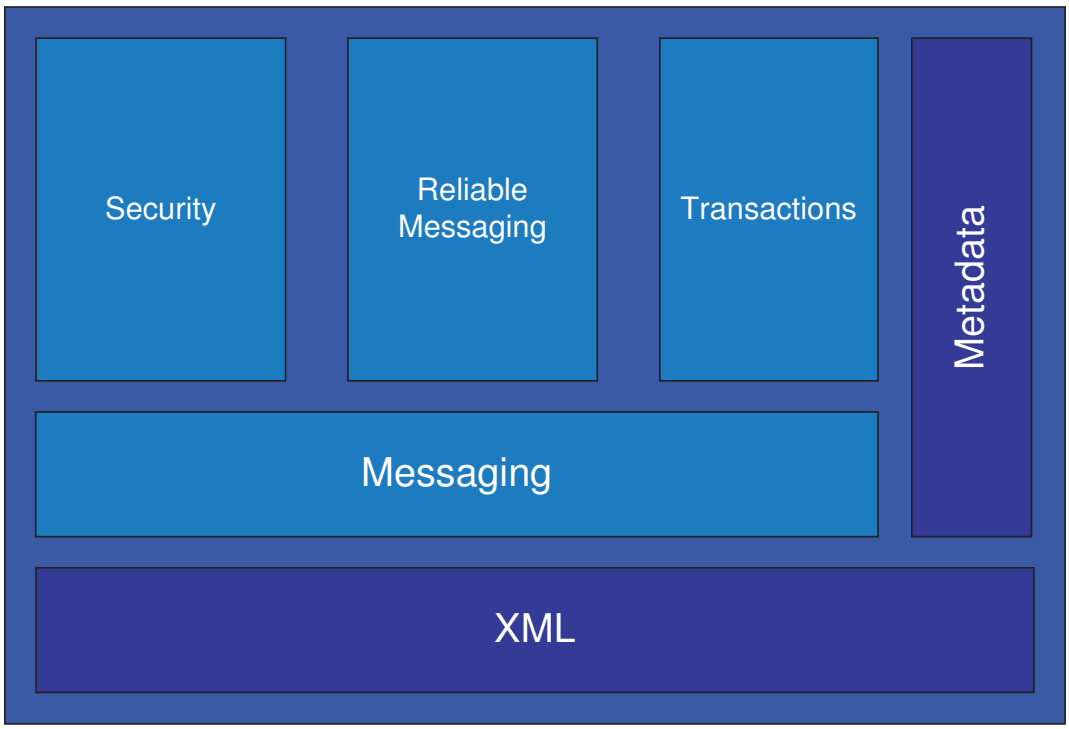

Abbildung 1.1: Gliederung der Webservices Spezifikationen

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es Konzepte für die Komponenten zu entwickeln, die die in den Spezifikationen WS-Coordination[7] und WS-BusinessActivites[9] erleuterten Funktionalitäten zur Verfügung stellen. Ferner ist eine Beispielimplementierung in Form eines Frameworks bereitzustellen.

1.2.1 Konzeptuelle Anforderungen

Das aufgebaute Framework sollte die in den Spezifikationen beschriebenen Protokolle unterstützen und die definierten Komponenten zur Verfügung stellen. Es sollte ermöglichen bestehende Webservices und Clients mit der Fähigkeit auszustatten, auf die in der WS-BusinessActivity Spezifikation beschriebene Weise miteinander zu kommunizieren, wobei die benötigten Veränderungen an bereits existierenden Komponenten zu minimieren sind. Ferner sind Konzepte zu entwickeln, die unabhängige Implementierung der Transactions- und Koordinationskomponenten von den applikationsspezifischen Komponenten ermöglichen.

1.3 Die Spezifikationen

Eine der wichtigsten Eigenschaften eines verteilten Systems ist dessen transaktionales Verhalten. Diese Eigenschaft bestimmt oft das Einsatzgebiet und die Einsatzmöglichkeiten eines Systems. Unter verteilten Transaktionen versteht man eine Serie von Aktionen verschiedener Teilnehmer, die zum Erreichen einer Übereinkunft durchgeführt wird. Um in der verteilten Umgebung zu einer Übereinkunft zu kommen, benötigt man gewisse Koordination. Diese Koordination wird im Bereich von Webservices durch die Spezifikation WS-Coordination beschrieben.
WS-Coordination definiert drei Rollen für die Kommunikationsteilnehmer und deren Nachrichtenaustausch. Sie definiert den Initiator als eine Instanz, die einen Konsenz anstrebt und den Participant als eine Instanz, die koordiniert werden soll, wobei die Koordination durch einen Dienst gesteuert wird, der in der Rolle des Coordinators aggiert. Der Coordinator besteht aus einem Activation Service und einem Registration Service. Nach WS-Coordination fordert der Initiator zuerst einen Koordinationskontext beim Activation Service an. Diesen Koordinationskontext, der den Konsens eindeutig identifiziert, fügt er an jede Nachricht an, die an die Webservices versandt werden. Bekommt ein Webservice eine Nachricht mit einem Koordinationskontext, kann er an dem Konsens teilnehmen. Für diese Teilnahme registriert sich der Participant, der auf der Seite des Webservices eingesetzt wird beim Registration Service. Bei der Registrierung wird das Koordinationsprotokoll festgelegt und die Adressen der Protokollinstanzendpunkten werden ausgetauscht. Verschiedene Kommunikationsszenarien bedürfen verschiedene Koordination. Diese Vielfältigkeit wird im WS-Coordination Modell durch austauschbare Koordinationsprotokolle erreicht.

Die Koordination der Teilnehmer wird insbesonders für die verteilten Webservice Transaktionen benötigt, die als logische Fortsetzung der proprietären verteilten Transaktionen

gesehen werden. Webservice Transaktionen stellen langlebige Aktivitäten sowie den Einsatz über die Vertrauensgrenzen hinweg zur Verfügung. Ein weiterer Unterschied liegt in der Verwendung verschiedener Kommunikationsmodelle[2].

In WS-AtomicTransactions[8] und WS-BusinessActivity[9] Spezifikationen werden konzeptionelle Ansätze zur Koordination von kurz- und langlebigen Transaktionen erläutert und die nötigen Koordinationsprotokolle definiert. Diese Spezifikationen bauen auf der WS-Coordination Spezifikation auf und ergänzen diese mit den entsprechenden Nachrichten. Im Unterschied zu den proprietären verteilten Transaktionsprotokollen basieren die WS-BusinessActivity Protokolle auf dem asynchronen Kommunikationsmodell (siehe Unterabschnitt 1.5.1).

Die WS-BusinessActivity Spezifikation stellt zwei Koordinationsprotokolle zur Verfügung: 'Business Activity With Coordinator Completion' (BACC) und 'Business Activity With Participant Completion' (BAPC). Es handelt sich dabei um zweiphasige Protokolle, an denen jeweils zwei Kommunikationsteilnehmer beteiligt sind. Eine Protokollinstanz wird dabei von dem Coordinator und eine vom Participant verwaltet.

1.4 Die Protokolle

1.4.1 Business Activity With Participant Completion

Das Business Activity With Participant Completion Protokoll (BAPC) ist das erste in WS-BusincessActivity definierte Protokoll. Es wird in den Fällen verwendet, wenn der Webservice entscheiden kann, wann seine Arbeit beendet ist. Die Ergebnisse werden persistent gespeichert und der Participant informiert den Koordinator mit Hilfe der completed-Nachricht über das Ende der Verarbeitung.

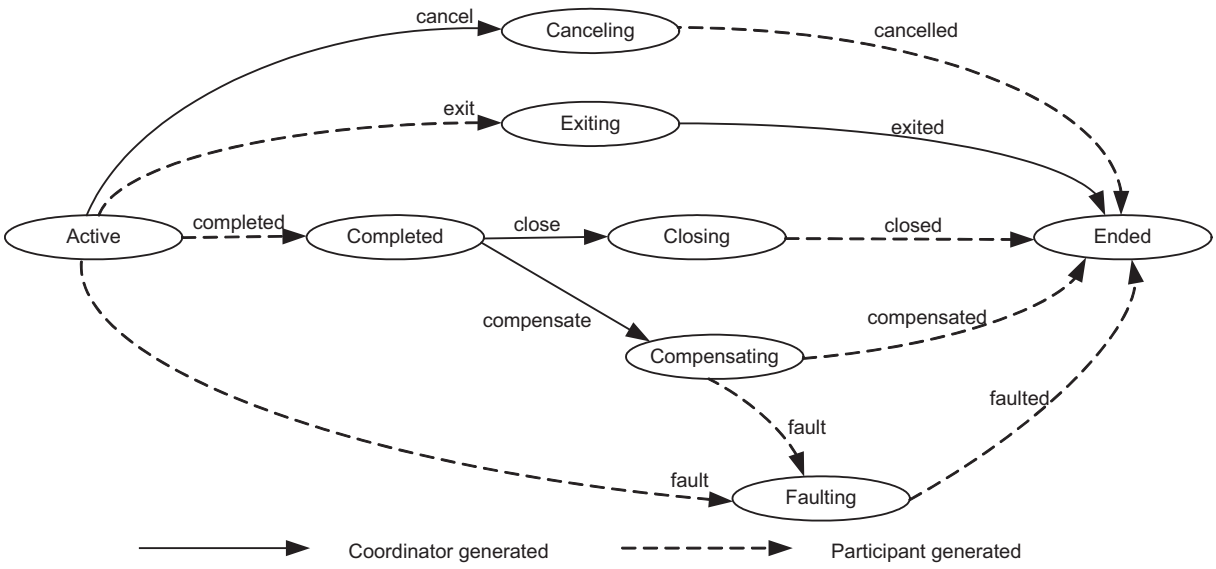

Abbildung 1.2: Business Activity With Participant Completion

1.4.2 Business Activity With Coordinator Completion

Das Business Activity With Coordinator Completion Protokoll (BACC) ist das zweite in WS-BusinessActivity definierte Protokoll. Es wird in den Fällen verwendet, wenn die Entscheidung über das Ende der Verarbeitung dem Koordinator überlassen wird. Der Koordinator sendet die complete-Nachricht, mit der er den Webservice informiert, daß seine Arbeit zu beenden ist und die Ergebnisse persistent zu speichern sind. Der weitere Aufbau der Protokolllogik entspricht der des BAPC Protokolls.

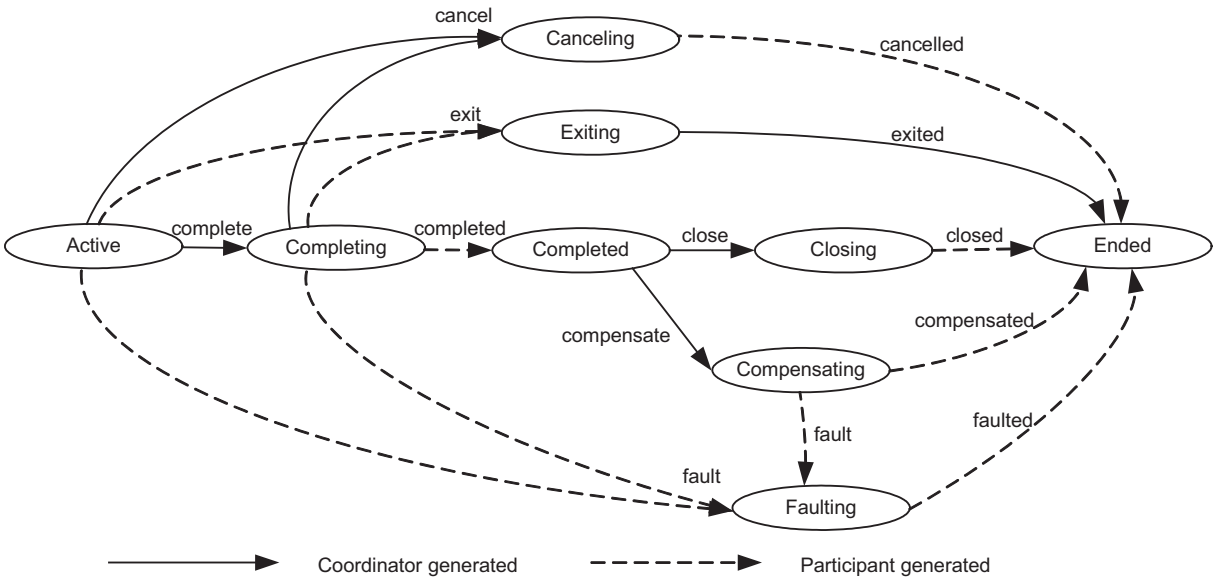

Abbildung 1.3: Business Activity With Coordinator Completion

1.5 Kommunikationsmodelle

1.5.1 Asynchrone Kommunikation

Der in WS-Coordination beschriebene Nachrichtenaustausch basiert auf einem asynchronen Kommunikationsmodell. Das bedeutet, daß die Adresse des Clients, der eine Operation eines Dienstes aufruft, als Parameter an den Dienst übermittelt wird. Nach der Ausführung der Operation ruft der Dienst eine Operation des Clients auf, die das Ergebnis als Parameter annimmt. Bei der Implementierung des asynchronen Kommunikationsmodells sind sowohl die Dienstschnittstellen, als auch die Client Schnittstellen zu entwickeln.

