Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projekt-Ausarbeitung

Martin Gerlach

WS-Coordination und WS-BusinessActivity in Action mit Apache Axis

Betreuender Prüfer: Prof. Dr. Kai von Luck

Fakultät Technik und Informatik
Studiendepartment Informatik

Martin Gerlach

Thema der Projekt-Ausarbeitung

WS-Coordination und WS-BusinessActivity in Action mit Apache Axis

Stichworte

Dienste, Web Services, Verteilte Transaktionen, Langlebige Transaktionen

Kurzzusammenfassung

In diesem Artikel beschreibt der Autor seinen Beitrag zu einem Studienprojekt im Bereich Angewandte Informatik. Auf Basis von Apache Axis wurde vom Autor für eine umfangreiche verteilte, auf dem Paradigma "'Service Oriented Architecture"' (SOA) basierende Anwendung eine Middleware zur Unterstützung der Web-Service-Spezifikationen WS-Coordination und WS-BusinessActivity entworfen und prototypisch realisiert. Das Ziel war die Ermöglichung der Durchführung kontrollierter langlaufender Transaktionen mit Web Services. Basierend auf vorangegangenen Arbeiten des Autors werden die Gesamtarchitektur und der Entwurf der Middleware und der zugehörigen API sowie einige nennenswerte Implementierungsdetails vorgestellt. Abschließend erfolgt eine Betrachtung der dabei gewonnenen Erkenntnisse sowie ein Ausblick auf mögliche weitere Schritte.

Inhaltsverzeichnis

1 Einleitung

In der Veranstaltung „Projekt Angewandte Informatik“ des 3. Semesters des Master-Studiengangs Informatik an der HAW Hamburg haben die Studenten im Wintersemester 2005/2006 praxisorientiert an verschiedenen Themen rund um das Szenario „Ferienclub“ gearbeitet.[1] Eine Gruppe von Studenten hat sich damit befasst, Konzepte für die Gesamt-Architektur der IT-Landschaft des Ferienclubs auszuarbeiten. Aufgrund der Popularität in der Industrie wurde sich dabei auf das Paradigma „Service Oriented Architectures“ (SOA) mit Web Services konzentriert (Siehe hierzu z.B. (Weerawarana u. a., 2005), (Zimmermann u. a., 2003/2005)). Im Projekt hat sich die Architekturgruppe zum Ziel gesetzt, in relativ kurzer Zeit (laut Studienplan ca. 100 Stunden) mit einfachen Mitteln folgende Komponenten zu entwerfen und prototypisch zu implementieren, die bereits im 2. und 3. Semester im Rahmen von Seminaren genauer untersucht wurden:

- Einen zentralen Enterprise Service Bus (ESB) mit grundlegenden Funktionalitäten für den Ferienclub (Sven Stegelmeier, siehe (Stegelmeier, 2005) und die darin enthaltenen Literaturhinweise).
- Eine Sicherheits-Middleware für Web Services gemäß WS-Security (Thies Rubarth, siehe (Rubarth, 2005b) und (Rubarth, 2005a)).
- Eine Transaktions-Middleware zur Koordinierung von asynchronen Web-Service-Aufrufen in langlebigen Transaktionen (der Autor, siehe (Gerlach, 2005b) und (Gerlach, 2005a)).

Aufgrund der kurzen zur Verfügung stehenden Zeit wurde entschieden, die Prototypen für die Sicherheits- und Transaktions-Schichten auf einem frei verfügbaren Web-Service-Container aufzubauen. Die Wahl fiel auf Apache Axis, siehe 2.1. Als Application Server wurde JBoss in Version 4.0.2 gewählt (JBoss, 2005), da eine gute Integration in die Java-Entwicklungsumgebung Eclipse ((JBoss-Eclipse, 2005), (Eclipse, 2005)) möglich ist. Als Enterprise Service Bus wurde Mule (Mule, 2005) gewählt, da dieser Open Source ist und außerdem Axis enthält.

In dieser Arbeit stellt der Autor seinen Beitrag zum Projekt, die Middleware für langlebige Transaktionen, vor. In Abschnitt 2 werden kurz der Apache Axis Web-Service-Container sowie die zu implementierenden Spezifikationen vorgestellt. Abschnitt 3 enthält Hinweise auf vorangegangene Arbeiten des Autors, auf die im Folgenden aufgebaut wird. Das Design der Middleware wird in Abschnitt 4 beschrieben, bevor in Abschnitt 5 kurz auf Implementierungsdetails eingegangen wird. Fazit und Ausblick finden sich in Abschnit 6.

[1] Siehe hierzu auch die Vortragsreihen der Seminare des 2. und 3. Semesters unter http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/vortraege.html sowie http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master05-06/vortraege.html

2 Hintergrund

2.1 Apache Axis

Apache Axis (Avar u. a., 2005) ist ein Open Source Web-Service-Container, welcher das Simple Object Access Protocol SOAP (Gudgin u. a., 2003) implementiert. Axis verarbeitet XML-Nachrichten gemäß der SOAP-Spezifikation und leitet die enthaltenen Informationen an registrierte Web Services weiter, wie in Abb. 1 angedeutet. Die SOAP-Nachrichten wer-

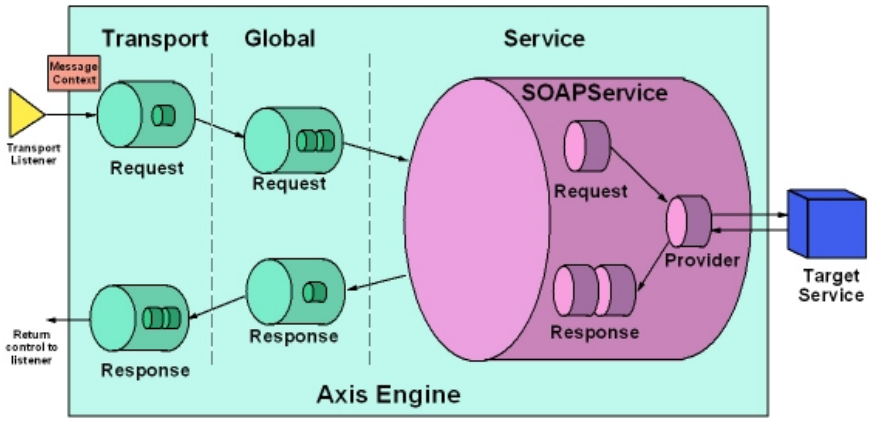

Abbildung 1: Axis Server Message Flow (Avar u. a., 2005)

den dabei so interpretiert, wie es für den aufgerufenen Service (dieser wird anhand der Aufruf-URL indentifiziert) in desses Web-Service-Beschreibung (WSDL-Dokument) angegeben wurde. Für den Prototypen der Transaktions-Middleware wird als Nachrichtentyp „Document/Literal“ vorausgesetzt. Das Root-Element des SOAP-Bodies der Nachricht bestimmt damit die Operation, die auf dem Service aufgerufen wird. Die Operation hat genau einen Parameter, nämlich das Root-Element selbst in deserialisierter Form.

