Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Martin Gerlach

Entwicklung eines Transaktions-Frameworks für mobile Web Services

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Martin Gerlach

Entwicklung eines Transaktions-Frameworks für mobile Web Services

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachterin: Dipl. Inform. Birgit Wendholt

Abgegeben am 10. August 2006

Martin Gerlach

Thema der Masterarbeit
Entwicklung eines Transaktions-Frameworks für mobile Web Services
Stichworte
Dienste, Dienstorientierte Architektur, Web Services, Mobile Web Services, Verteilte Transaktionen, Langlebige Transaktionen
Kurzzusammenfassung
In dieser Arbeit werden Erweiterungen eines existierenden J2ME-Web-Service-Frameworks um die Fähigkeit, Web Services an langlebigen Transaktionen teilnehmen zu lassen, beschrieben. Ausgehend von allgemeinen Betrachtungen zu dienstorientierten Architekturen und langlebigen Transaktionen vor dem Hintergrund von Workflow-Management und Enterprise Application Integration werden Spezifikationen für Web-Service-Transaktionen auf Eignung für mobile Web Services untersucht. Anhand ausgewählter Spezifikationen werden Anforderungen an die Erweiterung des Frameworks erarbeitet und in einer Systemarchitektur und einem Entwurf umgesetzt. Zum Beleg der Realisierbarkeit werden die prototypische Implementierung der Framework-Erweiterungen sowie eine Beispielanwendung auf einem stationären Server und einem mobilen Gerät beschrieben. Abschließend werden kritische Punkte, Konfigurationsempfehlungen sowie zukünftige Anwendungs- und Weiterentwicklungsmöglichkeiten der entwickelten Technologie zusammengefasst und diskutiert.

Martin Gerlach

Title of the paper
Development of a Transaction Framework for Mobile Web Services
Keywords
Services, Service-Oriented Architecture, Web Services, Mobile Web Services, Distributed Transactions, Long Running Transactions, Long Lived Transactions
Abstract
In this thesis, the author describes extensions to an existing J2ME Web Services framework so that Web Services hosted on small mobile devices can participate in long running transactions. Starting with some general views on how Service Oriented Architectures and long running transactions can facilitate Workflow Management and Enterprise Application Integration, existing Web Service transaction specifications are analyzed regarding their suitability for mobile Web Services. A set of specifications is then chosen as basis for developing requirements, which are subsequently implemented in a system architecture and a design. A prototype of the framework extensions and a sample application running on a stationary server and a mobile device are described, proving the feasibility of the design. Finally, the author summarizes and discusses critical issues and configuration recommendations, as well as future applications and enhancements for the developed technology.

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

Ich danke Birgit Wendholt und Prof. Dr. Kai von Luck von der Hochschule für Angewandte Wissenschaften Hamburg für die gute Betreuung, die produktive Zusammenarbeit und das ausführliche Feedback während der Erstellung dieser Arbeit.

Weiterhin bedanke ich mich herzlich bei Heike Reinking, Steffen Lassahn und Boris Petitjean für umfassendes Feedback und ausführliche Korrekturarbeiten.

Den Verantwortlichen für die Durchführung des Masterstudiengangs Informatik an der Hochschule für Angewandte Wissenschaften Hamburg gilt mein besonderer Dank für die hervorragende Ausbildung in diesem Studiengang.

Martin Gerlach, August 2006

1. Einleitung

1.1. Problemstellung

Im Rahmen von Enterprise Application Integration (EAI) spielt die IT-seitige Unterstützung von Geschäftsprozessen in Form so genannter Workflows eine große Rolle. Diese wird seit Anfang der 1990er Jahre zunehmend mit EAI-Tools realsisiert, die verschiedene, heterogene Systeme einbinden, welche im Rahmen der Prozessautomatisierung an den Workflows beteiligt sind. Anwendungsbeispiele umfassen Reengineering und Optimierung von Geschäftsprozessen, Unternehmens-Zusammenschlüsse und die Bildung „virtueller Unternehmen“ mittels „Business-to-Business“-Kollaboration (B2B).

Dienstorientierte Architekturen (Service Oriented Architectures, SOAs), in denen Kernaufgaben als Dienste implementiert werden und Workflows mittels Dienstaufrufen realisiert werden, eigenen sich besonders gut dafür, heterogene Systeme zu integrieren und haben in den letzen Jahren zunehmende Verbreitung gefunden. Dabei spielen insbesondere Web Services als technische Umsetzung des Dienst-Konzepts eine wichtige Rolle.

Datenkonsistenz und der Schutz vor dem Verlust unternehmens- und geschäftskritischer Daten sind dabei zentrale Aspekte der IT-Anwendungsentwicklung. Unabhängig von sich ständig weiterentwickelnden Technologien und wechselnden Paradigmen der Softwareentwicklung sind Transaktionen Mittel der Wahl, um Konsistenz von Daten und Fehlertoleranz von Systemen zu realisieren.

Bestimmte Geschäftsprozesse bzw. deren IT-seitige Repräsentation als Workflow sind per Definition langlebig, da sie komplizierte Berechnungen oder manuelle Vorgänge beinhalten. Ebenso können in dienstorientierten, offenen Umgebungen oft keine Aussagen über die Laufzeit von Teilprozessen oder über Antwortzeiten getroffen werden. Transaktionale Sicherung ist auch hier notwendig, wenn die Konsistenz von kritischen Daten gesichert sein soll. Es werden daher Konzepte für eine transaktionale Sicherung langlebiger Prozesse, also „langlebige Transaktionen“ benötigt.

Bereits 1981 forderte Jim Gray, dass die stringenten ACID-Anforderungen (Atomicity, Consistency, Isolation, Durability) an übliche transaktionale Systeme für langlebige Transaktionen gelockert werden müssten (Gray, 1981).

Kleine mobile Geräte wie Mobiltelefone, PDAs (Personal Digital Assistants) und die so genannten Smartphones (Kombinationen aus Mobiltelefon und PDA), haben seit einigen Jahren eine explosive Verbreitung erfahren.[1] Die Anforderungen an diese Geräte und der Umfang der von diesen Geräten übernommenen Aufgaben wachsen nach (Schrörs, 2005, 1.1) stetig. Werden diese Aufgaben als Teil einer verteilten Anwendung ausgeführt, so ist es wünschenswert, dass auch diese mobilen Geräte mit in die transaktionale Absicherung einbezogen werden.

Besondere Herausforderungen bestehen dabei im Umfang der zur Verfügung stehenden Ressourcen im Vergleich zu PCs oder Laptops und in der Tatsache, dass mobile Geräte nicht ununterbrochen online sind. Die Konnektivität von Mobiltelefonen ist durch lange, gewünschte (z. B. nachts) und kurze, unerwünschte (z. B. in einem Eisenbahntunnel) Offline-Zeiten gekennzeichnet. Der Ausdruck „kleine mobile Geräte“ wird in dieser Arbeit verwendet, um diese Herausforderungen zu betonen.

1.2. Zielsetzung

Im Rahmen dieser Arbeit sollen Lösungen erarbeitet werden, um mobile Web Services in langlebige Transaktionen einzubinden. Ausgehend von Betrachtungen des Dienstkonzepts und des Workflow-Managements ist das Ziel, existierende Spezifikationen für Web-Service-Transaktionen vorzustellen, kritisch zu betrachten und mit herkömmlichen Verfahren des verteilten Transaktionsmanagements zu vergleichen, wobei langlebige Transaktionen mit mobilen Transaktionsteilnehmern im Vordergrund stehen sollen.

Ein im Rahmen einer Diplomarbeit (Schrörs, 2005) entwickeltes und anschließend (Schrörs, 2006a) erweitertes, auf der Java 2 Micro Edition (J2ME) basierendes Framework für Web Services auf kleinen mobilen Geräten soll zur Verdeutlichung und zur Feststellung der Anwendbarkeit einiger ausgewählter Spezifikationen um entsprechende Fähigkeiten erweitert werden. Dabei ist nicht die möglichst vollständige Implementierung das Ziel, sondern Analyse und Entwurf der benötigten neuen Komponenten und deren Kernfunktionalitäten.

Anhand eines einfachen Beispiels, welches im Laufe dieser Arbeit an mehreren Stellen herangezogen wird, soll die Abwicklung langlebiger Transaktionen mit Web Services erklärt werden:

> Im Laufe eines als Web Service implementierten Workflows zur Unterstützung eines Geschäftsprozesses ist für verschiedene Entscheidungen die Freigabe

[1] Siehe z. B. (IZT, 2001), eine Zukunftsstudie aus dem Jahr 2001, die diverse Projektionen für die Verbreitung mobiler Anwendungen ab dem Jahr 2005 enthält, oder entsprechende Berichte auf http://www.gartner.com

durch eine verantwortliche Person, den so genannten „Freigabemanager“, notwendig, damit der Prozess erfolgreich beendet werden kann. Diese Person ist viel unterwegs. Sie verfügt daher über ein Handy, Smartphone oder sonstiges kleines mobiles Gerät, auf dem ein Teil der Workflow-Anwendung ebenfalls in Form eines Web Services installiert ist. Dieser Anwendungsteil wird vom zentralen Workflow-System über einen Web-Service-Aufruf benachrichtigt, sobald eine neue Freigabe benötigt wird. Die Anfrage wird dem Freigabemanager auf dem Bildschirm seines mobilen Gerätes angezeigt und er kann durch einfache Interaktion mit dem Gerät die Freigabe erteilen, ablehnen oder delegieren. Die Entscheidung des Freigabemanagers wird dann vom mobilen Teil der Anwendung an das zentrale System übermittelt. Das gesamte System soll auch Situationen handhaben, in der eine Freigabeanfrage im Laufe des Prozesses zurückgezogen wird. Dies soll sowohl vor als auch nach der Antwort des Freigabemanagers möglich sein.

Dieses Beispiel soll schließlich im Rahmen einer prototypischen Implementierung die Realisierbarkeit des zu erstellenden Entwurfs demonstrieren.

1.3. Zielgruppen

Ausgehend von der Zielsetzung bietet diese Arbeit Informationen für Personen, die sich mit der technischen Umsetzung von SOAs unter Einbeziehung mobiler Geräte befassen. Die technische Ausrichtung bedingt das Vorhandensein von Grundwissen über Web Services und die zugrunde liegenden Spezifikationen und Technologien. Folgende Zielgruppen sollen angesprochen werden:

- Entwickler und andere Mitglieder eines Projektteams, die die Konzepte und Zusammenhänge der für diese Arbeit maßgeblich wichtigen Bereiche kennen lernen wollen, finden in den Kapiteln 2 und 3 Informationen.
- Entwickler, die im Rahmen von SOA-Umsetzungen langlebige Transaktionen für Web-Services umsetzen wollen, finden im Entwicklungsteil dieser Arbeit in den Kapiteln 4, 5 und 6 Informationen.

Da der Entwicklungsteil dieser Arbeit auf (Schrörs, 2005) und der zugehörigen Erweiterung (Schrörs, 2006a) aufbaut, ist es sinnvoll, sich zunächst mit diesen Arbeiten vertraut zu machen.

1.4. Inhaltlicher Aufbau der Arbeit

In Kapitel 2, „Motivation“, wird zunächst erläutert, weshalb Dienste und dienstorientierte Architekturen für viele Probleme der Softwareentwicklung zur Unterstützung von Geschäftsprozessen sinnvolle Lösungsansätze darstellen. Weiterhin wird dort genauer dargestellt, wozu langlebige Transaktionen dienen und wie diese prinzipiell funktionieren, bevor mobile Dienste (im Wesentlichen Web Services) motiviert werden. Abschließend wird außerdem noch die Verwendung von Frameworks und Entwurfsmustern motiviert.