1.5.2 Kommunikation mit Bestätigungen

Bei den in WS-BusinessActivity beschriebenen Protokollen werden die Entscheidungen über die Zustandsübergänge alternierend auf beiden Seiten getroffen. Diese Eigenschaft

wird von einer Partei dazu benutzt, den Empfang einer Nachricht von der anderen Partei als eine Bestätigung[1] für den Empfang eigener, zuvor gesendeter Nachricht anzusehen.

[1] aus dem Englischen - Acknowledge

Kapitel 2

Systemaufbau

Als erstes werden die für die Implementierung benötigten Spezifikationserweiterungen aufgedeckt und begründet. Im Rahmen dieser Arbeit wurde das Proxy-Konzept entwickelt. Dieses wird vorgestellt und dessen Notwendigkeit erläutert. Anschließend wird der Systemaufbau erörtert. Dieser gliedert sich entsprechend dem Einsatzort der einzelnen Systemkomponenten. Entsprechend wird zwischen der Webservice Seite, der Client Seite und der Middleware Seite unterschieden. Dieser Zusammenhang ist in der Abbildung 2.1 dargestellt.

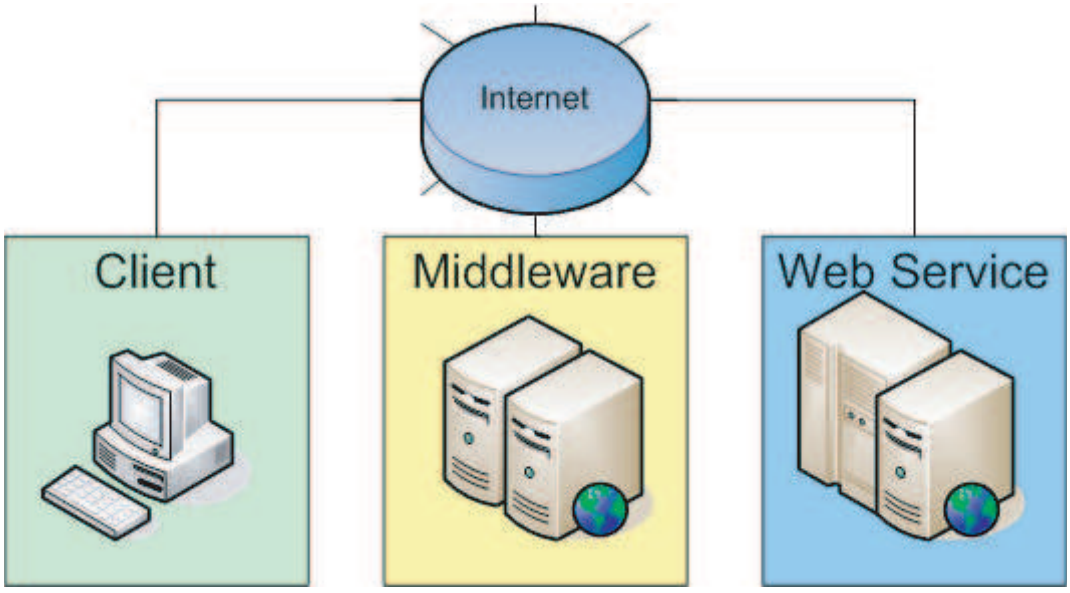

Abbildung 2.1: Systemübersicht

2.1 Erweiterung der Spezifikationen

Das in WS-Coordination und in WS-BusinessActivity vorgestellte Kommunikationsmodell lässt an einigen Stellen Fragen offen, deren Antworten implementierungsabhängig sind.

Um die spätere Spezifikationserweiterung zu ermöglichen, wurden die Elemente, die in den Nachrichten verwendet werden, von den Autoren erweiterbar definiert. Im Rahmen dieser Arbeit wurde Gebrauch von dieser Möglichkeit gemacht. In diesem Abschnitt wird eine Implementierungsstrategie vorgestellt, die zeigt, wie die Antworten auf diese Fragen aussehen könnten. Diese Entscheidungen bilden die Grundlage für die Implementierung.

2.1.1 Initialisierung und Terminierung

Der durch WS-Coordination definierte Nachrichtenfluss erfordert von den Teilnehmern der Kommunikation Verständnis der Kommunikationssemantik und Unterstützung der vorgestellten Protokolle. Die Implementierungsstrategie, die im Rahmen dieser Arbeit vorgestellt wird, legt einen besonderen Wert auf den Erhalt der Kompatibilität der Schnittstellen und die Wiederverwendbarkeit der bereits existierenden verteilten Komponenten, den Webservices und den Clients. Um eine stufenlose Einführung der Technologie zu ermöglichen, wurden für die existierenden Komponenten im Rahmen dieser Arbeit Mechanismen vorgesehen, die nur bei Bedarf die Unterstützung der WS-BusinessActivity aktivieren. Dafür wird das Kommunikationsmodell um einige Nachrichten erweitert und ein spezieller Dienst eingeführt, der diese Nachrichten akzeptiert. Mit der **begin**-Nachricht kann der Client signalisieren, daß eine koordinierte Kommunikation stattfinden soll. Das Ende dieser Kommunikation wird mit einer **end**-Nachricht gekennzeichnet. Die genauere Semantik dieser Nachrichten wird im Abschnitt Transaktionsdienst (Abschnitt 2.5.1) erläutert. Für den Webservice dient die Präsenz des Koordinationskontextes im SOAP-Header einer Nachricht als einziges Kriterium dafür, daß eine koordinierte Kommunikation stattfindet.

2.1.2 Registrierung

Die WS-Coordination Spezifikation schreibt vor, daß der Participant sich für die Teilnahme an der Business Activity bei einem Coordinator für eins der Koordinationsprotokolle registrieren muss. Die Adresse des Coordinators und den Koordinationstyp kann der Participant aus dem Koordinationskontext entnehmen. Für die Registrierung ruft der Participant eine Register-Operation auf dem Registration Service auf. Bei diesem Nachrichtenaustausch fungiert der Participant in der Rolle des Registration Requesters. Die in der Spezifikation beschriebene Registrierungsnachricht erhält nicht ausreichend Information für den Registration Service, um eindeutig festzustellen, für welche Business Activity der Participant sich registrieren möchte.

Um benötigte Information an den Koordinator zu übermitteln, wurde die Registrierungsnachricht im Rahmen dieser Arbeit um ein **Identintifikationselement aus dem Koordinationskontext** und eine **Adresse des eigentlichen Applikationswebservices** erweitert. Die Letztere wird von dem Coordinators benötigt, um die Entscheidungen des Clients über den Ausgang der Business Activity an den korrespondierenden Protokollendpunkt zu übermitteln.

```
<wsc:Register
    xmlns:wsc="http://schemas.xmlsoap.org/ws/2003/09/wscoor"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
>
    <wsc:ProtocolIdentifier>
        http://schemas.xmlsoap.org/ws/2004/01/wsba/
            BusinessAgreementWithParticipantCompletion
    </wsc:ProtocolIdentifier>
    <wsc:RequesterReference>
        <wsa:Address>http://wst.zambrovski.org/
            RegistrationRequesterPort1</wsa:Address>
    </RequesterReference>
    <wsc:ParticipantProtocolService>
        <wsa:Address>http://wst.zambrovski.org/BAPC-Participant1</
            wsa:Address>
    </wsc:ParticipantProtocolService>
    <wsu:Identifier>http://wst.zambrovski.org/?identifier=1</
        wsu:Identifier>
    <wsa:EndpointReference>
        <wsa:Address>http://wst.zambrovski.org/Service1</wsa:Address>
    </wsa:EndpointReference>
</wsc:Register>
```

Listing 2.1: Register Nachricht

Nach der WS-Coordination Spezifikation wird bei der Registrierung asynchron vorgegangen. Die Antwort auf die **register**-Nachricht erfolgt also asynchron, womit die Information für die eindeutige Zuordnung benötigt wird. Die registerResponse-Nachricht enthält diese Information nicht. Dieses führt bei der Teilnahme an mehreren Business Activities, verwaltet von verschiedenen Coordinator Diensten, auf das Problem der eindeutigen Zuordnung der Antwort der Anfrage. Die **registerResponse**-Nachricht wird daher um ein **Identintifikationselement aus dem Koordinationskontext** erweitert.

2.1.3 Übermittlung der Entscheidungen des Initiators

Die zu einer Business Activity gehörenden Protokollinstanzen werden einerseits auf dem Koordinator, andererseits auf dem Participant verwaltet. Sowohl bei dem BAPC (Abschnitt 1.4.1) als auch bei dem BACC (Abschnitt 1.4.2) existiert der **completed**-Zustand. Es existieren zwei Zustandsübergänge, die aus diesem Zustand möglich sind. Der **close**-Übergang signalisiert den Participanten darüber, daß die durchgeführte Arbeit bestätigt wird. Der **compensate**-Übergang signalisiert, daß eine Operation auf der Seite des Webservices durchgeführt werden soll, die die durchgeführten Arbeiten rückgängig macht. Beide Nachrichten werden von dem Koordinator verschickt. Der Koordinator hat im Allgemeinen keine Kenntnisse über die interne Daten- und Applikationslogik des Webservices. Darüberhinaus kennt er nicht die semantische Bedeutung der Nachrichten, die zwischen

```
<wsc:RegisterResponse
    xmlns:wsc="http://schemas.xmlsoap.org/ws/2003/09/wscoor"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
>
    <wsc:CoordinatorProtocolService>
        <wsa:Address>
            http://wst.zambrovski.org/BAPC-Coordinator1
        </wsa:Address>
    </wsc:CoordinatorProtocolService>
    <wsc:RequesterReference>
        <wsa:Address>
            http://wst.zambrovski.org/RegistrationRequesterPort1
        </wsa:Address>
    </wsc:RequesterReference>
    <wsu:Identifier>
        http://wst.zambrovski.org/?identifier=1
    </wsu:Identifier>
</wsc:RegisterResponse>
```

Listing 2.2: RegisterResponse Nachricht

dem Inititator und dem Participant ausgetauscht werden.

Die Informationen über die Semantik der Nachrichten besitzt der Client. Aufgrund dieser Informationen ist er in der Lage für jede Protokollinstanz über das weitere Vorgehen des Webservices zu entscheiden.

In Rahmen dieser Arbeit wird ein Dienst entwickelt, welcher ermöglicht die Entscheidungen des Clients an den Koordinator zu übermitteln. Ferner wird das Kommunikationsmodell um einige Nachrichten erweitert. Der in dem Unterabschnitt 2.1.1 eingeführter Transaktionsdienst kann diese Aufgabe übernehmen. Die genaue Semantik der Nachrichten wird im Abschnitt 2.5.1 vorgestellt.