Abb. 2 zeigt die Subsysteme von Axis. Wie für Web Services gefordert, werden Nachrichten-Encoding, Nachrichten-Transport sowie Service-Implementierungen (Service-Provider) getrennt und unabhängig voneinander gehandhabt. Axis beherrscht neben SOAP auch XML-RPC als Aufruf-Standard.

Axis enthält weiterhin Tools, die aus der Web-Service-Beschreibung und Angaben über die Art des Service-Providers (z.B. einfache Java-Klasse, EJB, ...) entsprechende Java-Klassen für folgende Teilkomponenten eines Services generieren:

- Service-Interface: Enthält die Operationen des Services.
- Service-Skeleton- und Service-Implementierungsklassen: Zur Verarbeitung empfangener Nachrichten. Die Service-Implementierungsklasse wird dabei mit leeren Methodenrümpfen generiert, in die der Entwickler dann die Geschäftslogik einfügt.

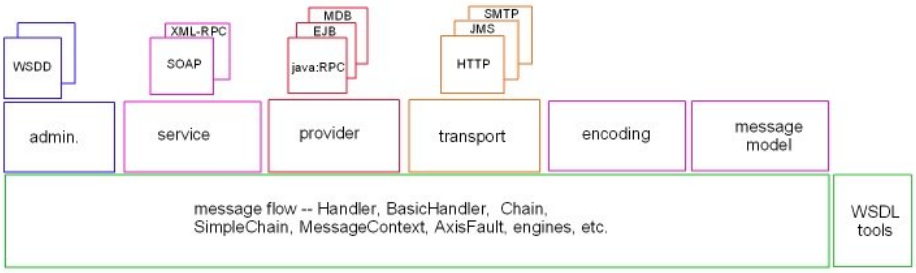

Abbildung 2: Axis Subsysteme (Avar u. a., 2005)

- Service-Locator-Klasse: Hilfsklasse, die eine Service-Stub-Instanz erzeugt, um darüber einen Service aufzurufen. (Zum Service-Locator-Entwurfsmuster siehe (Alur u. a., 2003)).
- Service-Stub-Klasse: Zum Aufrufen eines Web Services.
- Datentypenklassen gemäß `<types>`-Abschnitt des WSDL-Dokuments: Diese Klassen werden von Axis zum Serialisieren und Deserialisieren der XML-Nachrichten verwendert.

2.2 WS-Coordination und WS-BusinessActivity

WS-Coordination (Carbrera u. a., 2005c) ist eine Spezifikation über die Koordinierung von verteilten Web Services, die zusammen an der Erfüllung einer bestimmten Aufgabe arbeiten. Ähnlich wie bei klassischen verteilten Transaktionen stößt ein zentraler Web Service („Coordinator“) die verteilte Verarbeitung an und alle teilnehmenden Web Services („Participants“) registrieren sich nach ihrem Aufruf bei diesem zentralen Service innerhalb eines gemeinsamen Transaktionskontextes („Coordination Context“). WS-Coordination legt Syntax und Semantik der ausgetauschten Nachrichten fest. Es wird lediglich das erzeugen des Kontextes und die Registrierung spezifiert. Das eigentliche Terminierungsprotokoll ist beliebig. Sämtlicher Nachrichtenaustausch hat asynchron stattzufinden.

WS-BusinessActivity (Carbrera u. a., 2005b) spezifiziert zwei Terminierungsprotokolle, welche im Anschluss an WS-Coordination verwendet werden können, um die an der verteilten Transaktion beteiligten Web Services zu kontrollieren und einen konsistenten Ausgang der Transaktion sicher zu stellen. WS-BusinessActivity beschreibt dabei langlaufende, nichtatomare Aktivitäten, die aus einzelnen, von den aufgerufenen Web Services ausgeführten, Aktionen bestehen. Entscheidend ist dabei, dass jeder einzelne Teilnehmer in der Lage sein muss, eine Aktion jederzeit abzubrechen. Sollte die Aktion bereits ausgeführt worden sein, so muss der Teilnehmer solange dazu in der Lage sein, die Aktion zu kompensieren (z.B.

Storno einer Reservierung, siehe auch (Gerlach, 2005b, Abschnitt 4.2)), bis die gesamte Transaktion abgeschlossen wurde. Es werden wie bei WS-Coordination Syntax und Semantik der ausgetauschten Nachrichten festgelegt. Auch hier ist der Nachrichtenaustausch asynchron.

Details zu den Nachrichten sowie zum Ablauf des Nachrichtenaustauschs (Interaktion zwischen Coordinator und Participants) finden sich in den Spezifikationen. Einen Überblick geben (Gerlach, 2005a, Abschnitt 4.2) und (Gerlach, 2005b, Abschnitt 4.4.3).