Kapitel 3, „Grundlagen“, legt erforderliche Grundlagen aus den Bereichen „SOA“, „Klassisches Transaktionsmanagement“, „Workflow-Management mit langlebigen Transaktionen und Web Services“, „Transaktionsmanagement für Web Services“ sowie „Mobile Web Services“. Es wird in den einzelnen Abschnitten auch auf existierende Arbeiten in den Themenfeldern eingegangen und die Arbeit gegen diese abgegrenzt. Weiterhin wird eine Klassifizierung von für Frameworks geeigneten Entwurfsmustern vorgestellt. Abschließend werden weitere Ziele vor dem Hintergrund der verschiedenen Motivationen und Grundlagen definiert, um auf die folgenden Kapitel vorzubereiten.

Kapitel 4, „Analyse“, beschreibt die softwaretechnische Analyse (Anforderungen, Anwendungsfälle) des auf dem genannten Framework aufbauenden Transaktions-Frameworks für mobile Web Services. In Kapitel 5, „Entwurf“, werden Architektur und Komponenten des Frameworks genauer beschrieben.

Ausgewählte Aspekte der Implementierung werden in Kapitel 6 vorgestellt, bevor Fazit und Ausblick in Kapitel 7 die Arbeit abschließen.

2. Motivation

In diesem Kapitel wird der Einsatz der in dieser Arbeit grundlegenden Konzepte und Technologien — Dienstorientierte Architekturen, Web Services, langlebige Transaktionen, Kompensation, mobile Web Services sowie Frameworks — motiviert. Das in 1.2 beschriebene Beispiel wird dabei im Abschnitt über Mobilität und mobile Web Services (2.3) aufgegriffen.

2.1. Dienstorientierte Architekturen und Web Services

Erfahrungen aus der Softwareentwicklung zur Unterstützung von Geschäftsprozessen und im Rahmen dessen zur Integration von komplexen IT-Infrastrukturen (EAI) in Unternehmen haben nach (Krafzig u. a., 2005, Kap. 1) gezeigt, dass ein bestehendes System mit zunehmender Anzahl von Änderungen immer weniger wartbar wird. D.h. dass jede weitere Änderung aufwendiger durchzuführen ist, bis ein Punkt erreicht ist, an dem das System aus Rentabilitätsgründen ersetzt werden muss. Die Schwierigkeiten sind dabei sowohl technischer Art (Änderungen sind am laufenden System vorzunehmen), als auch organisatorischer Art (für die Produktionsphase ist ein anderes Team verantwortlich als für die Entwicklungsphase, außerdem können enge Budgets zu mangelnder Qualität, sowohl von ursprünglicher Entwicklung als auch von Änderungen, führen).

Aus diesen Erfahrungen lassen sich grundlegende Anforderungen für Softwarearchitekturen[2] wie etwa Einfachheit, Flexibilität, Wartbarkeit, Wiederverwendbarkeit und Entkoppelung von Funktionalität und Technologie, ableiten.

Konzeptionell sollen Dienste aktive Entitäten der Geschäftswelt repräsentieren. Durch die konzeptionellen Eigenschaften (s. o.) sollen sie das Erstellen von komplexen Systemen erleichtern. Dienstorientierte Architekturen als Weiterentwicklung von Distributed Object Systems und Message Oriented Middleware (Weerawarana u. a., 2005) erfüllen bei gelungener Umsetzung diese Anforderungen und bilden daher Lösungsansätze für EAI-Aufgaben.

[2] Der Begriff Softwarearchitektur wird hier im Sinne der Beschreibung der Struktur und Eigenschaften von Systemkomponenten und deren Schnittstellen gebraucht.
Unter http://www.sei.cmu.edu/architecture/definitions.html sind einige mögliche Definitionen des Begriffs zu finden.

Dazu tragen insbesondere die *konzeptionellen* Eigenschaften von Diensten bei:

- Lose Kopplung (minimale Abhängigkeit von anderen Diensten),
- Beschreibbarkeit (wohldefinierte Schnittstelle),
- Vertragserfüllung (Einhalten der in der Schnittstelle beschriebenen Vereinbarungen hinsichtlich Funktionalität und Dienstgüte),
- Autonomie (volle Kontrolle über die gekapselte Logik),
- Abstraktion (Betonung der Funktionalität und Verbergen technischer Details),
- Wiederverwendbarkeit,
- Zusammensetzbarkeit,
- Zustandslosigkeit (vor der Installation bzw. der Inbetriebnahme), sowie
- Entdeckbarkeit (für Anwendungen leicht auffindbar).

In (Erl, 2005, Kap. 3.1 und 8) werden diese Eigenschaften näher beschrieben und zueinander in Beziehung gesetzt.

Dienstorientierte Architekturen werden nun unter Ausnutzung dieser Eigenschaften verwendet, um dienstorientierte Systeme zu entwerfen, welche nach (Erl, 2005, 3.2) dann als Resultat folgende Eigenschaften aufweisen:

- Garantierte Dienstgüte,
- Verwendung offener Standards,
- Unterstützung von Dienstanbietervielfalt,
- inhärente Wiederverwendbarkeit,
- Interoperabilität, Autonomie und lose Kopplung der Systemkomponenten,
- Zusammensetzbarkeit auf architektonischer Ebene,
- einfache Erweiterbarkeit und Änderbarkeit bzw. Wartbarkeit („Organizational Agility“),
- Ermöglichung dienstorientierter Modellierung von Geschäftslogik, sowie
- schichtweise Abstraktion.

Bei gelungener Umsetzung sollten dienstorientierte Architekturen also EAI-Aufgaben wesentlich erleichtern.

Das Bündeln von Software zu Paketen mit dem Ziel allgemeinerer Benutzbarkeit und Anwendbarkeit ist nach (Weerawarana u. a., 2005) seit den Anfängen der Softwareentwicklung

ein wichtiges Gebiet. Flexiblere Einsetzbarkeit und einfaches Deployment sind dabei ebenso Ziele wie ein besseres Verständnis komplexer Systeme, was durch Trennung und Gruppierung von Verantwortlichkeiten erreicht werden soll. Diese Zielsetzungen erinnern bereits stark an das Dienstkonzept. Die „Entwicklungsstufen der Softwarebündelung“, nämlich Funktionsbibliotheken, Klassenbibliotheken und Komponentenumgebungen, sowie dazugehörige Kommunikationsinfrastrukturen wie RPC (Remote Procedure Call) oder Messaging lassen sich daher bereits als technische Ausprägungen des Dienstkonzepts auffassen.

Komponenten im Sinne von (Szyperski, 1998) weisen dabei am ehesten die für Dienstorientierung geforderten Eigenschaften auf, insb. sind sie im Gegensatz zu Objekten lose gekoppelt und grob granuliert: Eine Komponente benötigt weniger, im Idealfall keine weiteren Komponenten, um ihre Aufgaben zu erfüllen (Autonomie, lose Kopplung) und diese Aufgaben umfassen dabei mehr Bearbeitungsschritte als im Fall von Objekten (grobe Granulierung). Dienstgüte- und andere Laufzeitanforderungen werden für Komponenten separat spezifiziert und Komponenten transportieren mehr Semantik (im Sinne von Geschäftsanwendungen) als Klassenbibliotheken, die meist für spezielle Problemgebiete zusammengestellt werden. Ausführliche Vergleiche von Objekten und Komponenten finden sich in (Szyperski, 1998), die Entwicklung aus Sicht der Dienstorientierung wird in (Weerawarana u. a., 2005, Kap. 1), (Krafzig u. a., 2005, Kap. 2) und (Dokovski u. a., 2004) beschrieben. Komponenten können also konzeptionell als dienstorientiert bezeichnet werden.

Dienstorientierung fordert als Folge der Forderung nach Integrationsfähigkeit und Interoperabilität eine stärkere Abstraktion von der Implementierung und dem Kommunikationsprotokoll, als in bekannten Umsetzungen von Komponentenarchitekturen wie CORBA (Common Object Request Broker Architecture, s. z. B. (Siegel, 2000)) und EJB (Enterprise JavaBeans, spezifiziert in (EJB-2.1, 2003)) realisiert. Zum Beispiel können Operationen einer Dienstschnittstelle in der Dienstbeschreibung an verschiedene Kommunikationsmuster (z. B. RPC), Datentypen-Encodings sowie Transportmechanismen (z. B. HTTP oder JMS (Java Message Service)) gebunden werden, während Komponentenarchitekturen wie CORBA hier spezifische Vorgaben (IIOP, Internet Inter-ORB (Object Request Broker) Protocol, mit festgelegtem binären Format) machen. Damit wird auch der Anforderung nach Entkoppelung von Funktionalität und Technologien Rechnung getragen.

Aus den Anforderungen an Integrations-Architekturen wie oben und in (Weerawarana u. a., 2005, Kap. 1) beschrieben, lassen sich entsprechende *technische* Anforderungen an Dienste ableiten. Web Services als „virtuelle Komponenten“ (Leymann, 2003) werden am häufigsten zur Realisierung des Dienstkonzepts und damit zum Aufbau von dienstorientierten Architekturen verwendet. Web-Service-Technologien werden mittlerweile von fast allen Softwareherstellern angeboten und unterstützt, worin nach (Leymann, 2003) auch die Hauptneuigkeit gegenüber anderen Technologien für verteilte Systeme liegt, denn konzeptionell böten Web Services „nichts fundamental Neues“. Die technischen Eigenschaften von Web Services sorgen dabei nach (Erl, 2005) — bei Verwendung von bestimmten Erweiterungen

des grundlegenden, auf XML, XML-Schema, Simple Object Access Protocol SOAP (SOAP, 2003), Web Services Description Language WSDL (WSDL-1.1, 2001) und Universal Description, Discovery and Integration UDDI (UDDI, 2006) aufbauenden Basis-Frameworks für Web Services — für die implizite Erfüllung einiger Anforderungen an dienstorientierte Architekturen. Auch diese Arbeit konzentriert sich daher auf Web Services. In 3.1 wird auf diese Erweiterungen des Basis-Frameworks weiter eingegangen.

Hinsichtlich des für diese Arbeit zentralen Themas „Transaktionsmanagement“ lassen sich aufgrund der konzeptionellen Ähnlichkeit von Komponenten und Web Services grundlegende Konzepte des Transaktionsmanagements von bekannten Konzepten für Komponenten auf Web Services übertragen. In den Abschnitten 3.2 sowie 3.4 wird auf die Einzelheiten eingegangen.

2.2. Langlebige Transaktionen und Kompensation

Wie in 1.1 bereits angedeutet, spielen insbesondere in der verteilten Abarbeitung von Workflows zur Unterstützung von Geschäftsprozessen langlebige Transaktionen eine bedeutende Rolle. Dabei ist der Begriff Transaktion hier nicht als klassische (Datenbank-)Transaktion mit ACID-Eigenschaften nach (Gray, 1981) bzw. (Gray und Reuter, 1993) zu verstehen, sondern als „lange“ andauernder Vorgang, der wiederum aus weiteren verteilten Vorgängen zusammengesetzt ist. Standard-Beispiele für langlebige Transaktionen sind kombinierte Buchungen von Service-Leistungen, z. B. Reisen mit Flug, Hotel, Mietwagen oder Ausflüge mit verschiedenen Aktivitäten. Abb. 2.1 zeigt eine Abfolge von Dienstaufrufen, die innerhalb eines Reisebuchungsprozesses erfolgen, welcher durch den Web Service s0 repräsentiert wird.

Die von s0 aufgerufenen Dienste führen dabei folgende Aktivitäten durch, deren gesamte Dauer für eine übergreifende verteilte Transaktion zu groß ist ist.