2.1.4 Koordinationsprotokolle

Während der Registrierungsphase werden die Protokollendpunkte von Participant und Coordinator ausgetauscht. Dabei kann ein Participant an mehreren Business Activities gleichzeitig teilnehmen und ein Coordinator mehrere Business Activities gleichzeitig verwalten. Es existieren mehrere Strategien, wie die Zuordnung der Nachrichten, die die Zustandsübergänge wiederspiegeln, den richtigen Protokollinstanzen ermöglichen. Das kann durch die unterschiedlichen Endpunkte oder durch die Erweiterung der Nachrichten mit den Identifikationselementen garantiert werden. Im Rahmen dieser Arbeit wurde die Strategie gewählt die Nachrichten zu erweitern. Der Listing 2.3 verdeutlicht diesen Zusammenhang anhand einer **completed-Nachricht**, die von einem Participant an den Coordinator versandt wird

```
<wsba:Completed
    xmlns:wsba"http://schemas.xmlsoap.org/ws/2004/01/wsba"
    xmlns:wsc="http://schemas.xmlsoap.org/ws/2003/09/wscoor"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
>
    <wsu:Identifier>http://wst.zambrovski.org/?identifier=1</
        wsu:Identifier>
    <wsc:ProtocolIdentifier>
         http://schemas.xmlsoap.org/ws/2004/01/wsba/
             BusinessAgreementWithParticipantCompletion
    </wsc:ProtocolIdentifier>
    <wsa:Address>http://wst.zambrovski.org/Service1</wsa:Address>
    <wsa:Address>http://wst.zambrovski.org/BAPC-Coordinator1</
        wsa:Address>
</wsba:Completed>
```

Listing 2.3: Completed-Nachricht

2.1.5 Zusammenfassung

Mit Hilfe der Erweiterungen der Spezifikationen bietet das Kommunikationsmodell eine Grundlage für die Implementierung. Die eindeutige Zuordnung von den Nachrichten bei den Teilnehmern der vorgestellter Protokolle ist gewährleistet. In den Nachrichten der Koordinationsprotokolle wurde eine Auflistung der Parameter verwendet. An dieser Stelle können auch komplexere Datentypen verwendet werden, um die benötigte Information zu übertragen. Die Verwendung Letzterer ändert nichts an dem Gesamtansatz der Erweiterung der Nachrichten und sollte bei einem realen System vor der einfachen Auflistung bevorzugt werden.

2.2 Der Proxy

Um die Komplexität des Clients nicht unnötig zu erhöhen, wird die in WS-Coordination beschriebene Initialisierungsphase auf die Middleware verlagert. Der Nachrichtenfluss zwischen dem Client und den Webservices wird dabei auf die Middleware umgelenkt, wodurch direkte Kommunikation vermieden wird. Für die Zuordnung und Verwaltung verschidener Clients wird eine Abstraktion **Proxy-Verbindung** eingeführt, die die gesamte Kommunikation zwischen dem Client und der Middleware kapselt. Die zeitliche Überlappung von mehreren Proxy-Verbindungen von einem Client ist nicht zulässig.

2.2.1 Proxy Client

Der Proxy Client ist eine im Rahmen dieser Arbeit entwickelte Komponente. Ihr Ziel ist es, Nachrichten an den Proxy Dienst weiterzuleiten. Diese Komponente verwendet das Interceptor Design Pattern[3]. Der Proxy Client ist transparent (inaktiv), bis die begin-

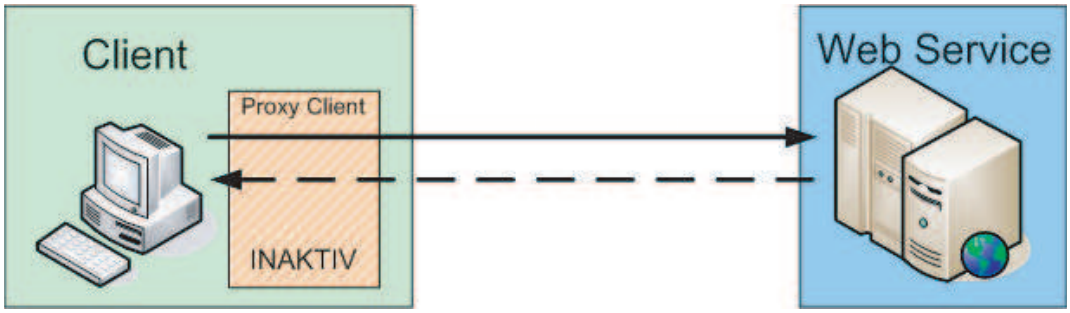

Abbildung 2.2: Inaktiver Proxy Client ist transparent

```
...
 <soapenv:Header>
    <prx:ProxyReference
         xmlns:prx="http://simon.zambrovski.org/wst/proxy"
         xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    >
         <prx:EndpointReference>
              <wsa:Address>
                   http://wst.zambrovski.org/services/ProxyService
              </wsa:Address>
         </prx:EndpointReference>
         <prx:ProxyID>
              17
         </prx:ProxyID>
    </prx:ProxyReference>
 </soapenv:Header>
...
```

Listing 2.4: Proxy Response Header

Nachricht an den Transaktionsdienst verschickt wird. Nach der Aktivierung benötigt der Proxy Client Informationen, wie die Adresse des Proxy Dienstes und die Proxy Client ID. Diese Informationen sind in einem SOAP Header der Antwortnachricht enthalten. Die nachfolgenden Nachrichten, die den Proxy Client passieren, werden an den Proxy Dienst unter der angegebenen Adresse weitergeleitet. Um dabei die Proxy Verbindung eindeutig zu identifizieren, wird die Proxy ID verwendet. Die Adresse des Webservices, für den die Nachricht bestimmt war, wird zusammen mit der ProxyID in den Header der Nachricht angefügt. Das Listing 2.5 verdeutlicht diesen Zusammenhang: In diesem Fall enthält das EndpointReference Element die Adresse des Webservices, der vom Client aufgerufen wurde.

2.2.2 Proxy Dienst

Der Proxy Dienst beschäftigt sich in erster Linie mit der Weiterleitung der Nachrichten. Er bedient sich dazu der Informationen aus dem Proxy SOAP Header. Desweiteren verwaltet

```
...
<soapenv:Header>
    <prx:ProxyReference
        xmlns:prx="http://simon.zambrovski.org/wst/proxy"
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    >
        <prx:EndpointReference>
            <wsa:Address>
                http://wst.zambrovski.org/services/MyApplication1
            </wsa:Address>
        </prx:EndpointReference>
        <prx:ProxyID>
            17
        </prx:ProxyID>
    </prx:ProxyReference>
</soapenv:Header>
...
```

Listing 2.5: Proxy Header

er die Proxy-Verbindungen und fungiert als Coordination Initiator. Für jede aufgebaute Proxy-Verbindung fragt der Proxy Dienst einen Koordinationskontext beim Activation Service an. Dieser Kontext wird dann an alle weitergeleiteten Nachrichten angefügt.

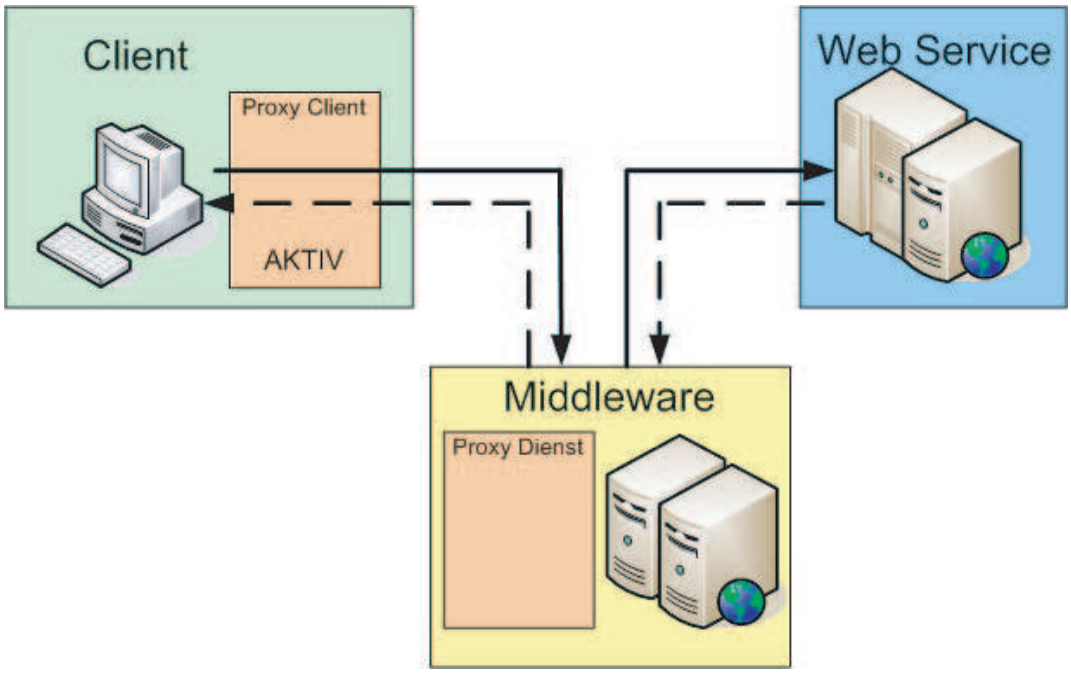

Abbildung 2.3: Aktiver Proxy Client lenkt den Nachrichtenfluß um

Genauere Beschreibungen der Konfiguration und Schnittstellen sind in 2.5.2 aufgeführt. An dieser Stelle wird nur eine Proxy relevante, jedoch nicht Initiator relevante

Konfigurations- und Schnittstellenbeschreibung gegeben.

Proxy Konfiguration

Der Proxy Dienst braucht die Angabe der Adresse, unter der er verfügbar ist. Diese Adresse wird an die Proxy Clients für die Weiterleitung der Nachrichten übermittelt.

Proxy Schnittstelle

Der Proxy Dienst bietet die **proxy** Operation an, die der Proxy Client aufrufen kann. Die vollständige WSDL Beschreibung ist unter A.1 zu finden.

2.2.3 Zusammenfassung

Mit dem Einsatz des Proxy Clients und des Proxy Dienstes wird sichergestellt, daß der gesamte Nachrichtenfluss zwischen dem Client und den Webservices über die Middleware verläuft. Diese Tatsache ermöglicht es benötigte Nachrichtenmanipulationen auf der Middleware Seite durchzuführen. Darüber hinaus wird sichergestellt, daß der Koordinationskontext erzeugt und an jede Nachricht angefügt wird.

2.3 Webservice Seite

Der durch WS-Coordination definierte Participant dient in erster Linie zur Kapselung der Daten über Aktivitäten und der Instanzen der Koordinationsprotokolle. Zusätzlich beschäftigt er sich mit der Zuordnung der internen Datenzustände eines Webservices zu den entsprechenden Zuständen der Koordinationsprotokolle. Die Übermittlung der Koordinationsentscheidungen an den Webservice ist auch die Aufgabe des Participanten. Im Folgendem werden im Rahmen dieser Arbeit entwickelte Komponenten vorgestellt und eine Strategie erläutert, die diese Zuordnung allein aus der Analyse des Nachrichtenaustausches mit dem Webservice ermöglicht. Ferner wird erläutert, wie die Übertragung der Koordinationsentscheidungen ohne eine zusätzliche API zu Stande kommt.

2.3.1 Vorbedingungen

Um die in WS-BusinessActivity definierte Koordinationsprotokolle zu verwenden, wurde im Rahmen dieser Arbeit ein Participant-Dienst entwickelt. Die Verwendung von Business Activity stellt folgende Anforderungen an den Webservice. Ein Webservice muss für die Abstraktion 'Operation' eine 'kompensierende Operation' anbieten, um die Eigenschaft von Konsistenz[1] zu garantieren. Desweiteren stellt die gewählte Strategie zusätzliche Anforderung an den Webservice in Bezug auf Informationsinhalte, die bei einzelnen Nachrichten übermittelt werden, und Lokalität der Ressourcen. Die von dem Webservice ver-

[1] Consistency - als eine der ACID Eigenschaften

sandten Nachrichten müssen ausreichen, um mit Hilfe der prädikativen Ausdrücken den Nachrichten die entsprechende interne Zustände eindeutig zuzuordnen.

2.3.2 Participant-Dienst

Der Participant-Dienst ist eine Softwarekomponente, die an der Seite des Webservices eingesetzt wird, und aus mehreren Komponenten besteht. Zu diesen Komponenten zählen die Abfangskomponente, die Zuordnungskomponente und der dynamischer SOAP-Client. Im Folgenden wird die Funktion und Interaktion von diesen Komponenten beschrieben.