3 Vorarbeiten

Der Autor hat im Rahmen der Ausarbeitung zum Seminar (SR) des Master-Studiengangs bereits diverse Anforderungen an eine Middleware für langlaufende Transaktionen erarbeitet. Weiterhin wurde die grobe Architektur der Middleware wie in Abb. 3 gezeigt skizziert. In den

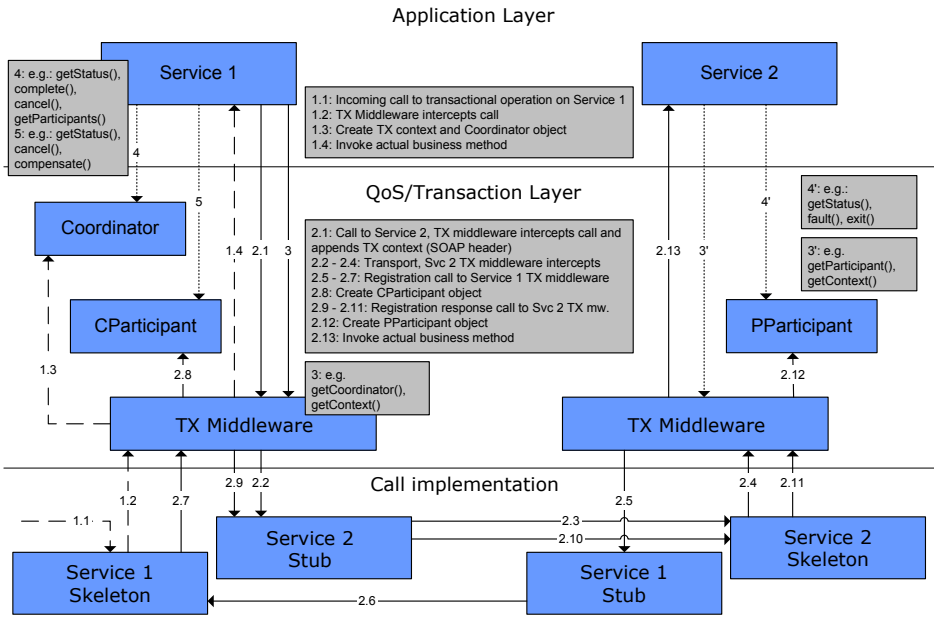

Abbildung 3: Architektur mit beispielhaften Abläufen (Gerlach, 2005a)

folgenden Abschnitten geht es nun darum, wie Apache Axis erweitert werden muss und wie Web Services für Apache Axis entworfen und implementiert werden müssen, um den WS-Coordination- und WS-BusinessActivity-Spezifikationen zu genügen. Hauptanforderungen und -ziele sind dabei gemäß (Gerlach, 2005a, Abschnitt 4.1):

- Verwendung von ausschließlich asynchronen Aufrufen unter Beachtung der Tatsache, dass die Geschäftslogik u.U. ihren Zustand über mehrer Aufrufe und Callbacks hinweg bewahren muss.
- Automatische Abwicklung der Koordinierung (Erzeugen des Kontexts, Registrierung) sowie automatische Erzeugung und Erkennung der benötigten SOAP Header (z.B. für den Transaktionskontext).
- Entwicklung einer einfachen API, das eine Interaktion der Geschäftslogik mit der Transaktions-Middleware erlaubt, denn dies ist aufgrund der Offenheit der WS-BusinessAcitvity-Protokolle je nach Anwendungsfall mehr oder weniger nötig. Es sind hier auch verschiedene Transaktionsmuster vorstellbar, ähnlich wie bei EJBs (DeMichiel, 2003).
- Möglichst große Transparenz für den Entwickler der eigentlichen Geschäftslogik sowie Generierung des Codes der Transaktions-Middleware, der spezifisch für jeden einzelnen Web Service ist.

4 Entwurf

4.1 Integration der Transaktionslogik in Axis

Die Transaktions-Middleware muss gemäß Abb. 3 eingehende und abgehende Web-Service-Aufrufe abfangen und notwendige Schritte der Transaktionslogik durchführen bevor die aufgerufene Business-Operation ausgeführt wird. Die Transaktionslogik ist insofern nah an der Geschäftslogik, als dass auch sie unabhängig von Transport, Encoding und Nachrichtenformat arbeitet. Weiterhin ist die Transaktionslogik mit der Geschäftslogik verzahnt, da die Geschäftslogik über die Transaktions-API selber in Transaktionen eingreifen kann bzw. können muss.

Nachrichten werden in Apache Axis von sogenannten Handler-Chains abgearbeitet, wobei ein Handler für eine bestimmte Aufgabe zuständig ist. Dabei wird das Aufrufen der Business-Operation ebenfalls von einem Handler durchgeführt. Die Transaktionslogik könnte also als Axis-Handler implementiert werden, der in der Handler-Chain vor dem Aufruf der Business-Methode abläuft. Axis-Handler sind jedoch vollständig unabhängig voneinander, es gibt keine Möglichkeit Daten von einem Handler an den nächsten zu übergeben, außer die Nachricht selbst zu verändern. Es müssen jedoch zumindest der Transaktionskontext sowie die Adresse des aufrufenden Web Services aus den SOAP-Headern extrahiert werden und der Transaktions-Middleware für die Dauer der folgenden Business-Operation zur Verfügung gestellt werden.

Für den Prototypen wurde daher entschieden, den Axis-Aufruf-Handler selbst so zu verändern, dass dem Skeleton die benötigten Header-Informationen vor dem Aufruf der

Business-Operation zur Verfügung gestellt werden. Dazu müssen die Skeletons eine neue Instanzvariable erhalten, die die Header-Informationen für jeden Aufruf enthält. Auf diese Weise muss für jeden Aufruf eine neue Skeleton-Instanz erzeugt werden, d.h. die Service-Operationen müssen Request-Scope haben.

In Axis delegiert das Skeleton alle Methodenaufrufe an eine Instanz der Service-Implementierungsklasse. Das heißt, auch diese Instanz muss die Header-Informationen kennen. In Abb. 4 ist dies unten durch die `Axis_Coordinator...`-Klassen für einen Coordinator-Service angedeutet. Für einen Participant-Service gilt dies analog, lediglich enthalt das Service-Interface (`ParticipantService` anstelle von `CoordinatorService`) dann die WS-Coordination- und WS-BusinessActivity-Operationen für Participants (also „registerResponse“, „complete“, „close“, „exited“, usw.).

Jeder Web Service, der an Transaktionen in Coordinator- oder Participant-Rolle teilnimmt, muss die von den Spezifikationen WS-Coordination und WS-BusinessActivity definierten Operationen zur Verfügung stellen. Diese müssen in der Web-Service-Beschreibung (WSDL) entsprechend enthalten sein. Der Axis-Code-Generator erzeugt daher in der Service-Implementierungsklasse auch Methodenrümpfe für die WS-Coordination- und WS-BusinessActivity-Operationen.