- s1 Flugbuchung
- s2 Hotelbuchung
- s3 Mietwagenbuchung
- s4 Buchung von Zusatzleistung am Zielort, welche jedoch nicht verfügbar sind. Dies wird als Fehler angesehen und entsprechend an s0 zurückgemeldet.
- s2’ Stornierung des Hotels
- s3’ Stornierung des Mietwagens
- s5 Alternative Hotelbuchung

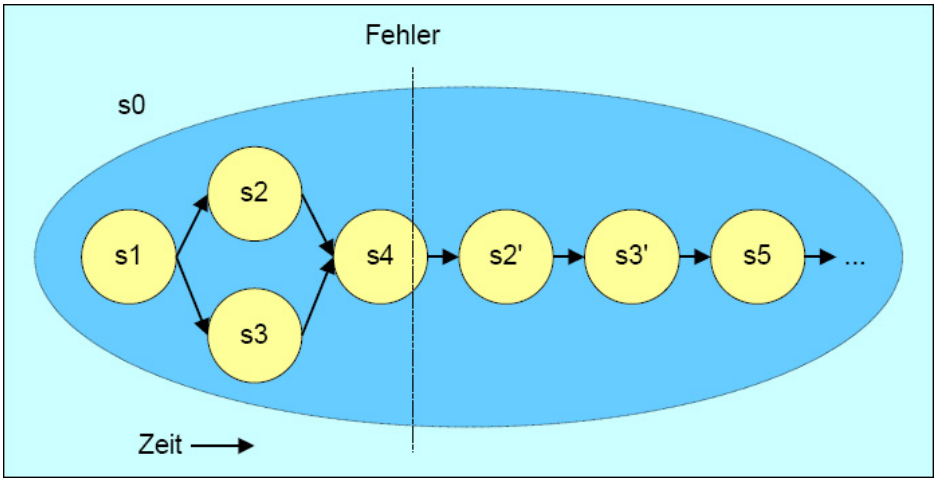

Abbildung 2.1.: Beispiel für eine langlebige Transaktion mit Kompensation nach (Little, 2003a) mit leichten Änderungen

- etc. Weitere Buchungen

In diesem Beispiel wird selektiv storniert, d. h. der Dienst s0 entscheidet aufgrund des Fehlers, welche Leistungen storniert werden müssen und welche neu gebucht werden müssen. Dabei wird angenommen, dass der Dienst s4 selbst dafür sorgt, dass eventuelle von diesem Dienst durchgeführte Datenänderungen rückgängig gemacht werden (z. B. durch ein Rollback auf eine lokale Datenbanktransaktion). Wäre der Fehler technischer Art, d. h. wäre der Zustand von s4 nach dem Fehler unbekannt, so könnte s0 auch entscheiden, allen bisher erfolgreich beendeten Diensten die Anweisung zur Stornierung zu schicken und selbst eine Fehlermeldung zurückzumelden, z. B. an eine entsprechende Benutzerschnittstelle.

Der Überbegriff für Stornierung und ähnliche Vorgänge ist „Kompensation“. Entscheidend bei der Kompensation ist in diesem Beispiel, dass der Dienst s0 nicht wissen muss, wie die Stornierung für die einzelnen Dienste aussieht. Vielmehr wird den Diensten ein Kompensationsbefehl geschickt. Anhand eines Transaktionskontextes (s. 3.2 und 3.4) können die Dienste dann selbst die Stornierung vornehmen. Es ist hierzu anzumerken, dass Stornierung von einem Dienst auch als eigenständige Operation angeboten werden kann, die innerhalb anderer langlebiger Transaktionen aufgerufen werden kann. Die Kompensation dazu wäre dann das erneute Buchen der Leistung.

Das Konzept langlebiger Transaktionen mit Kompensation ist also für Dienste und damit für Web Services relevant. Langlebige Transaktionen mit Kompensation erfordern ein spezielles Transaktionsmanagement (Koordination), welches Unterschiede zum Management klassischer ACID-Transaktionen aufweist und Elemente aus dem Workflow-Management enthält.

Es sollte also zusätzlich zum Management von ACID-Transaktionen auch das Workflow-Management daraufhin untersucht werden, ob Konzepte auf das Transaktionsmanagement für Web Services übertragbar sind. Einzelheiten werden in Abschnitt 3.3 sowie 3.4 erklärt.

2.3. Mobilität und mobile Web Services

2.3.1. Überblick

Wie in Abb. 2.2 angedeutet, hat sich das Verhältnis der möglichen Anzahl der Computer zur Anzahl der Menschen zumindest in westlichen Industrienationen mit der Zeit umgekehrt. Jeder Mensch hat mittlerweile die Möglichkeit eine Vielzahl „zunehmend kleinere, mobile Geräte“ zu nutzen, die „spontane Netzwerke“ bilden und „immer anspruchsvollere Aufgaben“ übernehmen, die „zum Teil schon vollkommen selbständig“ erledigt werden (Schrörs, 2005, 1.1).

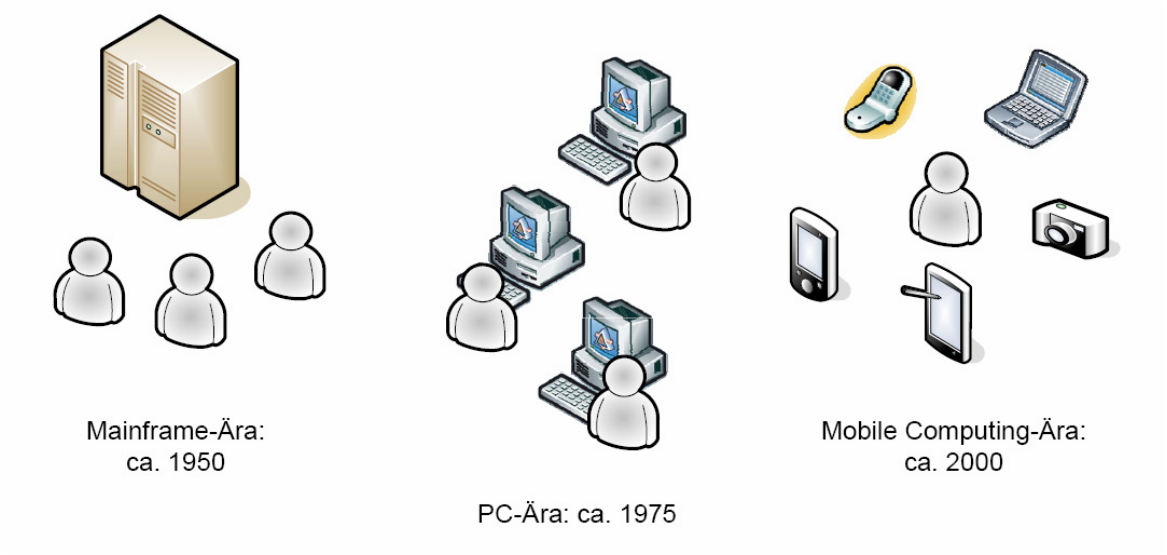

Abbildung 2.2.: Entwicklung des Mensch-zu-Computer-Verhältnisses (Thomé, 2006)

Wie in 2.1 bereits herausgestellt wurde, haben sich Web Services bei der Integration von Geschäftsanwendungen als Lösung für heterogene Systeme bewährt. Mobile Geräte und die darauf laufenden Anwendungen sind höchst heterogen: Die Geräte verwenden unterschiedliche Betriebsysteme und verfügen über unterschiedliche Netzwerkschnittstellen und unterschiedliche Softwareausstattungen. Es liegt daher nahe, Web Services auch für die Interaktion mit und zwischen kleinen mobilen Geräten einzusetzen.

Dazu ist es nicht nur notwendig, Web Services von mobilen Geräten aus aufzurufen (Client-Funktionalität), sondern Web Services müssen auch auf den Geräten betrieben werden (Server-Funktionalität). Es wird also ein Web-Service-Framework für mobile Geräte benötigt.

Die auf den mobilen Geräten angebotenen Web Services sollten an Transaktionen teilnehmen können, um einen konsistenten Ablauf der Geschäftsprozesse sicherzustellen, von denen die Web Services benutzt (aufgerufen) werden. Das Framework sowie die darin implementierten Web Services sollten also entsprechende Mechanismen unterstützen.

Die Ressourcen kleiner mobiler Geräte reichen für lokale ACID-Transaktionen nicht aus, da hierfür eine aufwendige Verwaltung von Sperren für alle verwendeten, kritischen Systemteile erforderlich wäre. Und selbst wenn eine Anwendungen ACID-Transaktionen unterstützte, könnten die bei der Kommunikation mit mobilen Geräten möglichen Verzögerungen und Verbindungsabbrüche zumindest im Falle längerer Offline-Zeiten zu zu langen Sperrungen wichtiger Systemteile führen. Aus diesen Gründen ist es zunächst ausreichend, langlebige Transaktionen mit Kompensation zu unterstützen.

2.3.2. Beispiel-Szenarien

Mobiltelefone, PDAs und andere kleine mobile Geräte haben nicht nur den Vorteil, dass sie jederzeit und überall nutzbar und vernetzt sein können, sondern sie haben, unter Umständen mit gewisser Zusatzhardware, umgebungsbezogene Daten zur Verfügung, aus denen Nutzen gezogen werden kann. Es gilt nun, diese Daten auch Anwendungen zur Verfügung zu stellen, die nicht auf dem mobilen Gerät selbst ausgeführt werden, sondern auf anderen mobilen Geräten oder stationär. Dafür gibt es folgende, in Abb. 2.3 visualisierte Möglichkeiten unter Verwendung von Web Services[3]:

1. Serverbasiert: Das mobile Gerät schickt die Daten in regelmäßigen Abständen an einen zentralen Web Service. Konsumenten fragen die Daten dann über einen anderen zentralen Web Service ab. So wird auf mobilen Geräten nur Web-Service-Client-Funktionalität benötigt.

[3] Anmerkung: Die Web Services müssen geeignet registriert werden, so dass sie entdeckt und benutzt werden können. Insbesondere im Falle von mobilen Web Services ist darauf zu achten, dass sich die logische Adresse, unter der das mobile Gerät erreicht werden kann (z. B. eine IP-Adresse), mit der Bewegung des Gerätes ändern kann. Es muss entweder dafür gesorgt werden, dass die Registrierung des Web Services aktuell gehalten wird, oder dass des Gerät transparent immer über die gleiche Adresse oder den gleichen Namen angesprochen werden kann. In dieser Arbeit wird vorausgesetzt, dass sämtliche Web Service Clients sich auf eine geeignete Art und Weise die notwendigen Informationen beschaffen, um den richtigen Web Service aufrufen zu können.

2. Peer-to-Peer: Das mobile Gerät bietet selber einen Web Service an, über den die entsprechenden Daten abgerufen werden können.

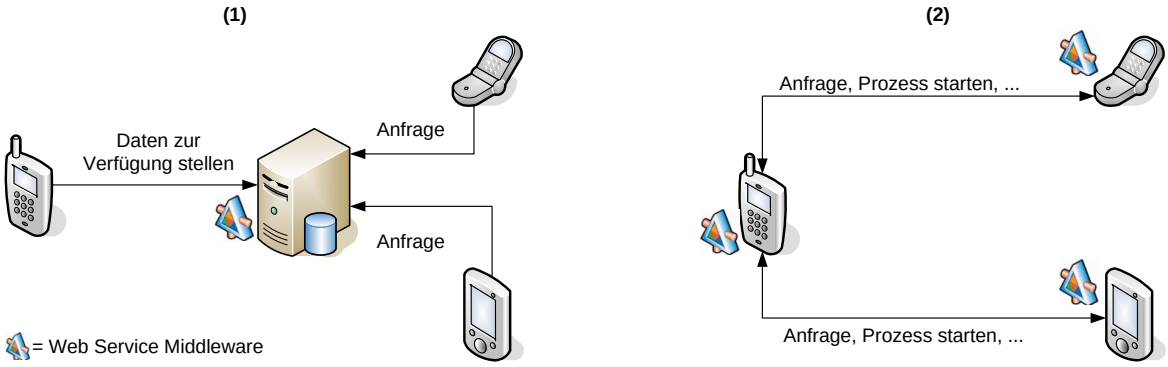

Abbildung 2.3.: Stationäre vs. mobile Web Services (eigene Darstellung)

Möglichkeit 1 ist dabei vor allem für Daten interessant, die in kurzen Abständen aktualisiert werden müssen und in kurzen Abständen von zahlreichen anderen Stellen aus abgefragt werden. Dies könnten z. B. Sensordaten (Position, Geschwindigkeit, Temperatur, Herzfrequenz, etc.) sein. Es macht dann Sinn, die Daten auf einem zentralen Server vorzuhalten, der viele Anfragen in kurzer Zeit ohne Verzögerungen und mit wenig Datenverlust verarbeiten kann.