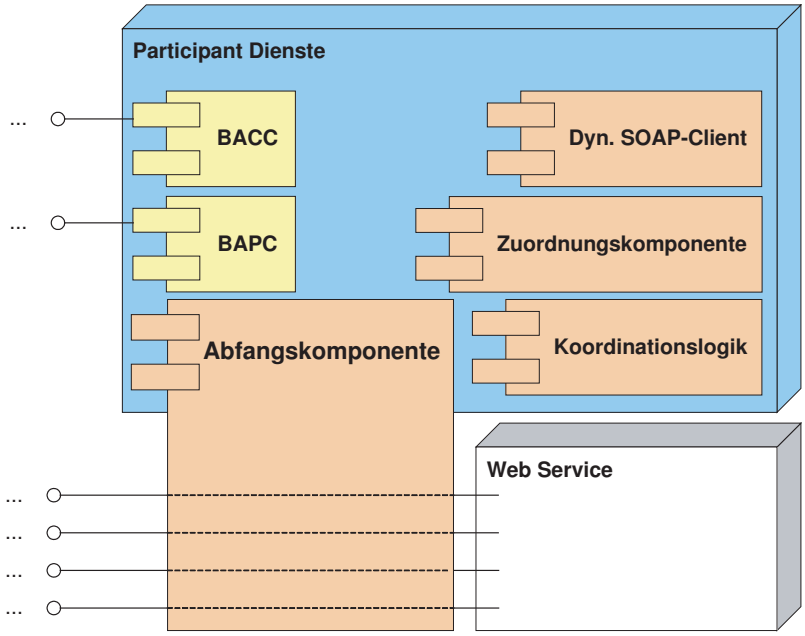

Abbildung 2.4: Participant-Dienst Komponente

Konfiguration

Der Participant-Dienst kann zusammen mit verschiedenen Webservices eingesetzt werden. Dafür wurde im Rahmen dieser Arbeit eine Konfigurationsmöglichkeit vorgesehen. Die Konfiguration in Form von Parametern wird zur Initialisierungszeit eingelesen und stellt die Information über den Webservice zur Verfügung, der koordiniert werden soll. Folgende Parameter werden mit der Konfiguration für jeden Webservice festgelegt: die Adresse des Webservices, die Adresse des Registrierungsendpunktes, die Adresse des Koordinationsprotokolendpunktes sowie das zu verwendene Koordinationsprotokol (BAPC oder BACC). Ferner stellt die Konfiguration Informationen über die Zuordnungskomponente zur Verfügung. Um die lose Kopplung der Zuordnungskomponente zu ermöglichen, wird diese mit Hilfe einer Reflection[2] aufgerufen. Die Konfiguration beinhaltet die Namen der Klassen, die die Schnittstellen implementieren, und den Pfad zur Konfigurationsdateien, die die prädikative Regeln beinhalten.

[2]Unter der Verwendung von Java Reflection API

Schnittstellen

Der Schnittstellenaufbau der Komponenten ist im Wesentlichen durch die Spezifikationen WS-Coordination und WS-BusinessActivity festgelegt. Die Schnittstellen bestehen einerseits aus dem Registration Requester Endpunkt und andererseits aus den Endpunkten der Koordinationsprotokolle BACC und BAPC.

Abfangskomponente

Um die Analyse der Nachrichten, die an den Webservice ankommen oder von dem Webservice verschickt werden, zu ermöglichen wurde von mir eine Abfangskomponente entwickelt. Diese verwendet das Interceptor Design Pattern[3]. Die Abfangskomponente analysiert die in den Nachrichten enthaltenen SOAP-Header, um den Koordinationskontext für den Participant-Dienst zur Verfügung zu stellen. Andererseits werden asynchron zu der Verarbeitung alle Nachrichten an die Zuordnungskomponente weitergeleitet.

Zuordnungskomponente

Die Aufgabe der Zuordnungskomponente ist es, die Nachrichten zu analysieren und Entscheidungen über die Zustandsübergänge der Koordinationsprotokollinstanzen zu treffen. Dazu bedient sich diese von mir entwickelte Komponente einer Konfiguration, die den Übergängen prädikative Ausdrücke zuordnet. Alle Nachrichten werden entsprechend ihrer Auftrittsreihenfolge in einem temporären Behälter gespeichert, wobei bei jeder Nachrichtenankunft der Behälter auf die Gültigkeit der Prädikate überprüft wird. Enthält der Behälter die Nachrichtenfolge, die einem Prädikat genügen findet ein korrespondierender Zustandsübergang des Koordinationsprotokols statt, wonach die betroffenen Nachrichten aus dem Behälter entfernt werden. Die Konfiguration der Zuordnungskomponente ermöglicht es einen weiteren Ausdruck anzugeben, der bei einem Zustandsübergang evaluiert wird. Das Ergebnis dieser Operation wird in dem neuen Zustand gespeichert. Dieser Mechanismus erlaubt es, für den Zustandsübergang relevante Daten für die spätere Verwendung abzuspeichern.
Auch die Umkehrzuordnung wird mit Hilfe der Zuordnungskomponente realisiert. Damit ist es möglich einem Zustandsübergang eine Nachricht oder deren Teil zuzuordnen. Von dieser Fähigkeit macht der Dynamischer SOAP-Client gebrauch.

Dynamischer SOAP-Client

Einige Zustandsübergänge der Protokollinstanz finden auf dem Coordinator statt. Der Coordinator synchroniziert diesen Zustandsübergang mit der korrespondierenden Protokollinstanz beim Participant-Dienst. Die Zuständsübergänge beim Participant-Dienst können die Ausführung weiterer Operationen auf dem Webservice benötigen. Für die Übermittlung dieser Nachrichten an den Webservice wurde in Rahmen dieser Arbeit ein dynamischer SOAP-Client entwickelt. Dieser bedient sich der Informationen aus den zuvor gesendeten Nachrichten (siehe 2.3.1). Aus der Kenntniss dieser Informationen und des Zustandsübergangs wird mit Hilfe der Zuordunungskomponente die Nachricht generiert

und an den Webservices verschickt.

2.3.3 Zusammenfassung

Mit Hilfe des Participant Dienstes, der an der Seite des Webservices eingesetzt wird, ist es für einen Webservice möglich, an einer BusinessActivity teilzunehmen. Dabei kapselt der Participant Dienst die gesamte Koordinations- und Transaktionslogik. Mit Hilfe der in Prädikaten definierten Bedingungen ist es möglich, die Zustandwechsel des Webservices zu identifizieren und die Koordinationsentscheidungen an den Webservice zu übermitteln. Dabei werden keine Veränderungen an den Webservice Schnittstellen oder Einführung einer zusätzlichen API zwischen dem Webservice und dem Participant Dienst benötigt.

2.4 Client Seite

Der Client stellt eine Komponente zur Verfügung, die die applikationsspezifischen Webservice Aufrufe kapselt. Um den von WS-Coordination definierten Nachrichtenaustausch zu gewährleisten, wurde von mir eine Komponente entwickelt, die auf der Client Seite eingesetzt wird. Diese Komponente fängt alle Nachrichten, die vom Client gesendet werden, ab und leitet diese an einen speziell für diese Zwecke erzeugten Proxy-Webservice weiter. Dieses Model wurde im Kapitel Proxy 2.2 beschrieben.

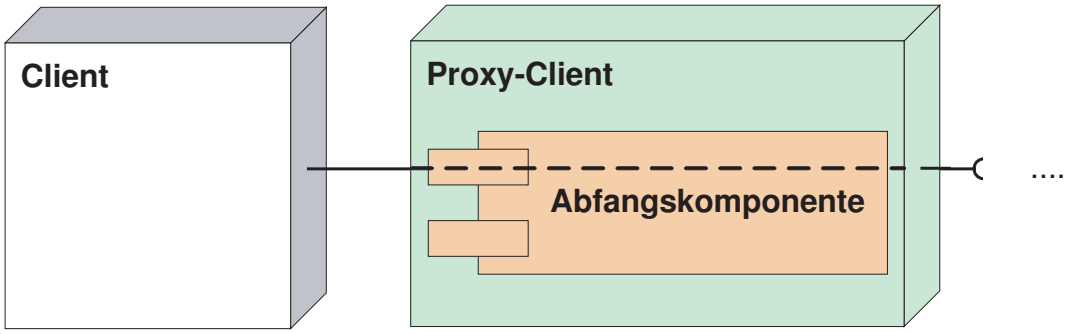

Abbildung 2.5: Client

2.4.1 Einsatzszenarien

Der Client stellt die Basisfunktionalität zur Verwendung von Webservices zur Verfügung. Die folgende Unterscheidung der Einsatzszenarien hat keinen technischen, sondern einen architekturellen Hintergrund.

Client als vom Menschen bedientes Programm

Der Client kann als Teil einer Applikation verwendet werden, die zur Präsentation der Daten oder Steuerung der hinter den Webservices gekapselten Vorgänge dient. Um die

Protokolllogik nicht zu verletzen muss in der Client Applikationslogik gewährleistet werden, daß die Entscheidungen des Clients über jeden Teilnehmer der Business Activity getroffen werden.

Client als Teil eines Composition Providers

Ein anderes Szenario für die Verwendung des Clients ist, ihn als einen Teil eines Webservices zu verwenden, der selbst die Leistungen von anderen Webservices anbietet. Ein solcher Webservice wird Webservice Composition[14] genannt.

2.4.2 Zusammenfassung

Der Client zusammen mit der Proxy Client Komponente ermöglicht die Kapselung der Webservice- und Business Activity Spezifik. Dabei kann die Komponente als Teil eines weiteren Dienstes oder einer Applikation verwendet werden.

2.5 Middleware Dienst

Der Middleware Dienst ist die Kernkomponente des Frameworks, das in Rahmen dieser Arbeit entwickelt wurde. Er beinhaltet den durch WS-Coordination eingeführten Activation Service und den Registration Service. Desweiteren kapselt er die durch WS-BusinessActivity definierte Koordinationsprotokolllogik und stellt den Proxy- und Transaktionsdienst zur Verfügung. Im Abbildung 2.6 wird dieser Zusammenhang dargestellt.

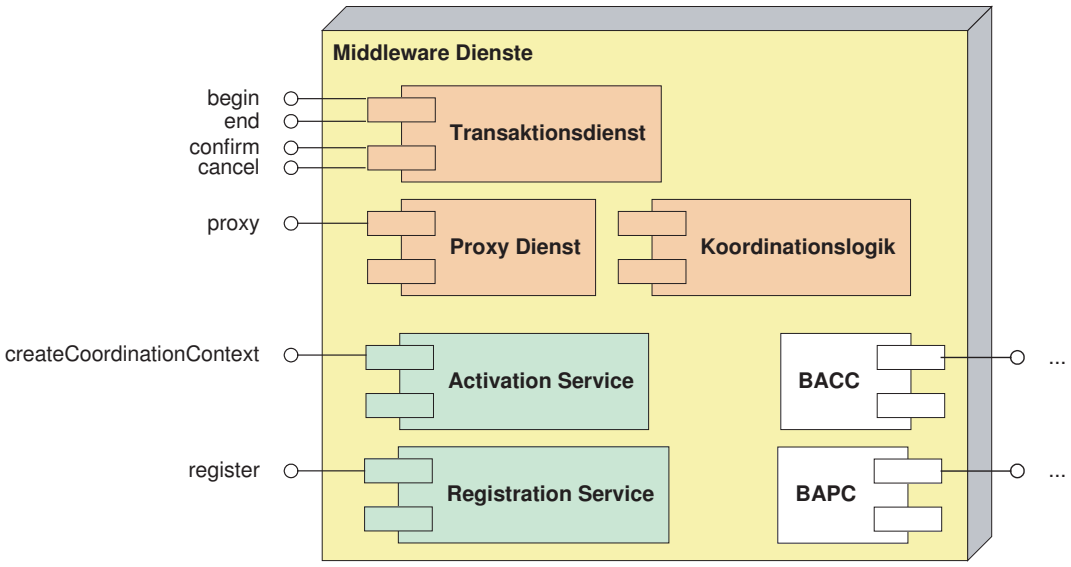

Abbildung 2.6: Middleware-Dienst Komponente

2.5.1 Transaktionsdienst

Der von mir entwickelte Transaktionsdienst stellt dem Client Steuerungsmechanismen zur Verfügung, um den Verlauf der Business Activity zu beeinflüssen. Dazu bietet er vier Operationen an. Mit dem Aufruf der **begin**-Operation beginnt der Client die Arbeit. Mit Hilfe der **end**-Operation kann der Client die Arbeit mit dem Middleware Dienst beenden. Zwei weitere Operationen **confirm** und **cancel** geben dem Client die Möglichkeit für jeden aufgerufenen Webservice seine Entscheidung über den Verlauf der BusinessActivity zu übermitteln. Der im Proxy Modell eingführte Zusammenhang impliziert eine enge physikalische Kopplung des Transaktionsdienstes mit dem Proxy Dienst. Die Schnittstellenbeschreibung in Form eines WSDL-Dokuments (Listing A.2) und die Typbeschreibung in Form eines XML Schema-Dokuments (Listing A.3) sind im Listing Anhang zu finden.