In der Service-Implementierungsklasse muss nun die Transaktionslogik implementiert werden. Die Business-Methoden der Service-Implementierungsklasse enthalten Transaktionslogik, die je nach Rolle des Web Services (Coordinator oder Participant) und Art der Methode (Coordinator-Methode, die eine neue Transaktion startet, Callback, Participant-Methode) benötigte Aktionen durchführt, bevor die Ausführung an die eigentliche Geschäftslogik delegiert wird. Die Methoden, die durch die WS-Coordination und WS-BusinessActivity-Operationen aufgerufen werden, arbeiten die Koordinierungs- und Terminierungsprotokolle ab. Die Spezifikationen definieren dabei genau, welche Operationen in allen möglichen Zuständen welche Auswirkungen (Aktionen, Folgezustände) für Coordinator und Participant haben.

Es wird also ein zusätzlicher Delegationsschritt eingeführt. Die eigentliche Geschäftslogik wird nun nicht mehr in der Service-Implmentierungsklasse implementiert, sondern in einer neuen Operations-Implementierungsklasse (`...ServiceOperationsImpl` in Abb. 4). Von dieser Klasse wird pro Transaktion von der Transaktionslogik (z.B. beim ersten Aufruf einer Methode innerhalb einer Transaktion) eine Instanz angelegt, die über alle Aufrufe innerhalb derselben Transaktion erhalten bleibt. Die Geschäftslogik kann in Instanzvariablen dieser Klasse also Zustand speichern. Da die Methoden asynchron aufgerufen werden, muss die Klasse reentrant implementiert werden. Die Operations-Implementierungsklasse enthält neben den Business-Methoden auch noch Listener-Methoden, die von der Transaktionslogik aufgerufen werden, nachdem bestimmte Transaktions-Nachrichten empfangen wurden.

Sollen aus einem transaktionalen Web Service andere transaktionale Web Services aufgerufen werden, so muss der Transaktionskontext, sowie weitere benötigte Informationen gemäß der Spezifikationen, automatisch der zu erzeugenden SOAP-Nachricht in Form von

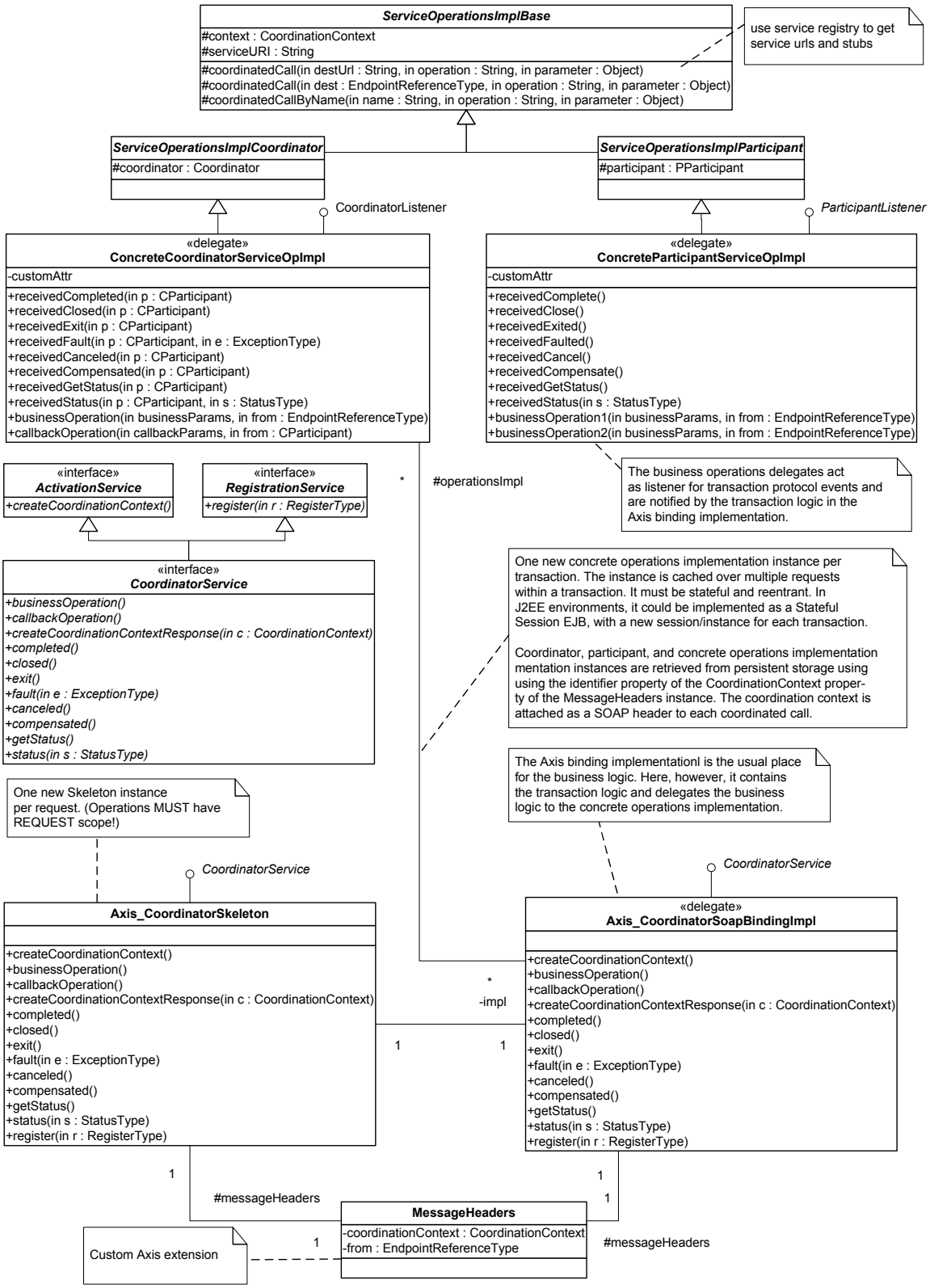

Abbildung 4: API für die Implementierungsklassen und Zusammenspiel mit Apache Axis

SOAP-Headern hinzugefügt werden. Axis arbeitet beim Aufruf anderer Web Services mit generierten Stubs, auf die über eine Service-Locator-Instanz zugegriffen werden kann. Mittels einer Methode des Stubs können SOAP-Header hinzugefügt werden, bevor die eigentliche Business-Operation über eine Methode des Stubs aufgerufen wird, die dann die notwendige Kommunikation veranlasst.