Möglichkeit 2 ist interessant für Daten, die gezielt angefragt werden und nicht sehr oft benötigt werden, wie z. B. Multimedia-Daten (Töne, Bilder, kurze Videos). Weiterhin können so auch Teile von Geschäftsprozessen auf dem mobilen Gerät angestoßen werden, wie z. B. eine Anwendung, die den Benutzer auffordert, bestimmte Daten einzugeben und diese Daten dann an einen anderen Dienst zurück- bzw. weiterleitet (Push-Funktionalität). Es ist eine Kommunikation ohne weitere Infrastruktur wie z. B. einen zentralen Server (Proxy) möglich.

Kategorien von Szenarien für Möglichkeit 2 umfassen:

- Location Based Services: Abhängig von seiner Position bietet ein mobiles Gerät bestimmte Dienste an. Zum Beispiel könnten verschiedene ortsbezogene Daten von anderen Diensten bezogen, ausgewertet, aufbereitet und die aufbereiteten Daten dann von einem Dienst angeboten werden. Darunter fallen kollaborative Umgebungen, wie z. B. Meetings, auf denen den Teilnehmern von den im Raum befindlichen (mobilen) Geräten verschiedene Dienste angeboten werden, die das Meeting unterstützen können.
- Elektronisches Bezahlen: Ein auf dem mobilen Gerät laufender Dienst verwaltet ein virtuelles Konto, welches beim Bezahlen von der verkaufenden Stelle abgefragt und

belastet werden kann, sowie von einer Anwendung eines Kreditinstituts aufgeladen und anderweitig verwaltet werden kann (ähnlich der Geldkarte).

- Teilnahme an Workflows: Der Benutzer eines Mobiltelefons oder PDAs kann aktiv in Workflows eingebunden werden, wenn das Gerät entsprechende Dienste anbietet, wie z. B. Aufforderung zur Dateneingabe, Aufforderung zur Freigabe oder Ablehnung von Vorgängen.

Das in 1.2 erwähnte Beispiel des Freigabemanagers fällt unter die Kategorie „Teilnahme an Workflows“. Es soll wie in Abb. 2.4 veranschaulicht für den Rest dieser Arbeit als Beispiel dienen.

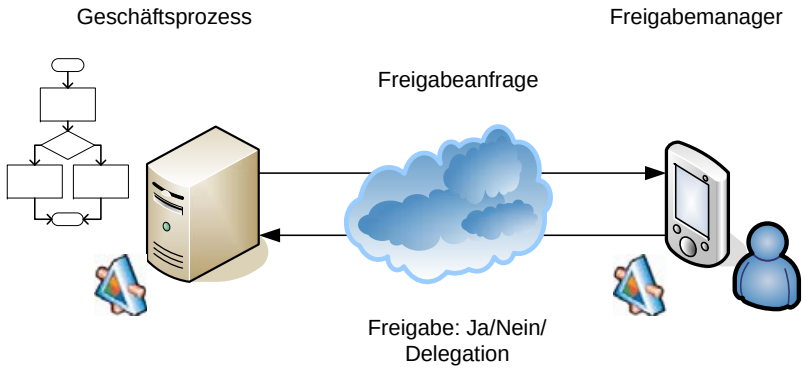

Abbildung 2.4.: Beispiel-Szenario Freigabemanager (eigene Darstellung)

2.4. Frameworks und Entwurfsmuster

Im Rahmen dieser Arbeit wird ein objektorientiertes Applikations-Framework erweitert. Laut (Gamma u. a., 1995) ist ein (objektorientiertes) Framework eine Menge von kooperierenden Klassen für eine spezifische Klasse von Software. Die Architektur bzw. der Entwurf eines Systems wird durch abstrakte Klassen mit Verantwortlichkeiten, Abhängigkeiten und Kollaborationen bis zu einem gewissen Grade vorgegeben. In (Schrörs, 2005) und (Schrörs, 2006a) wurde ein Framework für Web Services auf kleinen mobilen Geräten entworfen und umgesetzt (siehe 3.5). Dabei wird die Kommunikation durch generische Framework-Komponenten abgehandelt, so dass sich der Web-Service-Entwickler (fast) nur um die Anwendungslogik kümmern muss.

Auch das Management von langlebigen Transaktionen für Web Services sollte zwecks Wiederverwendbarkeit in Form eines Frameworks realisiert werden, zum Einen, weil dann das bereits vorhandene Framework erweitert werden kann, zum Anderen, weil Transaktionsmanagement sowohl generische als auch anwendungsspezifische Komponenten beinhaltet:

Zum Beispiel wird die Entscheidung über das Gesamtergebnis einer Transaktion von der Anwendungslogik getroffen, sofern kein technischer Fehler aufgetreten ist. Auch wenn alle Teilschritte erfolgreich durchgeführt wurden, kann die Auswertung der gesammelten Ergebnisse dazu führen, dass die Anwendungslogik die Transaktion rückgängig macht (durch Kompensation, s. 2.2). Es muss der Anwendungslogik daher möglich sein, an bestimmten Stellen in den Ablauf der Transaktion einzugreifen.

Nach (Jacobsen u. a., 1997) ist die Entwicklung von Frameworks aufgrund ihrer abstrakten und allgemeinen Natur komplizierter und anspruchsvoller als die Entwicklung von Anwendungen für spezielle Aufgaben. Durch die Benutzung von Entwurfsmustern (nach (Gamma u. a., 1995) eine generische Idee zum Entwurf von Lösungen für allgemeine, grundlegende Probleme) kann die Entwicklung von Frameworks und Anwendungen entscheidend erleichtert werden.

In Abschnitt 3.6 wird noch einmal näher auf Frameworks und Entwurfsmuster eingegangen. Im Rahmen des Entwurfs (Kap. 5) wird beschrieben, wie die Komponenten des Frameworks unter Benutzung bestimmter Entwurfsmuster realisiert werden können, so dass die in der Analyse (Kap. 4) erarbeiteten Anforderungen erfüllt werden.

2.5. Fazit

Web Services sind eine weit verbreitete Möglichkeit zur Integration heterogener Anwendungen in dienstorientierten Architekturen. Da immer mehr mobile, unter Umständen stark ressourcenbegrenzte Geräte im Einsatz sind, ist es wünschenswert, Web Services auch auf diesen laufen lassen zu können.

Da das Einsatzgebiet von mobilen Web Services auch Anwendungen mit Zugriff auf kritische Daten umfasst, die unter allen Umständen konsistent zu halten sind, ist es weiterhin wünschenswert, dass die mobilen Web Services an Transaktionen teilnehmen können, wie in 1.1 bereits angedeutet. Langlebige Transaktionen sind hier zunächst geeignet, da sie weniger Ansprüche an die Systeme stellen als ACID-Transaktionen und hauptsächlich auf Koordination und Kompensation beruhen.

Das mit Hilfe von Entwurfsmustern zu entwickelnde Framework soll anwendungsunabhängige Komponenten für die Erstellung transaktionaler Web Services zur Verfügung stellen.

3. Grundlagen

In diesem Kapitel werden für das Verständnis und die Realisierung des Transaktions-Frameworks für mobile Web Services notwendige Grundlagen aufbauend auf der vorangehenden Motivation erarbeitet. Es werden weiterhin Literaturhinweise sowie Hinweise auf existierende Arbeiten in den berührten Themenfeldern gegeben.

3.1. Dienstorientierte Architekturen und Web Services

Dienstorientierte Architekturen (SOAs) stellen wie in 2.1 erwähnt Lösungsansätze für EAI-Aufgaben dar, zu denen auch Workflow-Management gehört. Wie auch zum Begriff Softwarearchitektur (s. Fußnote auf S. 5) gibt es zum Begriff SOA verschiedene, mehr oder weniger klare Definitionen. Für diese Arbeit ist die W3C-Definition (WS-Arch, 2004, 3.1) ausreichend, in der SOA als eine Form der Architektur verteilter Systeme mit den folgenden Eigenschaften dargestellt wird:

- Dienste als logische, abstrakte, funktionsorientierte Sicht auf konkrete Komponenten
- Nachrichtenorientierung
- Beschreibung der relevanten Details von Diensten durch maschinenlesbare Metadaten
- Grobe Granularität sowohl was Funktionalität als auch Nachrichten betrifft
- Netzwerkorientierung
- Plattformunabhängigkeit

Andere Quellen definieren SOA nicht explizit, sondern verweisen auf die genauen Ausführungen der zugrunde liegenden Konzepte (z. B. (Weerawarana u. a., 2005, Kap. 1), (Krafzig u. a., 2005, Kap. 4), (Hohendahl, 2005, 2.2)) oder argumentieren ausgehend von den Eigenschaften von Web Services (Leymann, 2003, 2.4).

Zu den zentralen Bestandteilen von SOAs gehören nach (Krafzig u. a., 2005, Kap. 4) neben den Diensten, welche die Geschäftsoperationen zur Verfügung stellen, das „Service Repository“ (zur Erleichterung der Beschreibung, des Auffindens und schließlich der Wiederverwendung von Diensten), der „Enterprise Service Bus“ (ESB, ein Framework, das als

Messaging-basierte Infrastruktur einer SOA zur Erleichterung des Einbindens von Diensten beiträgt) sowie die Benutzerschnittstellen.

Service Repository, ESB und Benutzerschnittstellen spielen für diese Arbeit keine besondere Rolle, da nur die transaktionale Absicherung von Operationen, die mehrere Aufrufe von Diensten umfassen, behandelt wird.

Dienste werden nach (Krafzig u. a., 2005, Kap. 5) und (Erl, 2005, Kap. 9) in unterschiedliche Kategorien eingeteilt, z. B.:

- Basis- (auch: einfache, Applikations-) Dienste bieten einfache, atomare Operationen an, sind zustandslos und hochgradig wiederverwendbar. Basis-Dienste können datenzentrisch oder logikzentrisch sein.
- Zusammengesetzte (auch: Geschäfts-) Dienste bieten Operationen bezüglich bestimmter Geschäftsabläufe bzw. Geschäftsentitäten (z. B. „Konto“, „Kunde“) an und sind dadurch nicht sehr stark wiederverwendbar. Auch sie sind zustandslos. Zusammengesetzte Dienste mit technischem Fokus (z. B. Technologie-Gateways) werden auch vermittelnde Dienste genannt. Zusammengesetze Dienste rufen im Rahmen ihrer Anwendungslogik andere zusammengesetzte und einfache Dienste auf.
- Prozesszentrische (auch Orchestrierungs-) Dienste bilden Geschäftsprozesse ab. Sie sind auch zusammengesetzt, sind jedoch als einzige Dienste zur Ausführungszeit zustandsbehaftet. Auch sie sind nicht stark wiederverwendbar.

Diese Kategorien spiegeln auch Schichten unterschiedlichen Abstraktionsgrades wieder.

Im Rahmen dieser Arbeit werden zustandslose sowie zustandsbehaftete zusammengesetzte Dienste betrachtet, die Workflows realisieren und deren Operationen inklusive Aufrufe anderer Dienste transaktional abgesichert werden müssen. Es muss dabei, wie eingangs erwähnt, beachtet werden, dass Operationen, die im Rahmen der Anwendungslogik aufgerufen werden, lange Laufzeiten haben können. In 3.3 werden die Grundlagen dazu behandelt.

„Transacted Messages“ im Zusammenhang mit der Kommunikation von Services mit einem ESB, also Senden und Empfangen mehrerer Nachrichten als Transaktion unter Einbeziehung weiterer an der Transaktion teilnehmender Ressourcen wie z. B. Datenbanken (Chappell, 2004, Kap. 5), werden als lokale Transaktionen betrachtet, die innerhalb von langlebigen Transaktionen für kurzlebige Vorgänge verwendet werden können. In dieser Arbeit spielen ESB-Konzepte daher keine Rolle, sondern ein vorhandener ESB wird im Falle langlebiger Transaktionen wie ein üblicher Web Service behandelt und muss dementsprechend operieren.