Semantik der begin- und end-Nachrichten

Die begin- und end- Nachrichten dienen zur logischen Kapselung der koordinierten Kommunikation zwischen dem Client und dem Middleware Dienst. Sie werden speziell für den Proxy Dienst, als Träger der benötigten Informationen, und für den Proxy Client benötigt.

Semantik der confirm-Nachricht

Die confirm-Nachricht hat eine ähnliche semantische Bedeutung wie die commit-Nachricht in Two Phase Commit (2PC) Protokoll. Sie signalisiert, daß der Client mit dem Ergebnis der durchgeführeten Operationen zufrieden ist. Im Unterschied zum 2PC commit wird bei der confirm-Nachricht über jede einzelne Participant-Protokollinstanz entschieden und nicht über die gesamte Business Activity. Dieser Unterschied wird dadurch begründet, daß bei Business Activity keine Atomarität gewährleistet wird.

```
<tx:confirm
    xmlns:tx="http://simon.zambrovski.org/2003/transaction/
        webservice"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
>
    <wsa:EndpointReference>
        <wsa:Address>
            http://wst.zambrovski.org/
                MyTargetApplicationEndpoint1
        </wsa:Address>
    </wsa:EndpointReference>
</tx:confirm>
```

Listing 2.6: confirm-Nachricht

Semantik der cancel-Nachricht

Die cancel-Nachricht stellt das Gegenstück zur confirm-Nachricht. Sie ist vergleichbar mit der rollback-Nachricht des 2PC Protokolls, mit den gleichen Unterschieden, wie in dem

Fall der confirm-Nachricht.

```
<tx:cancel
    xmlns:tx="http://simon.zambrovski.org/2003/transaction/
        webservice"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
>
    <wsa:EndpointReference>
        <wsa:Address>
            http://wst.zambrovski.org/
                MyTargetApplicationEndpoint1
        </wsa:Address>
    </wsa:EndpointReference>
</tx:cancel>
```

Listing 2.7: cancel-Nachricht

2.5.2 Activation Komponenten

Die Phase der Erzeugung oder des Imports eines Koordinationskontexts wird in WS-Coordination Aktivierung[3] genannt. Während dieser Phase wird das asynchrone Kommunikationsmodell verwendet. Es kann zwischen dem Activation Service und dem Activation Requester unterschieden werden. Nach den im Kapitel Proxy (2.2) erläuterten Überlegungen übernimmt eine Komponente des Proxy Dienstes die Rolle des Activation Requesters. Für die Erzeugung und Verwaltung der Koordinationskontexte wurde in Rahmen dieser Arbeit ein Aktivierungsdienst entwickelt.

2.5.3 Registration Komponenten

Die Phase der Anmeldung vom Participant für die Teilnahme an einer Business Activity wird in WS-Coordination als Registrierung[4] bezeichnet. Bei der Registrierung findet ein Nachrichtenaustausch zwischen dem Registration Requester und dem Registration Service statt. In der Rolle des Registration Requester agiert der Participant Dienst. Für die Verwaltung der Teilnehmer der Business Activity auf dem Coordinator wurde ein Registrierungsdienst entwickelt. Die Registrierungsnachrichten wurden im Abschnitt 2.1 genau erläutert.

2.5.4 Koordinationsprotokolllogik und Komponenten

Der Aufbau der Logik der Koordinationsprotokolle wird durch die WS-BusinessActivities Spezifikation festgelegt. Für die Verwaltung von mehreren Business Activities auf der Coordinator- bzw. Participant-Seite wurden Registries entwickelt. Diese verwenden generische Hashing-Mechanismen, die eine Zuordnung zwischen den Activity Instanzen und Koordinationskontexten herstellen.

[3] Aus dem englischen Activation

[4] Aus dem englischen Registration

2.5.5 Zusammenfassung

Der im Rahmen dieser Arbeit entwickelte Middlewaredienst beinhaltet alle Komponenten, die in WS-C und WS-BA beschrieben sind. Ferner unterstützt er die konzeptionellen Erweiterungen, die im Kapitel 2.1 vorgestellt wurden.

2.6 Monitoring und Logging

Die WS-Coordination und WS-BusinessActivity Spezifikationen schreiben die persistente Speicherung der Daten vor. Ferner wurden in Rahmen dieser Arbeit Monitoring und Logging Komponenten entwickelt, die die Analyse und die Kontrolle der Komponenten und des Nachrichtenflusses ermöglichen. In diesem Abschnitt werden diese Komponenten vorgestellt.

2.6.1 Monitoring Komponenten

Die Monitoring Komponenten gliedern sich in Server- und Client-Komponente. Die Serverkomponente stellt die Information zur Verfügung. Die Client-Komponente dient zur Präsentation dieser Information.

WS-BA Monitoring Dienst

Serverseitige Komponenten liefern die Informationen über die Koordinationslogik des Coordinators oder des Participants. Für jeden Participant werden Daten über die Teilnahme an den Business Activities und die darin verwendeten Instanzen der Koordinationsprotokolle zur Verfügung gestellt. Diese Daten werden zur Laufzeit in Real-Time abgefragt.

WS-BA Monitoring Client

Für die benutzerfreundliche Präsentation der Daten über die Koordinationsaktivitäten wurde der **WS-BA Monitoring Client** entwickelt. In Abbildung 2.7 ist ein Screenshot der Benutzeroberfläche dargestellt. Der Monitoring Client stellt den Zustand der entsprechenden Protokollinstanz dar. Dunkelgrün wird der aktuelle Zustand gekennzeichnet, hellgrün - die durchlaufende Zustände. Dieser Zusammenhang ist in der Abbildung 2.8 dargestellt.

2.6.2 Logging

Die WS-Coordination und WS-BusinessActivities erfordern von den implementierenden Systemen persistentes Speichern der Daten. Für die Speicherung der applikationsspezifischen Daten ist die hinter dem Webservice verborgene Applikation zuständig. Für die Speicherung der Koordinationsdaten schreiben die Komponenten verschiedene Logs. Die

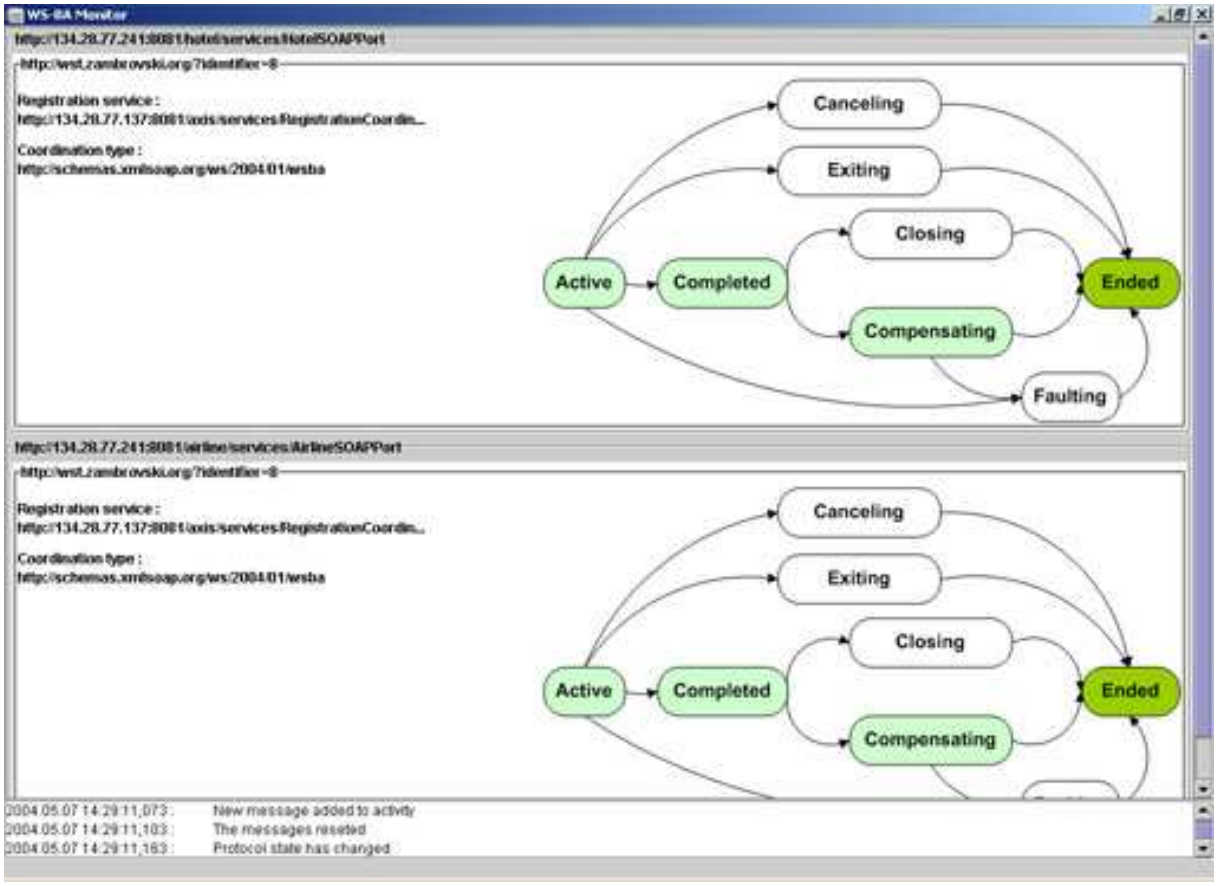

Abbildung 2.7: Benutzeroberfläche des WS-BA Monitoring Clients

Wiederherstellung der Zustände der Koordinationslogik aus den Logs ist kein Gegenstand dieser Arbeit. Die Log-Mechanismen erlauben jedoch eine Kontrolle der Funktionalität und erleichtern die Wartung.

2.6.3 Zusammenfassung

Mit Hilfe der Monitoring und Logging Komponenten wird die Wartung und Unterstützung der im Rahmen dieser Arbeit entwickelten Komponenten vereinfacht. Ferner kann Logging ausgebaut und für die Recovery Zwecke verwendet werden.

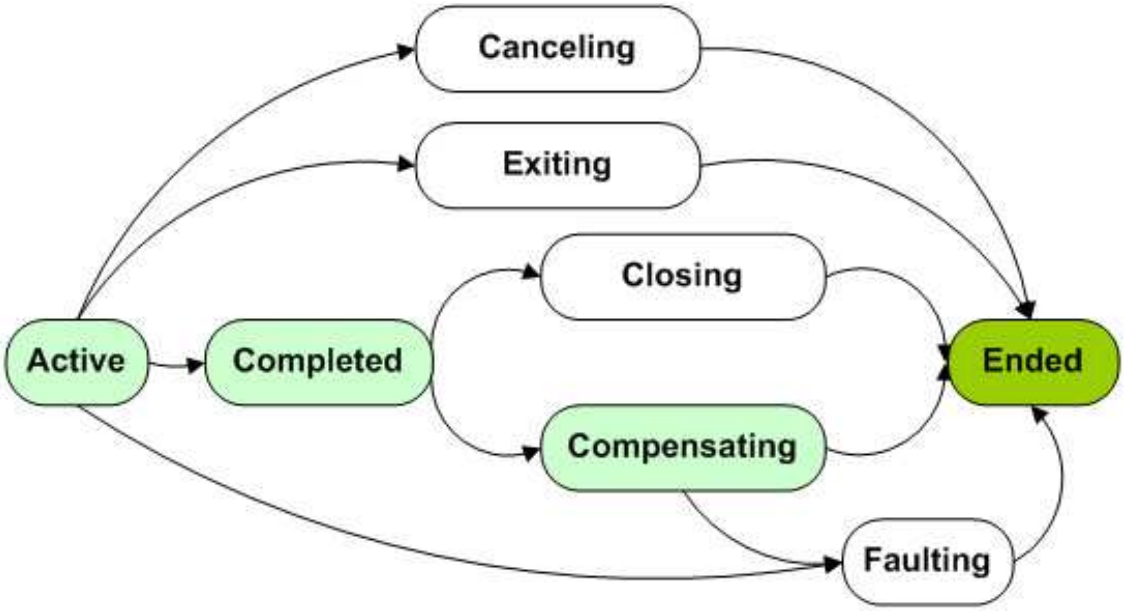

Abbildung 2.8: Zustände der Protokollinstanz

Kapitel 3

Implementierung

In Kapitel 2 wurden die entwickelten Komponenten benannt und ihre Wirkungsweise und Interaktion erläutert. Im folgenden Kapitel werden die Schlüsselaspekte der Implementierung erklärt. Diese umfassen die Implementierungsumgebung, geben einen Überblick über die verwendete Technologien. Anhand eines Beispielszenarios wird erläutert, wie das System eingesetzt wird.