Den koordinierten Aufruf weiterer Web Services transparent zu gestalten ist schwierig, da die Axis-Stubs für jeden Aufruf neu erzeugt werden und keine Informationen über den aktuellen Transaktionskontext haben. Für den Prototypen wurde daher eine simple „Coordinated Call"-API vorgesehen, die sich einer Service-Registry bedient, die Web Services als 3-Tupel mit Service-Namen, Service-URL und sowie Service-Locator-Instanz speichert und über den Namen und die URL finden kann. Die `coordinatedCall`-Methoden der Klasse `ServiceOperationsImplBase` (Abb. 4 oben) fügen den aktuellen Kontext sowie die Absender-URL der SOAP-Nachricht hinzu, bevor die eigentliche Operation aufgerufen wird.

4.2 Coordinator und Participant API

Über Instanzen der Klassen `Coordinator`, `CParticipant` sowie `PParticipant`, die den Operations-Implementierungsklassen der transaktionalen Web Services zur Verfügung stehen, kann von der Geschäftslogik in die Transaktion eingegriffen werden. Die Instanzen werden beim ersten Aufruf einer Business-Methode erzeugt und stehen dann für die Dauer der Transaktion, auch über mehrere Aufrufe, zur Verfügung. Abb. 5 gibt einen Überblick.

Auf Coordinator-Seite stehen `Coordinator` und `CParticipant` zur Verfügung, wobei jeder Participant, der sich beim Coordinator registriert, als `CParticipant`-Instanz zur Verfügung steht. Über diese Instanz können die WS-BusinessActivity-Operationen des Participants aufgerufen werden. Instanzen der Operations-Implementierungsklasse werden im Gegenzug über die `received...`-Methoden informiert, wenn ein Participant eine WS-BusinessActivity-Operation des Coordinators aufgerufen hat.

Auf Participant-Seite steht eine `PParticipant`-Instanz zur Verfügung, über die WS-BusinessActivity-Operationen des Coordinators aufgerufen werden können. Instanzen der Operations-Implementierungsklasse werden im Gegenzug über die `received...`-Methoden informiert, wenn der Coordinator der aktuellen Transaktion eine WS-BusinessActivity-Operation des Participants aufgerufen hat.

4.3 Asynchronität und Persistenz

Sowohl WS-Coordination als auch WS-BusinessActivity spezifizieren einen asynchronen Nachrichtenaustausch. Es müssen daher folgende Punkte beachtet werden:

Nachrichten müssen miteinander korreliert werden. WS-Addressing (Box u. a., 2004) spezifiziert hier verschiedene Mechanismen, die sich SOAP-Header-Informationen be-

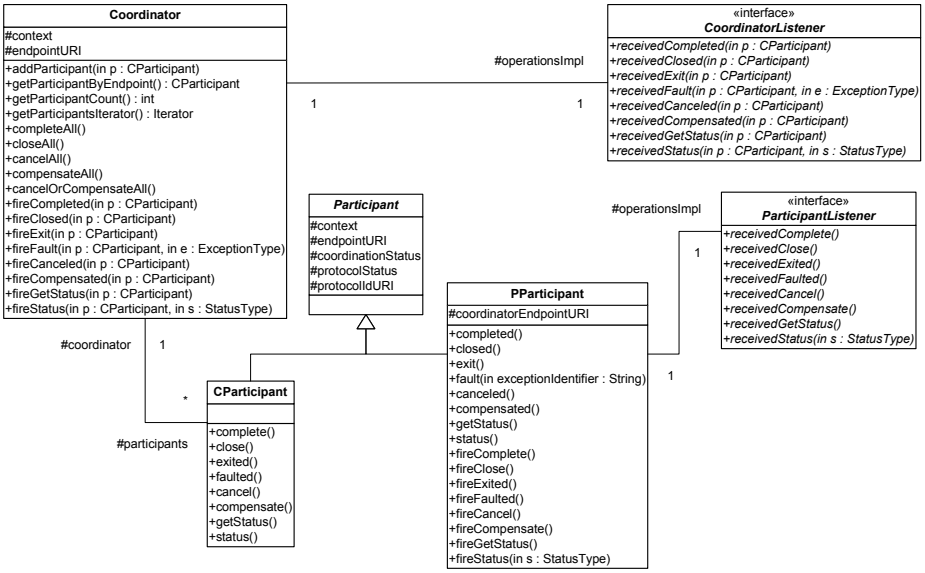

Abbildung 5: WS-BusinessActivty Coordinator und Participant API

dienen. Für diesen ersten Entwurf ist es ausreichend, neben dem Transaktionskontext noch die Adresse des Absenders als `From`-SOAP-Header mitzuschicken, da die WS-Coordination- und WS-BusinessActivity-Protokollnachrichten genug Informationen enthalten, die zusammen mit dem aktuellen Status des Protokolls zwischen Coordinator und Participant ausreichend sind, um das Protokoll gemäß Spezifikation abzuarbeiten. Für komplexere Kommunikationsmuster sieht WS-Addressing noch `To`-, `ReplyTo`-, `MessageID`- und `RelatesTo`-Header vor.

Wird eine transaktionale Operation eines Web Services als erste Operation innerhalb einer Transaktion aufgerufen, so muss zunächst das WS-Coordination-Protokoll abgearbeitet werden. Im Falle des Coordinators muss ein Transaktionskontext erzeugt werden, im Falle eines Participants muss dieser sich beim Coordinator registrieren. Da beide Operationen asynchron ablaufen, muss der ursprüngliche Aufruf zwischengespeichert werden, bis das Callback der WS-Coordination-Operation (`createCoordinationContextResponse` bzw. `registerResponse`) wieder aufgerufen wird. Erst dann kann die Operation ausgeführt werden.