Web Services sind die am häufigsten gewählte Implementierung von Diensten. Aufbauend auf dem Basis-Framework für Web Services (XML, XML-Schema, SOAP, WSDL, UDDI) existiert eine Vielzahl von Spezifikationen (WS-* Erweiterungen), die sich an Schichtenarchitekturen wie in (Weerawarana u. a., 2005, 1.4.2), in (WS-Arch, 2004, 3.1) und — wie in Abb. 3.1 gezeigt — in(Endrei u. a., 2004) beschrieben orientieren und festlegen, wie Web Services miteinander kommunizieren.

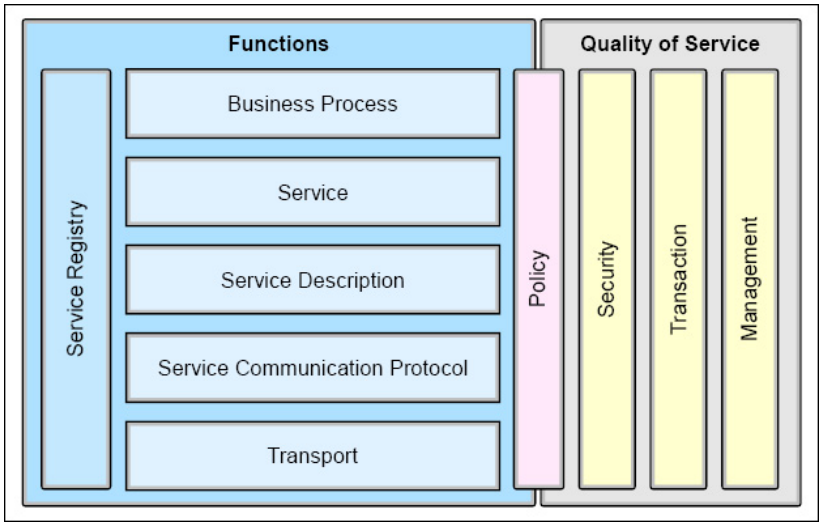

Abbildung 3.1.: Web Services Architektur Referenzmodell (Endrei u. a., 2004)

Besonders hervorzuheben ist hierbei, dass Web Services genau wie CORBA-Komponenten und EJBs (laut 2.1 sind Dienste ja konzeptionell mit Komponenten vergleichbar) eine Laufzeitumgebung benötigen, damit die vom Interface separat spezifizierten Eigenschaften, wie z. B. das Transaktionsverhalten als Dienstgüte-Eigenschaft, garantiert werden können. Auch die Kommunikation wird von der Laufzeitumgebung übernommen. Spezifikationen zum Transaktionsmanagement für Web Services werden in 3.4 vorgestellt.

Ohne Laufzeitumgebungen sind Komponenten bzw. Web Services nicht benutzbar. Komponenten-Frameworks wie CORBA (in verschiedenen Sprachausprägungen), Enterprise JavaBeans (Java) und .NET/COM (C++, C#) bieten Umgebungen, in denen Softwarekomponenten in so genannten Containern laufen („leben“), wobei die Container das Laufzeit- bzw. Lebenszyklusmanagement übernehmen und der Komponentenentwickler sich (fast) ausschließlich mit der Anwendungslogik befassen muss. Im Falle von Java Web Services kann als Container z. B. Axis (Axis, 2006) verwendet werden, das selbst als J2EE Komponente (nämlich als Servlet) in einem J2EE Web Container läuft. Aber auch EJBs können als Web Service aufgerufen werden (s. (EJB-2.1, 2003)).

Das in (Schrörs, 2005) und (Schrörs, 2006a) entwickelte Framework dient als Laufzeitumgebung für Web Services auf J2ME-basierten mobilen Geräten (s. 3.5). Dieses Framework wird in dieser Arbeit erweitert.

Für allgemeine Betrachtungen sowie detaillierte Beschreibungen von SOA und Web Services siehe (Krafzig u. a., 2005), (Weerawarana u. a., 2005) und zu Web Services (Zimmermann u. a., 2003/2005), zu ESB im speziellen auch (Krafzig u. a., 2005, Kap.9) und (Chappell, 2004). Beispiele für Anwendungen von SOAs in der Praxis finden sich in (Brandner u. a., 2004), (Zimmermann u. a., 2004) und (Zimmermann u. a., 2005).

3.2. Transaktions-Management in Komponentenumgebungen

In Abschnitt 2.1 wurde erarbeitet, dass Dienste konzeptionell Komponenten ähneln. Es wird daher angenommen, dass Transaktionsmanagement für Dienste ähnlich funktionieren kann wie für Komponenten. Im Folgenden werden Ansätze zum Management von Transaktionen in den Komponentenumgebungen (d. h. Frameworks und Laufzeitumgebungen) CORBA und J2EE kurz dargestellt, um dann daraus die wesentlichen Eigenschaften auf das Transaktionsmanagement für Web Services zu übertragen.

Die hier vorgestellten Verfahren, insbesondere das 2-Phase-Commit-Protokoll (2PC, s. u.), wurden allerdings nur für ACID-Transaktionen entwickelt. Vorgreifend sei hier schon erwähnt, dass grundlegende Eigenschaften, Verfahren und Entitäten auch beim Management von langlebigen Transaktionen zum Tragen kommen.

3.2.1. X/Open DTP

Das X/Open-Konsortium hat von Anfang bis Mitte der 1990er Jahre das Referenzmodell X/Open DTP zu verteiltem Transaktionsmanagement (engl. Distributed Transaction Processing) erstellt und weiterentwickelt, um eine Basis für die Interoperabilität zwischen verschiedenartigen an Transaktionen beteiligten Systemen zu definieren. Abb. 3.2 zeigt die wesentlichen beteiligten Systemkomponenten und Protokolle zur Abwicklung von globalen, flachen (d. h. nicht geschachtelten) Transaktionen. Systemkomponenten sind als schwarze Kästen dargestellt, die Protokolle als Pfeile mit Beschriftung. Die Protokollbefehle (Verben) sind in schwarz, die Protokollnamen darüber in hellerer Schrift wiedergegeben, mit der Quelle des Protokolls in eckigen Klammern, da hier auch ISO/OSI-Protokolle aufgenommen wurden. Das linke System (Applikation bzw. Client) kann dabei mit mehreren entfernten Systemen wie auf der rechten Seite dargestellt (Server) kommunizieren. Das Modell wird in (Gray und

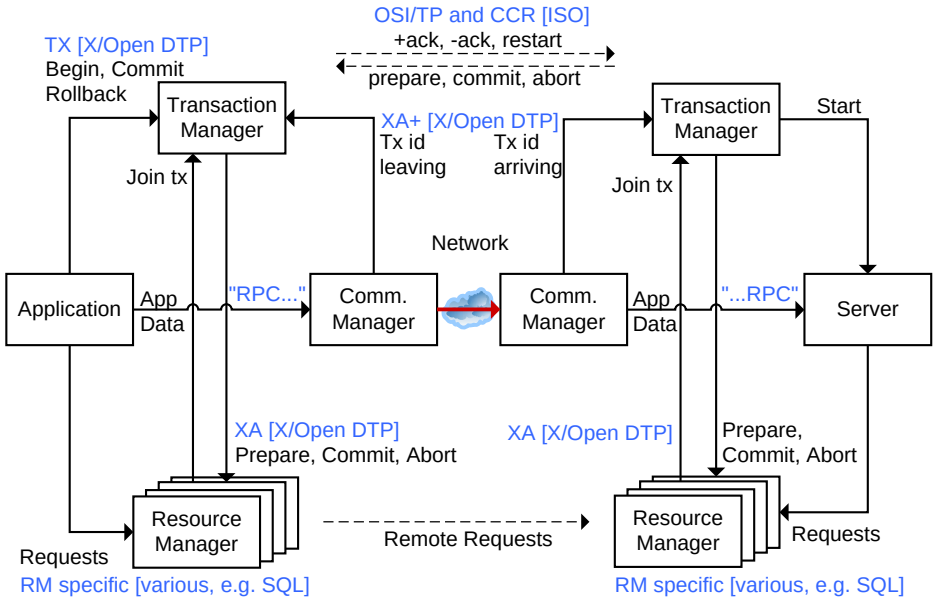

Abbildung 3.2.: X/Open DTP nach (Gray und Reuter, 1993)

Reuter, 1993) genauer beschrieben. An dieser Stelle reicht die Beschreibung, der wesentlichen beteiligten Systeme, deren Aufgaben, sowie der verwendeten Protokolle aus:

Die Applikation startet eine Transaktion durch Senden von `Begin` an den „Transaktionsmanager“. Alle folgenden Aufrufe lokaler oder entfernter „Ressourcenmanager“ (z. B. Datenbank- oder Messaging-Systeme) durch die Applikation werden um den „Transaktionskontext“ erweitert, welcher es den aufgerufen Systemen ermöglicht, sich beim Transaktionsmanager für die Transaktion zu registrieren, bevor sie die Aufrufe abarbeiten.

Wird ein entferntes System aufgerufen, so wird der Aufruf (z. B. RPC) über die „Kommunikationsmanager“ an eine Server-Komponente weitergeleitet. Benutzt der Server im Laufe der Abarbeitung des Aufrufs einen Ressourcenmanager, so registriert sich dieser über einen entfernten (d. h. für den Server und den Ressourcenmanager lokalen) Transaktionsmanager für die Transaktion. Der entfernte Transaktionsmanager leitet die Registrierung dann über die Kommunikationsmanager an den lokalen Transaktionsmanager der Applikation weiter. Ressourcenmanager können innerhalb der Kommunikation auch direkt miteinander kommunizieren (z. B. verteilte Datenbanksysteme), die Registrierung für die Transaktion läuft aber genauso wie für die Aufrufe durch die Applikation.

Soll die Transaktion beendet werden, so sendet die Applikation entweder `Commit` (erfolgreiche Terminierung mit Festschreiben aller Änderungen) oder `Rollback` (Rückgängigmachen aller Änderungen) an den Transaktionsmanager, welcher diese Entscheidung an die Ressourcenmanager weitergibt. Dabei können unterschiedliche „Terminierungsprotokol-

le“ zum Einsatz kommen. Das für atomare Transaktionen übliche Protokoll ist das 2-Phase-Commit-Protokoll (s. z. B. (Tanenbaum und van Steen, 2002, Kap. 7)). In der ersten Phase des Protokolls wird das Ergebnis der Verarbeitung von allen registrierten Ressourcenmanagern vom Transaktionsmanager eingefordert (`Prepare`), bevor in der zweiten Phase aufgrund des Gesamtergebnisses jeder Ressourcenmanager zum Festschreiben (`Commit`) oder Verwerfen (`Rollback`) der gemachten Änderungen aufgefordert wird.

In Abb. 3.2 werden der Vollständigkeit halber die benutzten Kommunikationsprotokolle genannt: „TX“ (Transaction) zwischen Applikation und Transaktionsmanager, „XA“ (Eigenname, keine Abkürzung) zwischen Transaktions- und Ressourcenmanager mit „XA+“-Erweiterung zur Einbeziehung des Kommunikationsmanagers, sowie „OSI/TP“ (Transaction Processing) und „OSI/CCR“ (Commit, Concurrency, Recovery) für die Kommunikation mit entfernten Transaktionsamanagern zur Weiterleitung der XA-Befehle an die entfernten Ressourcenmanager. Die Namen und technischen Details sind für diese Arbeit jedoch nicht von Bedeutung.