3.1 Implementierungsumgebung

Die Implementierung der im Kapitel 2 vorgestellten Konzepte in Form eines Frameworks wurde im Rahmen dieser Arbeit durchgeführt. Als Implementierungssprache wurde Java gewählt. Dafür wurden die J2SDK 1.4.2 von Sun Microsystems© und die JBoss© J2EE Bibliotheken verwendet.

3.2 Verwendete Technologien und Systeme

Für das im Rahmen dieser Arbeit entwickelte Framework wurden verschiedene Produkte und Bibliotheken benutzt. In diesem Abschnitt werden diese kurz vorgestellt und erläutert, an welchen Stellen die verwendet wurden.

3.2.1 Apache AXIS

Apache AXIS ist ein Opensource Framework, daß für die Implementierung von Webservices gebaut wurde. Es ermöglicht aus WSDL Beschreibungen der Webservice Schnittstelle server- und clientseitige Stubs bzw. Skeletons zu generieren. Mit Hilfe der generierten Stubs kann dann die Implementierung erfolgen. Das Apache AXIS Framework benötigt einen Web-Container als Laufzeitumgebung. Der entwickelte Webservice kann in diesem Container eingesetzt werden. Dabei übernimmt AXIS die De-/Serialializierung von XML-Strukturen, die in den SOAP-Nachrichten enthalten sind.

Handler Konzept

Apache AXIS hat eine besonders für Erweiterungen ausgelegte Architektur. Im allgemeinen besteht AXIS aus vielen Komponenten, die Nachrichten verarbeiten. Diese Komponenten werden Handler genannt und sind in Gruppen, sogenannten Chains, eingeteilt. Jede Nachricht verläuft von ihrer Ankunft über ein Transportprotokoll bis zum Methodenaufruf auf dem Stub eine Reihe von Handlern. Das Ergebnis des Aufrufs durchläuft ebenfalls die Handlerkette, bis es schließlich in Form von SOAP mit Hilfe des Transporthandlers verschickt wird. Auf dem Client wird die Kette in der umgekehrten Reihenfolge durchlaufen.

Die Erweiterbarkeit von AXIS liegt in der Tatsache, daß eigene Handler implementiert werden können. Dies erweist sich als sehr nützliche Eigenschaft für die Verarbeitung von SOAP-Header.

3.2.2 De-/Serializierung

Apache AXIS ist mit einem Satz von De-/Serializern ausgestattet. Es ist dabei möglich auch eigene Serializer und Deserializer zu verwenden. Die Im Rahmen meiner Arbeit verwendete Strukturen erforderten spezielle Serializierung. Für diese Zwecke wurde Castor eingesetzt[12].

3.2.3 JBoss Applikationsserver

Der Web-Container

Für den Einsatz von AXIS wird serverseitig ein Web-Container benötigt. Der JBoss Applikationsserver stellt diesen in Form eines eingebauten Apache Tomcat zur Verfügung.

Asynchrone Verarbeitung

Ferner benötigt die Zuordnungskomponente einen Behälter, in dem die zu analysierende Nachrichten gespeichert werden. Um die relativ teuere Behälteranalyse von der eigentlichen Verarbeitung des Participant-Dienst zu entkoppeln, wurde die JBoss Implementierung des Java Messaging Service (J2EE JMS) verwendet. Die Abfangkomponente versorgt die Message Queue mit abgefangenen Nachrichten. Die dafür entwickelte Message Driven Bean, die an die Message Queue bekoppelt ist, befüllt den Behälter.

Monitoring Komponente

Für die Bereitstellung der Monitoringschnittstellen wurde die JBoss Implementierung von Java Management Extention (J2EE JMX) verwendet. Dabei wurden MBeans entwickelt, die nur die benötigten Daten zur Verfügung stellen. Der Ansatz, die bestehenden

Komponenten mit MBean-Funktionalität auszustatten, wurden aus Performance Gründen verworfen.

Kontextisolierung

Während der Implementierung des Bespielszenarios (siehe Abschnitt 3.3) entstand die Notwendigkeit zwei Webservices auf dem gleichen JBoss Applikationsserver einzusetzen. Bei diesem Einsatz fiel die Problematik der Isolierung verschiedener Web-Container von einander auf. Die JBoss Implementierung erlaubt die vollständige Isolierung auf der Classloader-Ebene nicht für die Web-Container (J2EE WAR), sondern nur für die gesamte J2EE Applikationen (J2EE EAR). Eine solche Isolierung verhindert den Einsatz von Axis. Für die Lösung des Problems wurden die kontextkritischen Komponenten mit zusätzlichen Parametern versehen, die die benötigte Informationen über den Einsatzkontext lieferten. So wurden an einigen Stellen die Doppel-Hashtabellen statt einfachen Hashtabellen verwendet.

Beispiel Webservices

Die für das Beispielszenario entwickelten Dienste repräsentieren eine typische Kapselung der Geschäftslogik auf dem Server. Die Beispieldienste stellen einfache J2EE-basierte Business-Funktionalitäten zur Verfügung. Die Datenabstraktion und Prozesslogikabstraktion in den J2EE EJBs werden als gängige Architektur benutzt. Dabei stellen die Webservices die Kommunikationsplattform zwischen der Präsenationsschicht und der Middlewareschicht zur Verfügung.

3.3 Beispielszenario

Für die Verifizierung der Konzepte und Implementierung wurde im Rahmen dieser Arbeit ein einfaches Test-Szenario entwickelt. Dabei handelt es sich um ein bekanntes Szenario, welches für die transaktionalen Verarbeitungssysteme oft zur Verdeutlichung herangezogen wird.

3.3.1 Travel Agency System

Einleitung

Ein Reisebüro besitzt ein IT-gestütztes Reservierungssystem. Ein Kunde erscheint in diesem Reisebüro. Er wünscht sich eine Reise in eine andere Stadt. Sein Aufenthalt soll mehrere Tage andauern, wobei der Kunde für die Übernachtungen eine Unterkunft braucht. Er äussert ferner den Wunsch mit dem Flugzeug zu seinem Reiseziel zu gelangen.

Die klassischen Problemstellungen

In vielen transaktionsbehafteten Systemen werden ähnliche Beispiele vorgestellt. Dabei wird insbesonders auf die Konsistenz und Ausfallatomarität eingegangen. Es ist leicht verständlich, daß eine Bestellung der Unterkunft und des Fluges nicht gleichzeitig erfolgen kann. Führt man diese Bestellungen sequentiell ohne Transaktionsschutz durch, läuft man Gefahr, daß nach dem Bestellen des Fluges kein Hotel mehr vorhanden ist oder vice versa. Diese Probleme werden mit Hilfe gängiger Koordinationsprotokolle, wie z.B. 2PC gelöst.

Problemstellungen im Kontext von Webservice

Die Verwendung von gängigen Transaktionsprotokollen ist im Kontext von Webservices nur zu einem eingeschränkten Teil möglich. Für die generischen Transaktionprotokolle sind vor allem zwei Eigenschaften des Systems von enormer Bedeutung. Zum Einen muss die Durchführung der Verarbeitung schnell passieren, um die Locking Strategien effizient zu nutzen. Zum Zweiten muss ein hohes Maß an Vertrauen zwischen den Teilnehmern der Kommunikation exisitieren. Eine oder beide diese Bedingungen können im Kontext der Webservice Kommunikation verletzt werden.

Verwendung von WS-BusinessActivity

Das Beispielszenario wird für die Schilderung der Vorteilen von WS-BusinessActivity Idee auf folgender Weise umformuliert.

Ein IT-gestütztes Hotelreservierungssystem und Fluguchungssystem bieten den Reisebüros Bestellungsmöglichkeiten über Webservice-Interface an (siehe Listings A.4 und A.6). Um transaktionell geschützte Bestellungen zu unterstützen, setzen das Hotelreservierungssystem und Flugbuchungssystem die in WS-BA beschriebenen Participant Komponenten ein. Das Reisebüro verwendet die im Rahmen dieser Arbeit vorgestellten Middleware Dienste um eine Bestellung des Hotels und des Flüges zu koordinieren. Die zwischen dem Reisebüro und den Flug- und Hotelreservierungssystemen im Folge einer Reservierung stattgefundene Kommunikation ist in der Abbildung 3.1 dargestellt.

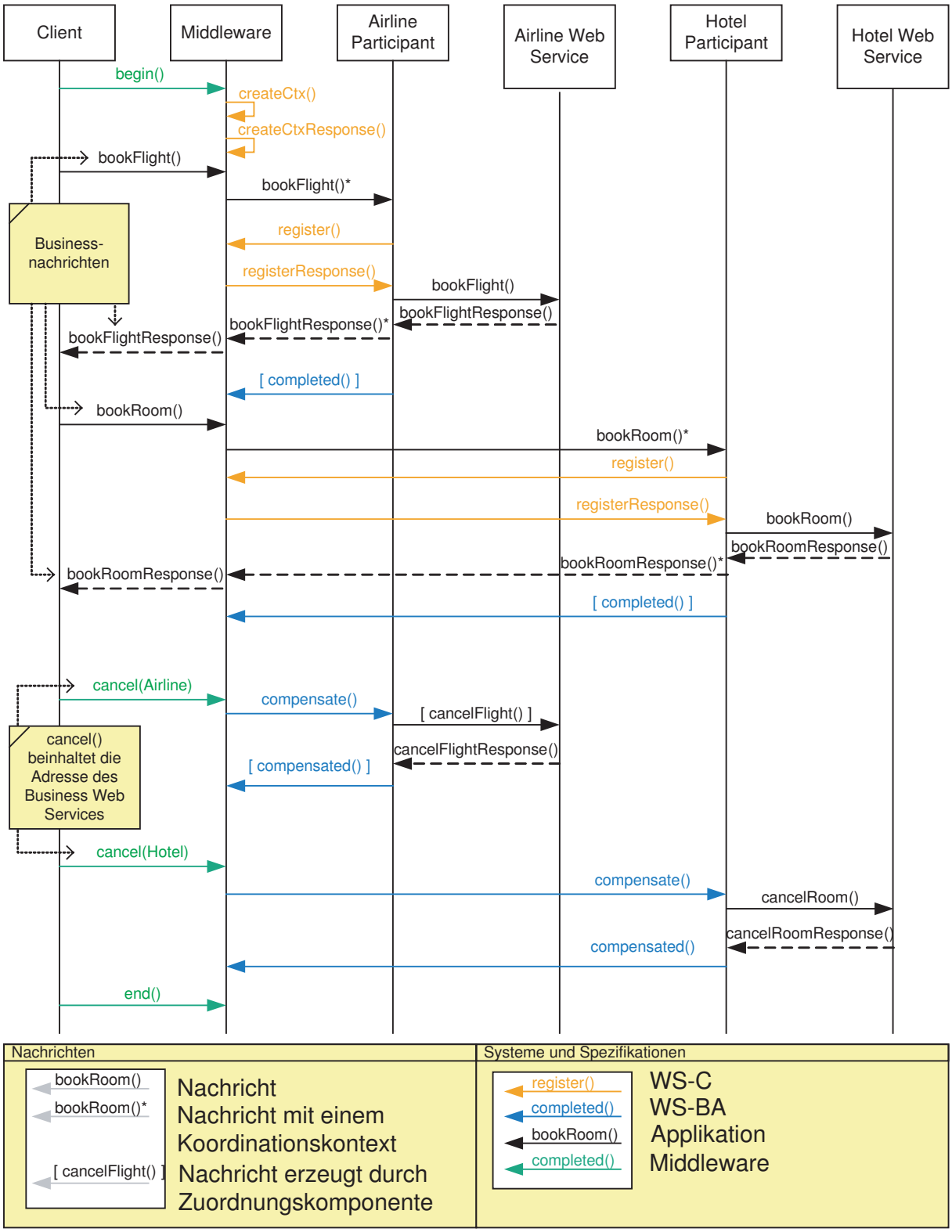

Abbildung 3.1: Kommunikation bei einer Bestellung und einer anschließenden Reklamation

Kapitel 4

Zusammenfassung

4.1 Die Spezifikationen

Im Rahmen der Analysephase dieser Arbeit wurden Entscheidungen getroffen, die die Entkopplung der Koordinations- und Transaktionsdiensten von den applikationsspezifischen Komponenten ermöglichten. Diese Entkopplung ist vor allem für die Adaptierung eines existierenden Webservices auf die in WS-BA Spezifikation beschriebene Kommunikationssemantik von großer Bedeutung. Desweiteren ist die Abspaltung der applikationsunabhängigen Middleware-Komponente für den Einsatz in reallen Bedingungen sinnvoll, weil sie es einem Dienstleistungsanbieter ermöglicht, den Koordinationsdienst als fertiges Produkt zu Verfügung zu stellen. Im Rahmen dieser Arbeit wurde gezeigt, dass die WS-C und WS-BA Spezifikationen einen erweiterbaren Grundgerüst für die Implementierung bieten.