Es wird also ein persistenter Zwischenspeicher benötigt, der dann über eine Transaktion hinweg auch den Zustand der Geschäftslogik (also das `...ServiceOperationsImpl`-Objekt inklusive dem `Coordinator`- und den `CParticipant`-Objekten bzw. dem `PParticipant`-Objekt sowie den aktuellen Transaktionskontext als `Coordination-Context`-Objekt) speichert. Der Speicher muss aus unterschiedlichen Java Virtual Ma-

chines (z.B. im Falle von Lastverteilung) zugreifbar sein. Hier können übliche J2EE-Mechanismen verwendet werden.

Weiterhin muss beachtet werden, dass die Participants theoretisch mehrere asynchrone Operationsaufrufe hintereinander empfangen können. Diese müssen dann alle solange zwischengespeichert werden, bis das WS-Coordination-Protokoll für die erste Operation (d.h. die Registrierung) durchgelaufen ist.

Der Autor empfiehlt, eine möglichst grobe Granulierung der transaktionalen Operationen vorzunehmen, so dass auf einem Service innerhalb einer Transaktion auch nur eine Operation transaktional aufgerufen wird. Eine Ausnahme bilden Callbacks aus transaktionalen Operationen der Participants an den Coordinator, diese werden jedoch in jedem Fall erst dann empfangen, wenn das WS-Coordination-Protokoll auf beiden seiten durchgelaufen ist. Die Callbacks können also ohne Zwischenspeichern sofort ausgeführt werden.

4.4 Transaktionsmuster und Service-Rollen

Ein Ziel des Entwurfs ist ja die Transparenz der Transaktionsabwicklung für den Geschäftslogik-Entwickler. Es exisitieren mehrere Parameter, die den Grad der Transparenz beeinflussen. Je nach Wahl der Parameter liegt die Verantwortlichkeit für die Einhaltung der Protokolle zu verschiedenen Teilen bei der Transaktions-Middleware (also bei den Erweiterungen des Web-Service-Containers sowie beim generierten Code) und bei der Geschäftslogik.

- WS-BA Koordinierungstyp (AtomicOutcome oder MixedOutcome): Legt fest, ob eine Transaktion erfolgreich terminieren darf, wenn nicht alle Participants erfolgreich terminieren. Im Falle von MixedOutcome muss die Geschäftslogik prüfen, welche Participants erfolgreich terminiert haben, und entscheiden, ob die Transaktion zurückgerollt (kompensiert) oder erfolgreich beendet werden soll.

- WS-BA Protokolltyp: Legt für ein Coordinator-Participant-Paar fest, ob der Coordinator „complete“ schicken soll, um dem Participant anzuzeigen, dass die Transaktion beendet werden soll (CoordinatorCompletion) oder ob einfach auf „completed“ vom Participant gewartet werden soll (ParticipantCompletion). In beiden Fällen muss der Coordinator auf „completed“-Nachrichten reagieren. Die Geschäftslogik muss prüfen, ob ein Ergebnis vorliegt und in diesem Fall alle Participants durch geeignete Operationen in den gewünschten Endzustand bringen, bevor die Transaktion mittels „close“ an alle Participants beendet wird.

- Service-Rolle (Coordinator oder Participant) und Art der Operation (Coordinator: Transaktions-Start oder Callback, Participant: Business-Operation): Für jede Kombination muss unterschiedlicher Transaktionslogik-Code generiert werden. Wenn die Business-Operation eines Participants die einzige Operation ist, die innerhalb einer

Transaktion ausgeführt werden soll, kann die Transaktionslogik automatisch „completed“ an den Coordinator schicken. Falls nicht, so muss die Geschäftslogik entscheiden, wann „completed“ gesendet wird.

- Behandlung von „exit“: Steigt ein Participant aus der Transaktion kontrolliert aus, so kann die Transaktionslogik das komplett abfangen. Die Geschäftslogik kann zusätzliche Operationen ausführen, muss dies aber nicht tun.

- Fehlerbehandlung: Hier sind verschiedene Muster denkbar. Automatisierbar wäre hier z.B. im Falle des AtomicOutcome-Koordinierungstyps das Zurückrollen (d.h. „cancel“ an alle Participants, die noch nicht „completed“ sind, „compensate“ an alle anderen) nach einem „fault“ (ggf. je nach Fehlercode).

Je nach Transaktionsmuster kommt also eine unterschiedliche Transaktionslogik zum Einsatz. Es besteht die Möglichkeit, dies zur Laufzeit zu entschieden, indem die Middleware entsprechend flexibel ausgelegt wird. Die Web Services werden jedoch effizienter arbeiten, wenn die Parameter zur Entwicklungszeit festgelegt werden können und entsprechender Code generiert werden kann.

5 Realisierung

Die Realisierung wurde auf Grund der geringen zur Verfügung stehenden Zeit als „Rapid Prototyping“ von drei Web Services vorgenommen, einem Coodinator und zwei Participants. Eine Operation des Coordinators startet eine Transaktion und ruft jeweils eine Operation der beiden Participants auf. Diese rufen dann jeweils eine Callback-Operation des Coordinators auf. Danach wird die Transaktion beendet.

Um den Code für die Web Services initial zu generieren, wurde zunächst eine übliche Web-Service-Beschreibung (WSDL) der drei Web Services erstellt und dann die WSDL-Elemente der WS-Coordination- und WS-BusinessActivity-Specifikationen inklusive XML-Namespaces hinzugefügt. Dabei wurden die Operationen nicht in eigenen PortTypes implementiert, sondern den Service-PortTypes hinzugefügt, da jeder PortType theoretisch unterschiedliche Transaktionsmuster annehmen kann.

Der Apache Axis Code-Generator wurde leicht angepasst, so dass die SOAP-Header-Informationen in der Service-Implementierungsklasse zur Verfügung stehen (siehe Abb. 4).

Die XML-Namespaces im WSDL-Dokument wurden so gewählt, das mit Hilfe eines geeigneten Mappings bei der Code-Generierung der Code für die Datentypen der unterschiedlichen Spezifikationen (WS-Addressing, WS-Coordination, WS-BusinessActivity) und die Service-spezifischen Datentypen in einer übersichtlichen Package-Struktur erzeugt wird.