3.2.2. CORBA

Der CORBA Object Transaction Service (OTS) stellt innerhalb der Object Management Architecture (OMA) der Object Management Group (OMG) einen so genannten „CORBAservice“ dar (Siegel, 2000). Der OTS wurde von führenden Herstellern von Transaktionsverarbeitungs-Software entwickelt und spezifiziert das Transaktionsmanagement für verteilte CORBA-Objekte. Er ist eine Implementierung der X/Open DTP Spezifikation und bildet eine Dienstgüte-Schicht für die Kommunikation zwischen transaktionalen Objekten. Im Jahr 2000 gab es mehr als 12 Implementierungen des OTS (Siegel, 2000). In Zusammenarbeit mit dem Object Request Broker (ORB) stellen OTS-Implementierungen den Anwendungen und Ressourcenmanagern die zur Kontrolle von Transaktionen notwendigen Interfaces zur Verfügung, assoziieren gestartete oder durch Methodenaufrufe ins System importierte Transaktionen mit dem aktuell ausgeführten Thread und sorgen bei der Kommunikation dafür, dass der Transaktionskontext automatisch als impliziter Parameter an entfernte Methodenaufrufe (engl. Remote Method Invocation, RMI) angehängt und somit an entfernte Systeme/Objekte propagiert wird. Die in CORBA-IDL (Interface Definition Language) spezifizierten Interfaces stellen die für die jeweiligen an einer Transaktion beteiligten Objekte wesentliche Funktionalität der X/Open DTP TX- und XA(+)-Protokolle zur Verfügung. Die Objekte, die als Ressourcenmanager fungieren, müssen den dafür relevanten Teil des XA-Protokolls als Interface anbieten.

3.2.3. J2EE

Der Java Transaction Service (JTS) der J2EE-Spezifikation ist eine Implementierung des OTS in Version 1.1 für die Sprache Java 2. Für die entfernten Methodenaufrufe wird dabei wie bei CORBA das Internet Inter-ORB Protocol (IIOP) benutzt (Java RMI over IIOP), was dazu führt, dass J2EE Software-Komponenten (also Enterprise JavaBeans, s. u.) zur Laufzeit theoretisch mit den so genannten „CORBA 3 Basic Components“ (s. (Siegel, 2000)) interoperabel sind. Eine Implementierung des JTS ist in den meisten J2EE Application Servern enthalten. Der JTS stellt den transaktionalen J2EE Subsystemen das Java Transaction Interface JTA (JTA, 2002) zur Verfügung. Die zum JTA gehörenden Java Interfaces ähneln dabei stark den OTS Interfaces. Folgende J2EE Subsysteme sind an Transaktionen beteiligt:

- Der Enterprise Java Bean Container sowie die darin installierten Enterprise JavaBeans (EJBs). EJBs können Transaktionen entweder selbst steuern (über das Interface `UserTransaction`, welches ein Subset des TX-Protokolls zur Verfügung stellt) oder dies vom Container erledigen lassen, wobei für jede Methode im EJB-Deployment-Descriptor angegeben wird, wie verfahren werden soll. Zum Beispiel kann spezifiziert werden, dass eine Methode nicht transaktional ist (`NotSupported`), was dazu führt, dass ein eventuell mit dem Aufruf empfangener Transaktionskontext ignoriert wird. Oder es kann spezifiziert werden, dass eine Methode immer transaktional sein muss (`Required`), was dazu führt, dass eine neue Transaktion gestartet wird, wenn kein Transaktionskontext mit dem Aufruf empfangen wurde. Dem Container steht dabei über die Interfaces `TransactionManager` und `Transaction` das volle TX-Protokoll zur Verfügung. Für Details zu EJB-Transaktionen siehe (EJB-2.1, 2003).
- Java Database Connection (JDBC) Provider und Java Message Service (JMS) Provider (z. B. Message Queues) spielen die Rolle von Ressourcenmanagern und müssen das `XAResource` Interface implementieren, um vom Application Server an Transaktionen beteiligt werden zu können.

Eine detaillierte Beschreibung des JTS und dessen Verwandtschaft zum OTS ist in (Little u. a., 2004) zu finden.

Mit dem JSR (Java Specification Request[4]) 95, „J2EE Activity Service for Extended Transactions“ (JSR-95, 2006) wurde auch ein JTA-ähnliches API für langlebige Transaktionen in J2EE spezifiziert. Dies betrifft J2EE Komponenten, aber nicht Web Services. Eine Beschreibung findet sich in (Little u. a., 2004, Kap. 9).

[4] JSRs werden im Rahmen des Java Community Process zur Entwicklung von Java-Spezifikationen erstellt, s. http://jcp.org/en/home/index.

3.2.4. Rollen und Entitäten

Die wichtigsten, in den verschiedenen Entwicklungsstufen und Ausprägungen des Transaktionsmanagements immer wiederkehrenden Rollen bzw. Entitäten sind Applikation, Transaktionsmanager, Ressourcenmanager, sowie Transaktionskontext.

Durch entsprechende Aufrufe des Transaktionsmanagers (OTS, JTS) startet die Applikation die Transaktion, koordiniert deren Verlauf und beendet sie schließlich mit oder ohne Erfolg. Im Folgenden wird für steuernde Applikation und Transaktionsmanager auch das Wort „Koordinator“ benutzt. Im Freigabemanager-Beispiel ist der Web Service, der den Ablauf Geschäftsprozesses zentral steuert, der Koordinator.

Ressourcenmanager (CORBA-Objekte, JDBC Connections, JMS Provider) sind an der Transaktion teilnehmende Systeme und Anwendungen. Die von ihnen verwalteten Ressourcen (z. B. Datenbanktabellen oder Message Queues) werden durch die Transaktion geschützt bzw. konsistent gehalten. Im weiteren Verlauf wird für Ressourcenmanager auch das Wort „Teilnehmer“ benutzt. Im Freigabemanager-Beispiel ist der mobile Web Service ein Teilnehmer. Es können beliebig viele weitere Teilnehmer vom Koordinator und von bereits registrierten Teilnehmern aufgerufen werden.

Der Transaktionskontext ist eine Datenstruktur, die mit den (entfernten) Aufrufen der beteiligten Anwendungen implizit oder explizit mitgeschickt wird. Anhand des Transaktionskontexts, der im Weiteren auch „Koordinationskontext“ genannt wird, können die Anwendungen eingehende Aufrufe einer bestimmten Transaktion zuordnen. Der Transaktionskontext kann neben einer eindeutigen Kennzeichnung der Transaktion auch weitere Informationen enthalten, z. B. Transaktionstyp oder Adresse für die Registrierung von teilnehmenden Ressourcenmanagern.

3.2.5. Protokolle und Zustandsautomaten

Die Erzeugung eines Transaktionskontextes, die Registrierung von Ressourcenmanagern beim Koordinator sowie das Beenden der Transaktion durch ein entsprechendes Terminierungsprotokoll (bei ACID-Transaktionen z. B. 2-Phase-Commit) erfordern entsprechende Protokolle, die die beteiligten Anwendungen benutzen müssen.

Im Weiteren werden Protokolle für „Aktivierung“ (Start der Transaktion, Erzeugen des Transaktionskontexts), „Registrierung“ (der Teilnehmer beim Koordinator) und „Terminierung“ (Beenden der Transaktion) unterschieden.

Um unnötige Komplexität zu vermeiden, läuft dabei die Kommunikation immer zwischen einem Teilnehmer und dem Koordinator ab und nicht zwischen zwei Teilnehmern. Mit dem Senden oder dem Empfangen einer Protokollnachricht durch eine Anwendung kann sich

der Zustand einer Anwendung bezüglich der Transaktion und dem betroffenen Koordinator-Teilnehmer-Paar ändern. Dieser Zustand muss für die Dauer einer Transaktion von den einzelnen Anwendungen verwaltet werden. Der Koordinator verwaltet dabei den Zustand für jedes Koordinator-Teilnehmer-Paar.

Die Anwendungen müssen sich also wie Zustandsautomaten verhalten, die beim Empfang einer Protokollnachricht eine Aktion durchführen, ggf. eine Antwort-Nachricht (synchron oder asynchron) zurückschicken, und in einen neuen Zustand übergehen.

Im Abschnitt über Web Service Transaktionen (s. 3.4) werden derartige Protokolle näher erläutert.

3.3. Workflow-Management mit langlebigen Transaktionen und Web Services

3.3.1. Workflows und langlebige Transaktionen

In 2.2 ist von „lange“ dauernden Vorgängen die Rede. „Lange“ bezeichnet dabei einen Zeitraum, der über die übliche Dauer einer klassischen ACID-Transaktion mit 2-Phase-Commit-Protokoll von einigen Millisekunden bis maximal einigen Sekunden (Little, 2003a) hinausgeht. Das für die Einhaltung der ACID-Eigenschaften notwendige Sperren von Ressourcen über die gesamte Dauer der Transaktion würde in diesem Fall die nicht-funktionalen Eigenschaften (z. B. Antwortzeiten) eines Systems oder Teilsystems derart verschlechtern, dass das System oder Teilsystem nicht mehr benutzbar im Sinne von Endanwendern wäre oder bei nicht-interaktiven Vorgängen Fehler produzieren würde. Das 2-Phase-Commit-Protokoll birgt nach (Krafzig u. a., 2005, 8.2.4) weiterhin die Probleme, dass viele Legacy-Systeme es schlichtweg nicht beherrschen und dass es nicht für diskontinuierliche Netzwerke (z. B. Funknetzwerke mit Empfangsunterbrechungen) geeignet ist. Es ist nach (Papazoglou, 2003) nicht für asynchron kommunizierende, lose gekoppelte Systeme geeignet.

Insbesondere wenn ein Vorgang manuelle Schritte beinhaltet (z. B. eine Freigabe einer Bestellung durch einen entsprechend berechtigten Benutzer im Rahmen des Freigabemanager-Beispiels aus 1.2), kann die Ausführung eines Geschäftsprozesses beliebig lange dauern. Auch langwierige Berechnungen können Vorgänge innerhalb von Geschäftsprozessen verlängern.

Workflow-Management-Systeme (WFMS) zur IT-seitigen Unterstützung und Durchführung von Geschäftsprozessen werden seit Anfang der 90er Jahre eingesetzt. Die Repräsentation eines Geschäftsprozesses wird dabei als Workflow bezeichnet. Workflows werden in

speziellen Workflow-Sprachen meist mit Tool-Unterstützung programmiert. Ein Workflow besteht aus einzelnen daten- und flussunabhängigen Tasks, die durch die Programmierung in eine bestimmte Reihenfolge (Fluss) gebracht und mit den benötigten Datenquellen und -senken verbunden werden (Leymann und Roller, 1997). Workflow-Management-Systeme sorgen dafür, dass die Workflows wie spezifiziert ausgeführt werden, so dass die Eigenschaften der korrespondierenden Geschäftsprozesse (z. B. Optimalität bzgl. Nutzung bestimmter Ressourcen) gewahrt bleiben. An einem Workflow sind können diverse heterogene Systeme und Applikationen beteiligt sein.

Um innerhalb eines Workflows unnötig lange Sperrungen von gemeinsam genutzten Ressourcen wie z. B. Datenbanken-Tabellen oder Message-Queues zu vermeiden, sollten für lange andauernde Vorgänge keine übergreifenden verteilten ACID-Transaktionen (z. B. Objekttransaktionen wie in 3.2 beschrieben) verwendet werden. Verzichtet man auf Sperren, so werden Zwischenergebnisse vor Ende der langlebigen Transaktion nach außen hin sichtbar. Die gesamte langlebige Transaktion erscheint daher für andere Transaktionen nicht mehr atomar und isoliert. Daten können so zwar unter Umständen vorübergehend inkonsistent sein, sollten aber sowohl nach erfolgreicher Durchführung als auch nach einem vorzeitigen Abbruch einer langlebigen Transaktion konsistent sein. Um dies zu erreichen, ist es notwendig, dass die an einer langlebigen Transaktion beteiligten Applikationen im Zeitraum zwischen Start und Ende des Gesamtvorgangs ihre bisher ausgeführten Schritte rückgängig machen können.