4.2 Die Implementierung

Die Implementierungsphase besaß adäquate Schwierigkeit. Insbesonders wiesen Produkte wie Apache AXIS sowie JBoss Application Server hohe Stabilität auf. Apache AXIS überzeugte mit seiner Erweiterbarkeit als Framework, bot jedoch wenig fertige Lösungen für generischen Problemstellungen.

4.3 Stellung im Gesamtkonzept

In dieser Arbeit wurden die Spezifikationen WS-Coordination und WS-BusinessActivity behandelt. Die zwei Spezifikation bilden nur einen Teil des gesamten WS-* Spezifikationsgerüstes, das zusammen mit der WS-AtomicTransactions Spezifikation transaktionelle Aspekte eines verteilten Systems beschreibt. Die Besonderheit der WS-* Protokollfamilie liegt in dem gleichzeitigen Einsatz beliebig ausgewählter Protokolle. Zur Zeit der Durchführung dieser Arbeit stand die Implementierung der Messaging-Protokolle (siehe Abbildung 1.1) nicht zur Verfügung. Diese Tatsache erschwerte die Implementierung

erheblich.

4.4 Ausblick

Die Beispielimplementierung wurde für die weitere Entwicklung als Projekt im SourceForge (http://sf.net/projects/jwst) angemeldet und steht öffentlich zur Verfügung. Von besonderem Interesse wäre die Erweiterung der Implementierung um die Komponenten, die einen implementierungsunabhängigen Transportmechanismus gewähren. Die aktuelle Implementierung basiert auf dem SOAP-over-HTTP Transportprotokoll, es wäre aber denkbar mit Hilfe von WS-Addressing[5] zum Beispiel auf das SOAP-over-SMTP oder mit Hilfe WS-Addressing und WS-ReliableMessaging[4] auf SOAP-over-JXTA aufzusetzen. Leider existierten zur Zeit der Durchführung dieser Arbeit noch keine Implementierungen dieser Spezifikationen.

Anhang A

Listings

A.1 Proxy Dienst WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
                targetNamespace="http://simon.zambrovski.org/2004/
                    ProxyService"
                xmlns:tns="http://simon.zambrovski.org/2004/
                    ProxyService"
                xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/
                    "
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">

        <wsdl:types>
                <schema
                        targetNamespace="http://simon.zambrovski.org
                            /2004/ProxyService"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                >
                 <xsd:element name="anyBody" type="xsd:anyType"/>
                </schema>
        </wsdl:types>

        <wsdl:message name="proxyServiceRequest">
      <wsdl:part element="tns:anyBody" />
   </wsdl:message>
   <wsdl:message name="proxyServiceResponse">
      <wsdl:part element="tns:anyBody" />
   </wsdl:message>

        <wsdl:portType name="ProxyService">
      <wsdl:operation name="proxyService">
         <wsdl:input message="tns:proxyServiceRequest" name="
             proxyServiceRequest"/>
```

```
        <wsdl:output message="tns:proxyServiceResponse" name="
            proxyServiceResponse"/>
      </wsdl:operation>
   </wsdl:portType>

        <wsdl:binding name="ProxyServiceSoapBinding" type="
            tns:ProxyServicePortType">
      <wsdlsoap:binding style="document" transport="http://schemas.
          xmlsoap.org/soap/http"/>
      <wsdl:operation name="proxyService">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="proxyServiceRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="proxyServiceResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

        <wsdl:service name="ProxyService">
      <wsdl:port binding="tns:ProxyServiceSoapBinding" name="
          ProxyServicePort">
         <wsdlsoap:address location="http://localhost:8080/
             ProxyService"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

Listing A.1: Proxy Dienst WSDL

A.2 Transaction Dienst WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
        WSDL descriptor of transaction web service

        Author:          Simon Zambrovski, simon@zambrovski.org
        Version:         $Id: transaction.wsdl,v 1.1 2004/06/18 08
            :58:54 sza Exp $
-->

<definitions
        targetNamespace="http://simon.zambrovski.org/2003/transaction
            /webservice"
        xmlns:tns="http://simon.zambrovski.org/2003/transaction/
            webservice"
```

```
        xmlns:types="http://simon.zambrovski.org/2003/transaction/
            webservice"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
        >
        <wsdl:import
                namespace="http://schemas.xmlsoap.org/ws/2003/03/
                    addressing"
                location="./wsa.xsd"
        />
        <wsdl:import
                namespace="http://simon.zambrovski.org/2003/
                    transaction/webservice"
                location="./transaction.xsd"
        />

<!-- elements -->
<!-- messages -->
        <message name="beginReq">
                <part name="parameters" element="types:begin" />
        </message>
        <message name="beginResp" >
                <part name="parameters" element="types:beginResponse"
                     />
        </message>
        <message name="endReq">
                <part name="parameters" element="types:end" />
        </message>
        <message name="endResp" >
                <part name="parameters" element="types:endResponse" /
                    >
        </message>
        <message name="confirmReq">
                <part name="parameters" element="types:confirm" />
        </message>
        <message name="confirmResp" >
                <part name="parameters" element="
                    types:confirmResponse" />
        </message>
        <message name="cancelReq">
                <part name="parameters" element="types:cancel" />
        </message>
        <message name="cancelResp" >
                <part name="parameters" element="types:cancelResponse
                    " />
        </message>
```

```
<!-- port definition -->
        <portType name = "TransactionPortType">
                <operation name="begin">
                        <input message="tns:beginReq" />
                        <output message="tns:beginResp" />
                </operation>
                <operation name="end">
                        <input message="tns:endReq" />
                        <output message="tns:endResp" />
                </operation>
                <operation name="confirm">
                        <input message="tns:confirmReq" />
                        <output message="tns:confirmResp" />
                </operation>
                <operation name="cancel">
                        <input message="tns:cancelReq" />
                        <output message="tns:cancelResp" />
                </operation>
        </portType>

<!-- binding description -->
        <binding name="TransactionSOAPBinding" type="
           tns:TransactionPortType" >
                <soap:binding style="document"
                        transport="http://schemas.xmlsoap.org/soap/
                           http" />

                <operation name="begin">
                        <soap:operation style="document" soapAction="
                           begin" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

                <operation name="end">
                        <soap:operation style="document" soapAction="
                           end" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

                <operation name="confirm">
```

```
                        <soap:operation style="document" soapAction="
                            confirm" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

                <operation name="cancel">
                        <soap:operation style="document" soapAction="
                            cancel" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

        </binding>

<!-- service -->
        <service name="TransactionService">
                <port name="TransactionService" binding="
                    tns:TransactionSOAPBinding" >
                        <soap:address location="http://localhost:8080
                            /axis/services/TransactionService" />
                </port>
        </service>
</definitions>
```

Listing A.2: Transaction Dienst WSDL

A.3 Transaction Dienst XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
        Schema of transaction web service

        Author:         Simon Zambrovski, simon@zambrovski.org
        Version:        $Id: transaction.xsd,v 1.1 2004/06/18 08
            :58:54 sza Exp $
-->

<xsd:schema
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
        xmlns:tns="http://simon.zambrovski.org/2003/transaction/
            webservice"
        targetNamespace="http://simon.zambrovski.org/2003/transaction
            /webservice"
>
        <xsd:complexType name="OperationType">
                <xsd:sequence>
                        <xsd:element name="EndpointReference"
                                type="wsa:EndpointReferenceType"
                                    minOccurs="1" maxOccurs="1" />
                        <xsd:any namespace="##other"
                                processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="CancelType">
                <xsd:complexContent>
                        <xsd:extension base="tns:OperationType" />
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="ConfirmType">
                <xsd:complexContent>
                        <xsd:extension base="tns:OperationType" />
                </xsd:complexContent>
        </xsd:complexType>

        <xsd:element name="begin"
                            type="xsd:string" />
        <xsd:element name="beginResponse"                   type="
            xsd:boolean" />
        <xsd:element name="end"
                            type="xsd:string" />
        <xsd:element name="endResponse"                     type="
            xsd:boolean" />

        <xsd:element name="confirm"
                    type="tns:ConfirmType" />
        <xsd:element name="confirmResponse"       type="xsd:boolean" />

        <xsd:element name="cancel"
                    type="tns:CancelType" />
        <xsd:element name="cancelResponse"        type="xsd:boolean" />

</xsd:schema>
```

Listing A.3: Transaction Dienst XML-Schema

A.4 Airline Dienst WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
        WSDL descriptor of airline web service in WS-T sample
            application

        Author:         Simon Zambrovski, simon@zambrovski.org
        Version:        $Id: airline.wsdl,v 1.1 2004/06/18 08:58:54
            sza Exp $
-->

<definitions
        targetNamespace="http://simon.zambrovski.org/2003/airline/
            webservice"
        xmlns:tns="http://simon.zambrovski.org/2003/airline/
            webservice"
        xmlns:types="http://simon.zambrovski.org/2003/airline"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        >

        <wsdl:import namespace="http://simon.zambrovski.org/2003/
            airline"
                location="./airline.xsd"
        />

<!-- messages -->


        <!-- getFlights -->
        <message name="getFlightsReq">
                <part name="parameters" element="types:GetFlights" />
        </message>
        <message name="getFlightsResp" >
                <part name="parameters" element="
                    types:GetFlightsResponse" />
        </message>

        <!-- bookFlight -->
        <message name="bookFlightReq">
                <part name="parameters" element="types:BookFlight" />
        </message>
        <message name="bookFlightResp" >
                <part name="parameters" element="
                    types:BookFlightResponse" />
        </message>
```

```
        <!-- cancelFlightReservation -->
        <message name="cancelFlightReservationReq">
                <part name="parameters" element="
                   types:CancelFlightReservation" />
        </message>
        <message name="cancelFlightReservationResp" >
                <part name="parameters" element="
                   types:CancelFlightReservationResponse" />
        </message>


<!-- port definition -->

        <portType name = "AirlinePortType">
                <operation name="getFlights">
                        <input message="tns:getFlightsReq" />
                        <output message="tns:getFlightsResp" />
                </operation>

                <operation name="bookFlight" >
                        <input message="tns:bookFlightReq" />
                        <output message="tns:bookFlightResp" />
                </operation>

                <operation name="cancelFlightReservation" >
                        <input message="
                           tns:cancelFlightReservationReq" />
                        <output message="
                           tns:cancelFlightReservationResp" />
                </operation>
        </portType>



<!-- binding description -->

        <binding name="AirlineSOAPBinding" type="tns:AirlinePortType"
            >
                <soap:binding style="document"
                        transport="http://schemas.xmlsoap.org/soap/
                           http" />

                <operation name="getFlights">
                        <soap:operation style="document" soapAction="
                           getFlights" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
```

```
                </output>
            </operation>

            <operation name="bookFlight">
                <soap:operation style="document" soapAction="
                    bookFlight" />
                <input>
                    <soap:body use="literal" />
                </input>
                <output>
                    <soap:body use="literal" />
                </output>
            </operation>

            <operation name="cancelFlightReservation">
                <soap:operation style="document" soapAction="
                    cancelFlightReservation" />
                <input>
                    <soap:body use="literal" />
                </input>
                <output>
                    <soap:body use="literal" />
                </output>
            </operation>

        </binding>


<!-- service -->

        <service name="AirlineService">
            <port name="AirlineSOAPPort" binding="
                tns:AirlineSOAPBinding" >
                <soap:address location="http://localhost:8080
                    /axis/services/AirlineService" />
            </port>
        </service>

</definitions>

<!--
        $Log: airline.wsdl,v $
        Revision 1.1  2004/06/18 08:58:54  sza
        init