Im generierten Code wurde die Transaktionslogik in der Service-Implementierungsklasse hinzugefügt. Die Geschäftslogik der Business-Methoden und Callbacks wurde in eine sepa-

rate Klasse (`...ServiceOperationsImpl`) ausgelagert. Eine Instanz dieser Klasse wird pro Web Service und Transaktion von der Transaktionslogik vorgehalten (siehe Abb. 4).

Es wurde lediglich die Logik implementiert, die für eine AtomicOutcome-Transaktion mit ParticipantCompletion einen fehlerfreien Durchlauf schafft. „Fault“, „exit“, „cancel“ und „compensate“ wurden nicht implementiert.

Die für die transaktionalen Aufrufe notwendige Service-Registry wurde als Dummy mit Hilfe eines Property-Files realisiert, welches zu den Service-Namen hart codiert die Service-URLs sowie die qualifizierten Namen der Service enthält. Dieses File muss für jeden beteiligten Web-Service-Container (Axis-Instanz) alle beteiligten Services enthalten.

Der Zwischenspeicher für Operationen, Service-Zustände und API-Objekte wurde ebenfalls als Dummy (`static Map ...`) implementiert, ist also nicht über mehrere JVMs hinweg funktionsfähig.

6 Fazit

6.1 Lessons learnt

Während der Entwurfs- und Implementierungsarbeiten wurden folgende Erkenntnisse gewonnen:

- Die Transaktionslogik kann bzw. muss u.U. stark mit der Applikationslogik verzahnt sein. Dabei sind verschiedene Muster von Transaktionsabläufen denkbar (ähnl. EJB), mit mehr oder weniger Transparenz für jedes Muster.
- Wie bei klassischen Transaktionen tritt ein gewisser Kommunikations-Overhead bei den ersten Operationen einer Transaktion sowie beim Durchlaufen des Terminierungsprotokolls auf.
- Die Vorarbeiten, also Entwicklungsumgebung aufsetzen und Spezifikationen lesen und verstehen, haben relativ viel Zeit gekostet.
- Die Implementierung der Protokolle ist an sich einfach (Zustandsautomat), aufwendig sind dagegen Synchronisierung, Persistenz und Service-Registry, für die jedoch auf bewährte Techniken zurückgegriffen werden kann.
- Die Spezifikationen haben viele Freiheitsgrade, was bezüglich Interoperabilität zu Problemen führen kann, da alle beteiligten Container die Spezifikationen in gleicher Weise unterstützen müssen. Es existiert mit dem Web Service Composite Application Framework (WS-CAF, (Bunting u. a., 2003a), (Bunting u. a., 2003b), (Bunting u. a., 2003c), siehe dazu auch (Little u. a., 2004, Kap. 10) und (Little, 2003)) ein konkurrierender Satz an ausführlicheren Spezifikationen, die diesbezüglich auf Eignung untersucht werden könnten.

- Wenn für bestimmte Operationen (z.B. Registrierung) sichere Dienstgüte-Aussagen getroffen werden können, sollte erwogen werden, diese synchron zu implementieren, um den Implementierungs- und Laufzeit-Aufwand der Zwischenspeicherung von Operationsaufrufen zu sparen.

6.2 Ausblick

Bezüglich des Projekts bestünde der nächste Schritt darin, die Implementierung zu vervollständigen. Dies betrifft das Protokoll („exit“, „fault“, „cancel“ und „compensate“) und die Kompatibilität zu WS-Addressing. Es müsste auch geprüft werden, ob es weitere Stellen gibt, an denen eine explizite Synchronisation der asynchronen Operationsaufrufe erfolgen muss. Zudem müsste ein vollständiger Code-Generator für verschiedene Transaktionsmuster implementiert werden.

Weiterhin ist die Transaktions-Middleware in den Enterprise Service Bus zu integrieren. Alle Aufrufe, auch die Protokollabwicklung, können über den ESB laufen, so dass verschiedene Systeme über unterschiedliche Mechanismen an Transaktionen teilnehmen können. Sämtliche Kommunikation muss dafür aber asynchron ausgelegt werden, was sich bis in die Implementierungsdetails der Geschäftsanwendungen hinein auswirken kann. Der ESB muss auch alle benötigten SOAP-Header korrekt weiterleiten. Es hat sich während des Projekts herausgestellt, dass der „Mule“-ESB leider nicht mit vertretbarem Aufwand angepasst werden kann. Mule verwendet zwar auch Axis als Web-Service-Container, aber die verwendete Version ist nicht einfach zugänglich bzw. austauschbar.

Eine Security-Schicht (z.B. auf Basis von Axis-Handlern wie im Projekt realisiert) ließe sich dagegen relativ leicht mit der Transaktions-Middleware integrieren.

In jedem Fall müssen alle beteiligten Web-Service-Container, die transaktionale Web Services beinhalten, um eine Transaktions-Middleware erweitert werden. Dies betrifft im Falle von J2EE unter Umständen auch andere Container als Axis. Weiterhin wären die .NET-Plattform sowie mobile Geräte mit J2ME und .NET Compact zu unterstützen.

Auf lange Sicht wird es unerlässlich sein, die Transaktions-Middleware für langlaufende Transaktionen noch einmal grundlegend neu zu entwerfen, mit den Zielen einer besseren Integration in Axis oder anderen Web-Service-Containern, dem Einsatz bewährter Technologien für Service-Registry und Zwischenspeicher sowie der Umsetzung auf möglichst viele Plattformen.

Da in realen Geschäftsanwendungen klassische (d.h. atomare) verteilte und lokale Transaktionen eine große Rolle spielen, sollten auch diese für Web Services möglich sein und von der Transaktions-Middleware unterstützt werden. Mit WS-AtomicTransaction (Carbrera u. a., 2005a) existiert dafür bereits eine Spezifikation und mit Apache Kandula (Weeratunge u. a., 2005) auch ein Prototyp, der lokale JTA-Transaktionen (Cheung und Matena, 2002) über Web-Service-Koordinierung zu verteilten Transaktionen zusammenfasst.