Das Rückgängigmachen bereits abgeschlossener Vorgänge wird als Kompensation bezeichnet (Leymann und Roller, 1997). Im Speziellen kann damit z. B. die Stornierung einer Buchung gemeint sein (s. 2.2). Im Falle des Freigabemanager-Beispiels beträfe dies das zurückziehen einer Freigabeanfrage, nachdem der Freigabemanager die Freigabe erteilt oder abgelehnt hat. In diesem Fall müssen sowohl der Freigabemanager, als auch von der Freigabe abhängige Applikationen, die das Ergebnis der Freigabeanfrage bereits gesehen haben, informiert werden (solche Applikationen sind im Beispiel nicht näher beschrieben, es könnte sich aber beispielsweise um eine Anwendung zur automatischen Buchung von Reisen handeln, die dann bereits gebuchte Reisen stornieren müsste). Im Gegensatz dazu ist der Vorgang beim Rollback bei ACID-Transaktionen nicht abgeschlossen und die Zwischenergebnisse der Bearbeitung sind noch nicht außerhalb der Transaktion sichtbar.

Um im Fehlerfall geeignet kompensieren zu können, ist es notwendig, dass für jede langlebige Transaktion eine Instanz existiert, die über alle beteiligten Dienste und deren Ausführungszustand informiert ist und die im Fehlerfall allen bereits ausgeführten Dienste den Befehl zum Kompensieren geben kann. Eine solche Instanz wird als Koordinator bezeichnet. Wie in 3.2.4 bereits angedeutet, sind Workflow-Implementierungen Kandidaten für die Rolle des Koordinators.

Innerhalb von Workflows können von kurzlebigen Teilvorgängen verteilte ACID-Transaktionen wie üblich benutzt werden. Die Ergebnisse der Berechnung werden allerdings vor dem Ende des gesamten Workflows sichtbar.

Da Atomizität und Isolation von langlebigen Transaktionen nicht gewährleistet sind, besteht das Risiko, dass andere, parallel ausgeführte Transaktionen Zwischenstände von Daten lesen, die nach erfolgreichem Durchlauf oder Abbruch der ändernden Transaktion so nicht mehr gültig sind. Es ist daher beim Design der einzelnen Dienste der Wahl der Transaktionsgrenzen und der Benutzung von lokalen ACID-Transaktionen darauf zu achten, dass dieses Risiko minimiert wird.

Es wurden bereits verschiedene Konzepte zur Konsistenzsicherung mit langlebigen Transaktionen entwickelt. Eine Zusammenfassung findet sich in (Limthanmaphon und Zhang, 2004). Ein verbreitetes Prinzip wird an den Modellen „Transaction Chains“ und „Sagas“ deutlich:

Das Modell der Transaction Chains (s. z. B. (Krafzig u. a., 2005, 8.2.6, 8.2.7)) beschreibt die Zerlegung von komplexen, langlebigen Vorgängen in verkettete transaktionale Schritte mit kurzer Dauer und Benutzung von Kompensationstransaktionen, um einzelne Schritte rückgängig zu machen.

Sagas (Garcia-Molina und Salem, 1987), beschrieben auch in (Krafzig u. a., 2005, 8.2.8), sind formale Workflow-Modelle, welche auf Transaction Chains aufbauen und im Fehlerfall Kompensations-Transaktionen für alle bereits abgeschlossenen Schritte in umgekehrter Reihenfolge ausführen. Diese formalen Modelle bergen jedoch das Problem der nicht handhabbaren Komplexität der Workflow- und Kompensationsgraphen, so dass sie nicht generisch umsetzbar sind.

3.3.2. Workflows und Web Services

Web Services eignen sich aufgrund ihrer in 2.1 genannten Eigenschaften gut für die Implementierung von Arbeitsschritten innerhalb von flexiblen, leicht änder- und anpassbaren Workflows. Wenn Geschäftsprozesse im Rahmen von Outsourcing, Optimierung oder Unternehmenszusammenschlüssen angepasst werden müssen, ist dies besonders wichtig (siehe (Weerawarana u. a., 2005, 1.1) und (Leymann u. a., 2002)).

Web Services Orchestration (Erl, 2005, 6.6) verbindet verschiedene existierende Prozesse über Workflow-Logik ohne die Notwendigkeit diese neu zu entwickeln. Web Services nehmen an Orchestrierungen als „Prozessteilnehmer“ teil, ohne dass sie über den Gesamtprozess informiert sein müssen. Als Modellierungs- bzw. Programmiersprache für Web Services Orchestration hat sich die Flow-basierte Business Process Execution Language for Web Services — „BPEL4WS“ oder „WS-BPEL“ (WS-BPEL, 2003) — gegenüber anderen Ansätzen durchgesetzt (Hohendahl, 2005, 3.1).

Mit WS-BPEL geschriebene Orchestrierungen werden von einem Compiler in lauffähige Komponenten übersetzt, welche wiederum (Java) Web Services sein können. Damit sind verschachtelte Workflows möglich. WS-BPEL bietet Kompensationsmechanismen im Falle von Exceptions, allerdings wirken diese nicht verteilt, d. h. teilnehmende (aufgerufene) Web Services werden nicht implizit miteinbezogen. Automatische, standardisierte Koordinations- und Transaktionsmechanismen fehlen also und müssen durch andere Mechanismen ergänzt werden (Papazoglou, 2003).

Nach (Erl, 2005, Kap. 6) hängen Web-Service-Orchestrierung sowie langlebige- und ACID-Transaktionen im Rahmen von dienstorientierten Architekturen wie folgt zusammen: Web Services nehmen im Rahmen von Workflows als „Prozessteilnehmer“ an Orchestrierungen teil. Orchestrierungen benutzen Protokolle, die für „Business Activities“ definiert werden. Das Konzept der Web Service Orchestration trägt zur Erhöhung der Flexibilität des Workflow-Managements bei (Erl, 2005, 6.6.7).

„Business Activities“ verwalten langlebige komplexe Aktivitäten und stellen Kompensationsfähigkeiten für Web Services zur Verfügung. Sie können außerdem „Atomic Transactions“ enthalten. Die Kommunikation zwischen den verschiedenen Teilnehmern wird dabei durch Protokolle reglementiert, wodurch Eigenschaften wie Zusammensetzbarkeit und Interoperabilität betont werden. Das Konzept der Kompensation als Fehlertoleranzmechanismus trägt zur Erhöhung der Dienstgüte bei (Erl, 2005, 6.5.5).

„Atomic Transactions“ stellen verteiltes ACID-Transaktionsmanagement für Web Services zur Verfügung, wobei durch Spezifikation von speziellen Nachrichten und Nachrichtenteilen der Scope des 2-Phase-Commit-Protokolls über übliche Applikationsgrenzen hinaus ausgeweitet wird. Ebenso wie Business Activities erhöhen Standards bzw. Reglementierungen für Atomic Transactions Zusammensetzbarkeit, Interoperabilität und Dienstgüte innerhalb von Web-Service-Kollaborationen (Erl, 2005, 6.4.5).

Business Activities und Atomic Transactions sind Spezialfälle der Koordination komplexer Aktivitäten. Koordination bildet eine Kontrollschicht für die Komposition (Zusammensetzung) von Web Services und standardisiert den Austausch von Kontextinformationen (Erl, 2005, 6.3.6). Komplexe Aktivitäten (Erl, 2005, 6.2) organisieren (im Sinne von Aufrufen) zahlreiche Web Services und bestehen wiederum aus einfachen Aktivitäten, die wenige Web Services organisieren und Kommunikationsmuster (Erl, 2005, 6.1) für diese festlegen.

In dieser Arbeit sind nur die Komponenten des Transaktionsmanagements von Interesse, wobei der Fokus auf langlebigen Transaktionen (Business Activities) liegt. Die Begriffe stehen hier für beliebige derartige Spezifikationen für Web Services. Im folgenden Abschnitt werden drei existierende Spezifikationen kurz erklärt. Aus den vorgestellten Spezifikationen wird schließlich ein Satz für die Realisierung des Transaktionsframeworks für mobile Web Services ausgewählt.

3.4. Spezifikationen für Web Service Transaktionen

Es existieren mehrere Spezifikationen für Web-Service-Transaktionen. Die wichtigsten drei sind das Business Transaction Protocol (BTP), Web Service Coordination (WS-Coordination) sowie das Web Service Composite Application Framework (WS-CAF). Diese werden in (Little, 2003a), (Little, 2003b) und (Little u. a., 2004, Kap. 10) erklärt und verglichen.

Alle drei Spezifikationen werden von der Organization for the Advancement of Structured Information Standards (OASIS, http://www.oasis-open.org) betreut und haben gemeinsam, dass sie neben Koordinationsmechanismen sowohl Protokolle für atomare Transaktionen, sofern alle Teilnehmer dies auch unterstützen, als auch Protokolle für langlebige Transaktionen bzw. Aktivitäten spezifizieren. Weiterhin ist allen Spezifikationen gemeinsam, dass sie keinerlei sprachgebundene Interfaces festlegen, sondern lediglich die Syntax der XML-basierten Protokollnachrichten sowie die Semantik der unterstützten Protokolle. Es wird dabei insbesondere die Flexibilität und Erweiterbarkeit des Simple Object Access Protocols SOAP (SOAP, 2003) ausgenutzt, um die Protokollnachrichten zu transportieren.

3.4.1. Business Transaction Protocol

Beim BTP (BTP, 2004) handelt sich um ein Koordinator-basiertes Transaktionsprotokoll wie 2-Phase-Commit, jedoch mit wesentlich mehr Freiheiten für die koordinierende Applikation. Es wurde mit dem Ziel entwickelt, Zuverlässigkeit über unzuverlässige Kanäle (z. B. das Internet) zu erreichen und dabei explizit langlebige Transaktionen zu unterstützen. Aufgrund seiner Flexibilität eignet es sich auch für Workflow-Steuerungen. Durch das BTP hat der Koordinator die Kontrolle darüber, wann eine Transaktion für ein `COMMIT` vorbereitet wird (`PREPARE`). Mittels Business-Logik kann dann für jeden Teilnehmer (Service) entschieden werden, ob die dortige Transaktion bestätigt (`CONFIRM`) oder abgebrochen (`CANCEL`) wird. Anschließend kann dann auf Basis der von jedem Teilnehmer erhaltenen Antworten (`CONFIRM` oder `CANCEL`) entschieden werden, ob die Gesamttransaktion erfolgreich (`COMMIT`) oder nicht erfolgreich (`ROLLBACK`) beendet wird.

Im BTP gibt es zwei Arten von Transaktionen:

Atom: Atoms verhalten sich wie klassische ACID-Transaktionen. Sie sind nur dann erfolgreich, wenn alle Teilnehmer auch erfolgreich sind. Ergebnisse der Teilnehmer werden erst sichtbar, wenn die Gesamt-Transaktion erfolgreich abgeschlossen wird. Die Ablaufsteuerung entspricht dem 2-Phase-Commit-Protokoll.

Cohesion: Eine Cohesion entspricht einer langlebigen Aktivität und kann aus Unteraktivitäten und Atoms zusammengesetzt sein. In einer Cohesion können die Teilnehmer unterschiedliche Ergebnisse haben und trotzdem kann die Gesamttransaktion erfolgreich abschließen. Das 2-Phasen-Protokoll, das zur Anwendung kommt, erlaubt eine genaue Parametrisierung dafür, welche Teilnehmer für ein `COMMIT` bestätigt (`CONFIRM`) oder abgebrochen (`CANCEL`) werden. Das Ergebnis eines Teilnehmers wird bereits vor dem Gesamt-`COMMIT` außerhalb der Cohesion sichtbar. Schlägt die Cohesion fehl (`ROLLBACK`), so müssen alle Teilnehmer, die ein `CONFIRM` erhalten haben, im Protokoll zu definierende kompensierende Aktionen ausführen.

Beispiele für die Anwendung des BTP finden sich in (Little, 2003a) und (Limthanmaphon und Zhang, 2004).

Das BTP benötigt entsprechende Implementierungen (Framework und APIs, s. Abb. 3.3) auf Koordinator- und Teilnehmer-Seite. Die Protokollnachrichten sollen innerhalb der Nachrichten des verwendeten Kommunikationsprotokolls für Services übertragen werden. Im Falle von Web Services werden die Kommandos als SOAP-Nachrichten, der Transaktionskontext als SOAP-Header übertragen.