        Revision 1.2  2004/01/27 19:29:00  sza
        flight array

        Revision 1.1  2004/01/26 12:01:31  sza
        init
```

```
-->
```

Listing A.4: Airline Dienst WSDL

A.5 Airline Dienst XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DOCTYPE schema PUBLIC "-//W3C/DTD XML Schema Version 1.0//EN"
                          "http://www.w3.org/TR/2000/WD-xmlschema
                              -1-20000225/structures.dtd"-->
<!--
        Airline Schema
        Schema definition for the WS-I in JXTA travel agency sample

        Author:         Simon Zambrovski, simon@zambrovski.org
        Version:        $Id: airline.xsd,v 1.1 2004/06/18 08:58:54
           sza Exp $
-->
<xsd:schema
        targetNamespace="http://simon.zambrovski.org/2003/airline"
        xmlns:tns="http://simon.zambrovski.org/2003/airline"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:SOAP-ENC  ="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
        elementFormDefault="qualified">

<!-- elements -->

        <xsd:element name="TravelAgencyName" type="xsd:string" />

        <xsd:element name="AirlineName" type="xsd:string" />

        <xsd:element name="CityNameStart" type="xsd:string" />
        <xsd:element name="CityNameFinish" type="xsd:string" />

        <xsd:element name="SuccessStatus" type="xsd:boolean" />

        <xsd:element name="FlightDate" type="xsd:dateTime" />
        <xsd:element name="FlightTime" type="xsd:dateTime" />
        <xsd:element name="FlightNumber" type="xsd:int" />
        <xsd:element name="SeatNumber" type="xsd:int" />
        <xsd:element name="NumberOfFreeSeats" type="xsd:int" />

        <xsd:element name="Flight">
                <xsd:complexType>
                        <xsd:sequence>
```

```
                                <xsd:element ref="tns:CityNameStart"
                                    />
                                <xsd:element ref="tns:CityNameFinish"
                                     />
                                <xsd:element ref="tns:AirlineName" />
                                <xsd:element ref="tns:FlightNumber" /
                                    >
                                <xsd:element ref="tns:FlightTime" />
                                <xsd:element ref="
                                    tns:NumberOfFreeSeats" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="FlightArray">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element minOccurs="0" maxOccurs=
                                    "unbounded" ref="tns:Flight" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="FlightReservation">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:CityNameStart"
                                    />
                                <xsd:element ref="tns:CityNameFinish"
                                     />
                                <xsd:element ref="tns:AirlineName" />
                                <xsd:element ref="
                                    tns:TravelAgencyName" />
                                <xsd:element ref="tns:FlightNumber" /
                                    >
                                <xsd:element ref="tns:FlightTime" />
                                <xsd:element ref="tns:SeatNumber" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>


<!-- operations -->

        <xsd:element name="GetFlights">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:CityNameStart"
                                    />
```

```
                        <xsd:element ref="tns:CityNameFinish"
                            />
                        <xsd:element ref="tns:FlightDate" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="GetFlightsResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="tns:FlightArray"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="BookFlight">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="tns:FlightNumber" /
                            >
                        <xsd:element ref="
                            tns:TravelAgencyName" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="BookFlightResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="
                            tns:FlightReservation" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="CancelFlightReservation">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="
                            tns:FlightReservation" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="CancelFlightReservationResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element ref="tns:SuccessStatus"
                            />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
```

```
</xsd:schema>
<!--
        $Log: airline.xsd,v $
        Revision 1.1  2004/06/18 08:58:54  sza
        init

        Revision 1.2  2004/01/27 19:29:00  sza
        flight array

        Revision 1.1  2004/01/26 12:01:31  sza
        init

-->
```

Listing A.5: Airline Dienst XML-Schema

A.6 Hotel Dienst WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
        WSDL descriptor of hotel web service in WS-T sample
            application

        Author:         Simon Zambrovski, simon@zambrovski.org
        Version:        $Id: hotel.wsdl,v 1.1 2004/06/18 08:58:54 sza
            Exp $
-->

<definitions
        targetNamespace="http://simon.zambrovski.org/2003/hotel/
            webservice"
        xmlns:tns="http://simon.zambrovski.org/2003/hotel/webservice"
        xmlns:types="http://simon.zambrovski.org/2003/hotel"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        >

        <wsdl:import namespace="http://simon.zambrovski.org/2003/
            hotel"
                location="./hotel.xsd"
        />

<!-- messages -->


        <!-- getName -->
```

```
        <message name="getHotelNameReq">
                <part name="parameters" element="types:GetHotelName"
                    />
        </message>
        <message name="getHotelNameResp" >
                <part name="parameters" element="
                    types:GetHotelNameResponse" />
        </message>

        <!-- getAvailiableRooms -->
        <message name="getAvailiableRoomsReq">
                <part name="parameters" element="
                    types:GetAvailiableRooms" />
        </message>
        <message name="getAvailiableRoomsResp" >
                <part name="parameters" element="
                    types:GetAvailiableRoomsResponse" />
        </message>

        <!-- bookRoom -->
        <message name="bookRoomReq">
                <part name="parameters" element="types:BookRoom" />
        </message>
        <message name="bookRoomResp" >
                <part name="parameters" element="
                    types:BookRoomResponse" />
        </message>

        <!-- cancelReservation -->
        <message name="cancelReservationReq">
                <part name="parameters" element="
                    types:CancelReservation" />
        </message>
        <message name="cancelReservationResp" >
                <part name="parameters" element="
                    types:CancelReservationResponse" />
        </message>


<!-- port definition -->

        <portType name = "HotelPortType">
                <operation name="getHotelName">
                        <input message="tns:getHotelNameReq" />
                        <output message="tns:getHotelNameResp" />
                </operation>

                <operation name="getAvailiableRooms" >
                        <input message="tns:getAvailiableRoomsReq" />
```

```
                <output message="tns:getAvailiableRoomsResp"
                    />
            </operation>

            <operation name="bookRoom" >
                <input message="tns:bookRoomReq" />
                <output message="tns:bookRoomResp" />
            </operation>

            <operation name="cancelReservation" >
                <input message="tns:cancelReservationReq" />
                <output message="tns:cancelReservationResp" /
                    >
            </operation>
        </portType>



<!-- binding description -->

        <binding name="HotelSOAPBinding" type="tns:HotelPortType" >
            <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/
                    http" />

            <operation name="getHotelName">
                <soap:operation style="document" soapAction="
                    getHotelName" />
                <input>
                    <soap:body use="literal" />
                </input>
                <output>
                    <soap:body use="literal" />
                </output>
            </operation>

            <operation name="getAvailiableRooms">
                <soap:operation style="document" soapAction="
                    getAvailiableRooms" />
                <input>
                    <soap:body use="literal" />
                </input>
                <output>
                    <soap:body use="literal" />
                </output>
            </operation>

            <operation name="bookRoom">
                <soap:operation style="document" soapAction="
                    bookRoom" />
```

```
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

                <operation name="cancelReservation">
                        <soap:operation style="document" soapAction="
                           cancelReservation" />
                        <input>
                                <soap:body use="literal" />
                        </input>
                        <output>
                                <soap:body use="literal" />
                        </output>
                </operation>

        </binding>


<!-- service -->

        <service name="HotelService">
                <port name="HotelSOAPPort" binding="
                   tns:HotelSOAPBinding" >
                        <soap:address location="http://localhost:8080
                           /axis/services/HotelService" />
                </port>
        </service>

</definitions>

<!--
        $Log: hotel.wsdl,v $
        Revision 1.1  2004/06/18 08:58:54  sza
        init

-->
```

Listing A.6: Hotel Dienst WSDL

A.7 Hotel Dienst XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DOCTYPE schema PUBLIC "-//W3C/DTD XML Schema Version 1.0//EN"
```

```
                        "http://www.w3.org/TR/2000/WD-xmlschema
                            -1-20000225/structures.dtd"-->
<!--
        Hotel Schema
        Schema definition for the WS-T in JXTA travel agency sample

        Author:         Simon Zambrovski, simon@zambrovski.org
        Version:        $Id: hotel.xsd,v 1.1 2004/06/18 08:58:54 sza
           Exp $
-->
<xsd:schema
        targetNamespace="http://simon.zambrovski.org/2003/hotel"
        xmlns:tns="http://simon.zambrovski.org/2003/hotel"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified">

<!-- elements -->

        <xsd:element name="TravelAgencyName" type="xsd:string" />

        <xsd:element name="HotelName" type="xsd:string" />

        <xsd:element name="CityName" type="xsd:string" />

        <xsd:element name="NumberOfRooms" type="xsd:int" />

        <xsd:element name="RoomNumber" type="xsd:int" />

        <xsd:element name="SuccessStatus" type="xsd:boolean" />

        <xsd:element name="TimeSpan">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="begin" type="
                                    xsd:dateTime"/>
                                <xsd:element name="end" type="
                                    xsd:dateTime"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="Reservation">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:TimeSpan" />
                                <xsd:element ref="
                                    tns:TravelAgencyName" />
                                <xsd:element ref="tns:HotelName" />
                                <xsd:element ref="tns:RoomNumber" />
                        </xsd:sequence>
```

```
                </xsd:complexType>
        </xsd:element>


<!-- operations -->

        <xsd:element name="GetHotelName">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:CityName" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="GetHotelNameResponse">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:HotelName" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="GetAvailiableRooms">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:TimeSpan" />
                                <xsd:element ref="tns:HotelName" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="GetAvailiableRoomsResponse">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:NumberOfRooms"
                                    />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="BookRoom">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:TimeSpan" />
                                <xsd:element ref="
                                    tns:TravelAgencyName" />
                                <xsd:element ref="tns:HotelName" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="BookRoomResponse">
                <xsd:complexType>
```

```
                        <xsd:sequence>
                                <xsd:element ref="tns:Reservation" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

        <xsd:element name="CancelReservation">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:Reservation" />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="CancelReservationResponse">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="tns:SuccessStatus"
                                   />
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>

</xsd:schema>
<!--
        $Log: hotel.xsd,v $
        Revision 1.1  2004/06/18 08:58:54  sza
        init

-->
```

Listing A.7: Hotel Dienst XML-Schema

Anhang B

Literaturverzeichnis

[1] Understanding webservices, spezifikations. Technical report, Microsoft.

[2] Introducing ws-transaction. Technical report, Arjuna Technologies Ltd., May 2003.

[3] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2001.

[4] Ruslan Bilorusets, Adam Bosworth, Don Box, Luis Felipe Cabrera, Derek Collison, Donald Ferguson, Christopher Ferris, Tom Freund, Mary Ann Hondo, John Ibbotson, Chris Kaler, David Langworthy, Amelia Lewis, Rodney Limprecht, Steve Lucco, Matt Mihic, Don Mullen, Anthony Nadalin, Mark Nottingham, David Orchard, Shivajee Samdarshi, John Shewchuk, and Tony Storey. Web Services Reliable Messaging (WS-ReliableMessaging). Technical report, IBM, Microsoft, BEA, March 2004.

[5] Adam Bosworth, Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Tony Storey, and Sanjiva Weerawarana. Web Services Addressing (WS-Addressing). Technical report, IBM, Microsoft, BEA, March 2004.

[6] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. Technical report, W3C, May 2000.

[7] Luis Felipe Cabrera, George Copeland, William Cox, Max Feingold, Tom Freund, Jim Johnson, Chris Kaler, Johannes Klein, David Langworthy, Anthony Nadalin, David Orchard, Ian Robinson, John Shewchuk, Tony Storey, and Satish Thatte. Web services coordination (ws-coordination). Technical report, IBM, Microsoft, BEA, September 2003.

[8] Luis Felipe Cabrera, George Copeland, William Cox, Tom Freund, Johannes Klein, David Langworthy, Ian Robinson, Tony Storey, and Satish Thatte. Web services atomic transactions (ws atomic transactions). Technical report, IBM, Microsoft, BEA, Januar 2004.

[9] Luis Felipe Cabrera, George Copeland, William Cox, Tom Freund, Johannes Klein, David Langworthy, Ian Robinson, Tony Storey, and Satish Thatte. Web services business activity framework (ws-businessactivity). Technical report, IBM, Microsoft, BEA, Januar 2004.

[10] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, March 2001.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[12] Kevin Gibbs, Brian D Goodman, and Elias Torres. Create web services using apache axis and castor. Technical report, IBM, September 2003.

[13] J. Gray and A. Reuter. *Transaction Processing*. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1993.

[14] Biplav Srivastava and Jana Koehler. Web service composition - current solutions and open problems. Technical report, IBM.