Literatur

[Alur u. a. 2003] ALUR, Deepak ; CRUPI, John ; MALKS, Dan: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2003

[Avar u. a. 2005] AVAR, Andras ; CHAPPELL, David ; DANIELS, Glen u. a.: *Apache Axis Version 1.3*. 2005. – URL `http://ws.apache.org/axis/`. – Open Source Web Services Container (21.12.2005)

[Box u. a. 2004] BOX, Don ; CHRISTENSEN, Erik ; CURBERA, Francisco u. a.: *Web Services Addressing (WS-Addressing)*. August 2004. – URL `http://www.w3.org/Submission/ws-addressing/`. – (15.01.2006)

[Bunting u. a. 2003a] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin u. a.: *Web Services Context (WS-Context) Version 1.0*. 2003. – URL `http://www.arjuna.com/standards/ws-caf/index.html`. – (21.12.2005)

[Bunting u. a. 2003b] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin u. a.: *Web Services Coordination Framework (WS-CF) Version 1.0*. 2003. – URL `http://www.arjuna.com/standards/ws-caf/index.html`. – (21.12.2005)

[Bunting u. a. 2003c] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin u. a.: *Web Services Transaction Management (WS-TXM) Version 1.0*. November 2003. – URL `http://www.arjuna.com/standards/ws-caf/index.html`. – (21.12.2005)

[Carbrera u. a. 2005a] CARBRERA, Luis F. ; COPELAND, George ; FEINGOLD, Max u. a.: *Web Services Atomic Transation (WS-AtomicTransaction) Version 1.0*. August 2005. – URL `http://www.ibm.com/developerworks/library/specification/ws-tx/#atom`. – (21.12.2005)

[Carbrera u. a. 2005b] CARBRERA, Luis F. ; COPELAND, George ; FEINGOLD, Max u. a.: *Web Services Business Activity Framework (WS-BusinessActivity) Version 1.0*. August 2005. – URL `http://www.ibm.com/developerworks/library/specification/ws-tx/#ba`. – (21.12.2005)

[Carbrera u. a. 2005c] CARBRERA, Luis F. ; COPELAND, George ; FEINGOLD, Max u. a.: *Web Services Coordination (WS-Coordination) Version 1.0*. August 2005. – URL `http://www.ibm.com/developerworks/library/specification/ws-tx/#coor`. – (21.12.2005)

[Cheung und Matena 2002] CHEUNG, Susan ; MATENA, Vlada: *Java Transaction API (JTA) Version 1.0.1B*. November 2002. – URL `http://java.sun.com/products/jta/`. – (21.12.2005)

[DeMichiel 2003] DEMICHIEL, Linda G.: *Enterprise JavaBeans(TM) Specification, Version 2.1*. November 2003. – URL `http://java.sun.com/products/ejb/`. – (21.12.2005)

[Eclipse 2005] *Eclipse Development Platform*. 2005. – URL `http://www.eclipse.org/`. – (14.01.2006)

[Gerlach 2005a] GERLACH, Martin: *Langlebige Transaktionen in dienstorientierten Umgebungen*. Dezember 2005. – URL `http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master05-06/gerlach/abstract.pdf`. – Ausarbeitung zum Seminar (SR) des Master-Studiengangs Informatik an der HAW Hamburg (10.01.2006)

[Gerlach 2005b] GERLACH, Martin: *Service-Oriented Architectures: Transaktionsmanagement mit Services und Geschäftsprozessen*. Mai 2005. – URL `http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/gerlach/abstract.pdf`. – Ausarbeitung zum Seminar Anwendungen 1 des Master-Studiengangs Informatik an der HAW Hamburg (21.12.2005)

[Gudgin u. a. 2003] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noad u. a.: *SOAP Version 1.2*. 2003. – URL `http://www.w3.org/TR/soap/`. – (21.12.2005)

[JBoss 2005] *JBoss Application Server*. 2005. – URL `http://www.jboss.com/products/jbossas/`. – (14.01.2006)

[JBoss-Eclipse 2005] *JBoss Eclipse IDE*. 2005. – URL `http://www.jboss.com/products/jbosside/`. – (14.01.2006)

[Little 2003] LITTLE, Mark: Web Services transactions: Past, present and future. In: *Proceedings of the XML Conference and Exposition*. Philadelphia, USA, 2003

[Little u. a. 2004] LITTLE, Mark ; MARON, Jon ; PAVLIK, Greg: *Java Transaction Processing Design and Implementation*, Prentice Hall, 2004

[Mule 2005] *Mule ESB Framework*. 2005. – URL `http://mule.codehaus.org/`. – (14.01.2006)

[Rubarth 2005a] RUBARTH, Thies: *Sicherheitskonzepte in SOA auf Basis sicherer Web Services*. Dezember 2005. – URL `http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/rubarth/abstract.pdf`. – Ausarbeitung zum Seminar (SR) des Master-Studiengangs Informatik an der HAW Hamburg (14.01.2006)

[Rubarth 2005b] RUBARTH, Thies: *Web Service Security*. Juni 2005. – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/rubarth/abstract.pdf. – Ausarbeitung zum Seminar Anwendungen 1 des Master-Studiengangs Informatik an der HAW Hamburg (14.01.2006)

[Stegelmeier 2005] STEGELMEIER, Sven: *Service Oriented Architecture und Enterprise Service Bus*. Juli 2005. – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/stegelmeier/abstract.pdf. – Ausarbeitung zum Seminar Anwendungen 1 des Master-Studiengangs Informatik an der HAW Hamburg (14.01.2006)

[Weeratunge u. a. 2005] WEERATUNGE, Dasarath ; WEERAWARANA, Sanjiva ; GUNARATHNE, Thilina: *Apache Kandula*. 2005. – URL http://ws.apache.org/kandula/. – Versuch einer Open Source Implementierung von WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity auf Basis von Axis 2 (21.12.2005)

[Weerawarana u. a. 2005] WEERAWARANA, Sanjiva ; CURBERA, Francisco ; LEYMANN, Frank ; STOREY, Tony ; FERGUSON, Donald F.: *Web Services Platform Architecture*, Pearson Education, 2005

[Zimmermann u. a. 2003/2005] ZIMMERMANN, Olaf ; TOMLINSON, Mark ; PEUSER, Stefan: *Perspectives on Web Services*, Springer, 2003/2005