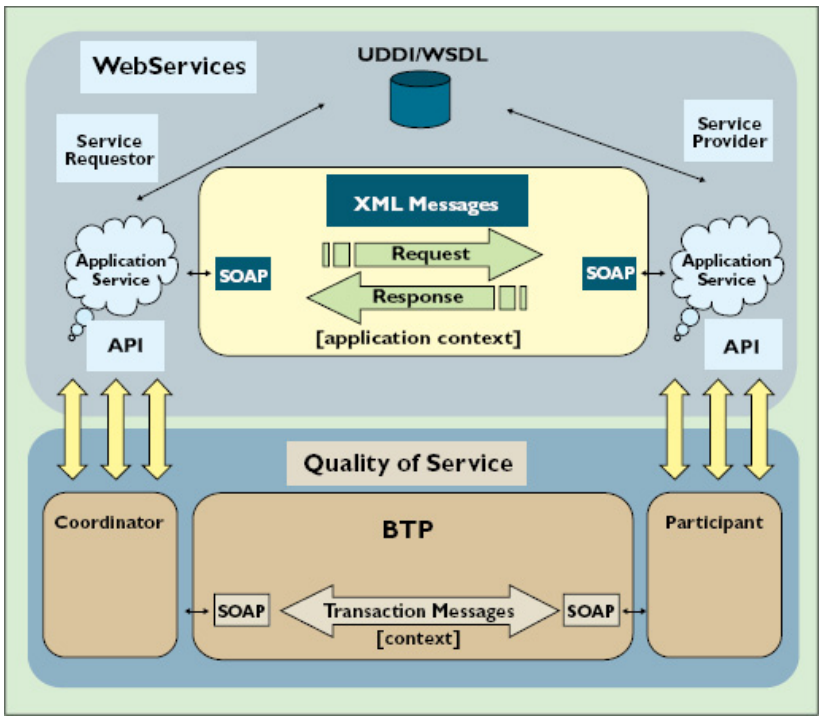

Abbildung 3.3.: Business Transaction Protocol (Little, 2003a)

3.4.2. Web Service Coordination

Bereits seit 2002 entwickeln verschiedene Unternehmen und Organisationen Spezifikationen und Vorschläge zur Standardisierung für Architekturen und Protokolle rund um das Web Services Referenzmodell (s. Abb. 3.1). Dabei geht es vor allem darum, Web Services für Geschäftsprozesse und B2B-Anwendungen zu benutzen und Nachteile der ursprünglichen Web Services Spezifikationen zu kompensieren. Insbesondere sind einfache Web Services zustandslos, nicht sicher und nicht transaktional. Des Weiteren ist das bisher am häufigsten genutzte Transportprotokoll HTTP nicht grundlegend zuverlässig und daher nicht geeignet für Dienstgütezusicherungen. Für Transaktionen wurden von Arjuna, BEA, Hitachi, IBM, IONA und Microsoft die Spezifikationen WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity entwickelt.

3.4.2.1. WS-Coordination

WS-Coordination (WS-C, 2005) legt die Rollen (Koordinator und Teilnehmer, siehe 3.2.4) der an der Transaktion beteiligten Web Services fest und definiert Protokoll und Nachrichten für das Anlegen des Transaktionskontextes (Aktivierung) sowie für die Registrierung von teilnehmenden Web Services. Das erzeugen des Kontextes ist gleichbedeutend mit dem Starten einer Transaktion, danach wird der Kontext wie beim BTP (s. 3.4.1) bei jedem (transaktionalen) Aufruf als XML-Element `CoordinationContext` in einem SOAP-Header angehängt. Das Interaktionsdiagramm in Abb. 3.4 zeigt anhand des Freigabemanager-Beispiels aus 1.2 die Komponenten, Rollen sowie den typischen Ablauf einer Transaktion. Dabei befindet sich die Anwendungslogik in den mit Client und Service bezeichneten Komponenten. Pro Transaktion existiert nur ein Koordinator, aber beliebig viele Teilnehmer. Die als „Coordination Service“ markierten Komponenten sind Bestandteil des Transaktions-Frameworks.

Die Art der Transaktion wird beim Anlegen des Transaktionskontextes als Koordinationstyp festgelegt. Sie bestimmt welches Terminierungsprotokoll (mit „Termination protocol“ bezeichnete Pfeile in Abb. 3.4) verwendet wird. Es sind beliebige Terminierungsprotokolle denkbar, im Wesentlichen existieren jedoch zwei Koordinationstypen: Atomare (ACID-) Transaktionen sowie langlebige Transaktionen (Business Activities).

WS-Coordination sieht vor, dass alle Web-Service-Aufrufe asynchron implementiert werden, eine synchrone Verarbeitung ist optional.

3.4.2.2. WS-AtomicTransaction

WS-AtomicTransaction (WS-AT, 2005) spezifiziert die Abwicklung von verteilten ACID-Transaktionen mit Web Services ähnlich den BTP-Atoms. ACID-Transaktionen verhalten sich

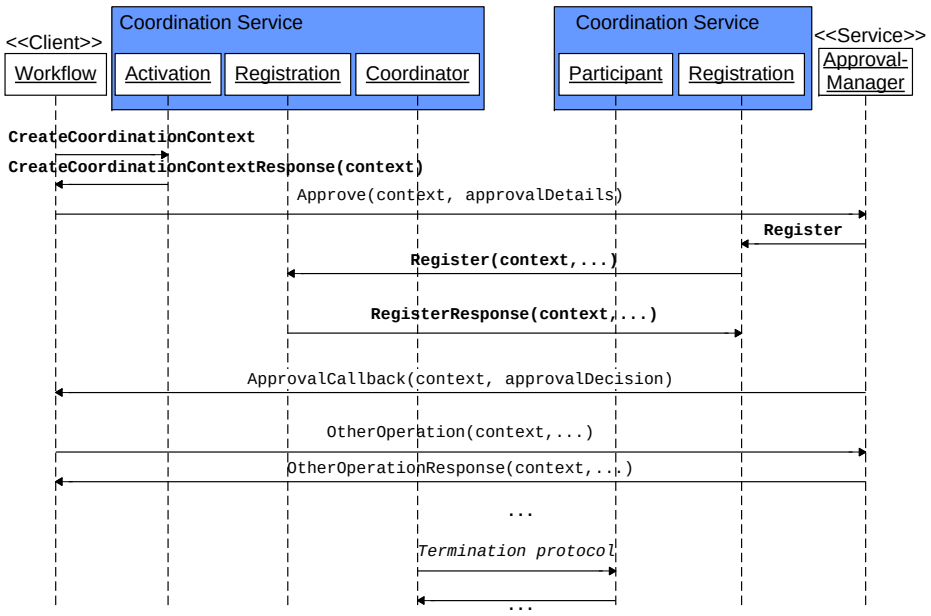

Abbildung 3.4.: WS-Coordination (eigene Darstellung)

hier wie X/Open-Transaktionen. Die Spezifikation beschreibt das so genannte „Completion“-Protokoll, welches dazu dient, es einem außenstehenden Web Service (Initiator) zu ermöglichen, die Transaktion mittels `Commit` oder `Abort` zu beenden. Der Initiator wird nach Beendigung der Transaktion über das Ergebnis (`Completed` oder `Aborted`) informiert. Die Beendigung wird über zwei 2-Phase-Commit-Protokolle durchgeführt, die jeweils zwischen dem Koordinator und einem Teilnehmer ablaufen. Services, die im Rahmen der Transaktion nur flüchtige Ressourcen benutzt haben, sollten sich für das „Volatile-2PC“-Protokoll registrieren, während Services, die auf persistenten Ressourcen wie z. B. Datenbanken arbeiten, das „Durable-2PC“-Protokoll benutzen sollten. Die Einzelheiten werden in (WS-AT, 2005) beschrieben, sind für diese Arbeit jedoch nicht interessant, da nur langlebige Transaktionen behandelt werden.

3.4.2.3. WS-BusinessActivity

WS-BusinessActivity (WS-BA, 2005) beschreibt die Abwicklung von Terminierungsprotokollen für langlebige Transaktionen zwischen den Koordinator- und Teilnehmer-Web-Services. Es sind dafür zwei Koordinationstypen vorgesehen, „AtomicOutcome“ und „MixedOutcome“. Im Falle von AtomicOutcome müssen alle Teilnehmer erfolgreich terminieren, oder es müssen alle Teilnehmer abbrechen bzw. kompensieren, wenn sie nicht kontrolliert aus der Transaktion aussteigen. MixedOutcome bedeutet, dass die Anwendungslogik entscheiden muss,

welche Teilnehmer erfolgreich terminieren oder abbrechen bzw. kompensieren müssen. Der Koordinationstyp muss beim Erzeugen des Transaktionskontextes angegeben werden.

Weiterhin werden zwei Protokolltypen definiert, die zwischen dem Koordinator und den verschiedenen Teilnehmern ablaufen können: „ParticipantCompletion“, d. h. der teilnehmende Web Service meldet dem Koordinator, dass die Aufgabe abgearbeitet ist und übernimmt keine weiteren Aufgaben, sowie „CoordinatorCompletion“, d. h. der Koordinator teilt dem Teilnehmer mit, dass die Transaktion beendet werden soll. Für beide Protokolltypen werden mögliche Aktionen und Zustandsübergänge sowohl aus Koordinator- als auch aus Teilnehmersicht exakt spezifiziert.

Die Funktionsweise des WS-BusinessActivity Terminierungsprotokolls wird hier an zwei Beispielen erklärt. In Abb. 3.5 wird eine Terminierung mit AtomicOutcome und CoordinatorCompletion verdeutlicht. Es haben sich vier teilnehmende Web Services registriert. Jeder erhält eine `Complete`-Nachricht. Der Koordinator wartet nun, bis von jedem Teilnehmer `Completed` empfangen wurde. In diesem Fall verlässt darauf hin einer der Teilnehmer die Transaktion mittels `Exit` (dies ist übrigens auch vor Start des Terminierungsprotokoll möglich). Der Koordinator registriert dies und quittiert diesen kontrollierten Ausstieg mit `Exited`. Die übrig bleibenden Teilnehmer werden mit `Close` dazu aufgefordert, die Transaktion abzuschließen. Die Transaktion ist beendet sobald der letzte Teilnehmer mit `Closed` quittiert hat.

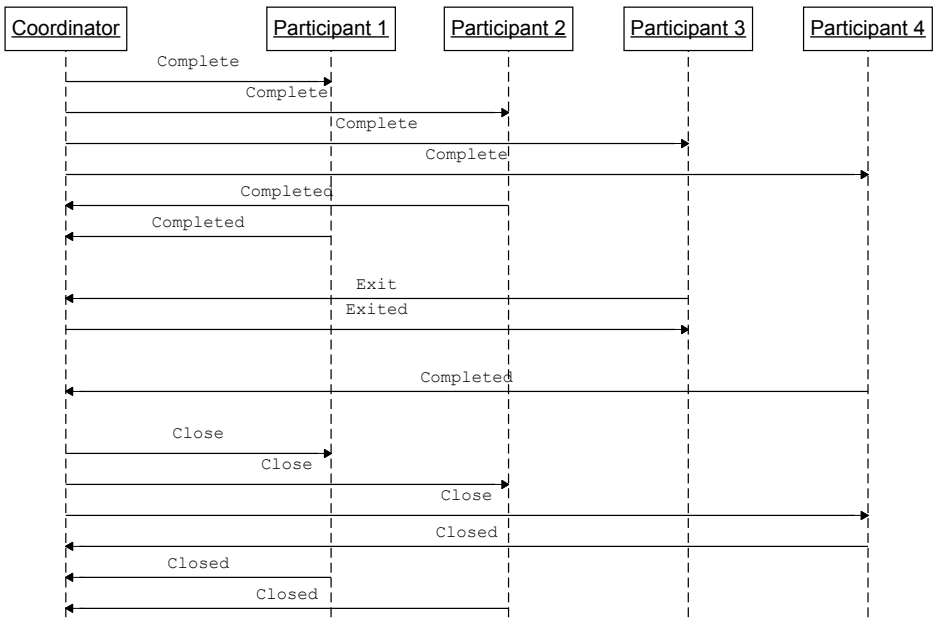

Abbildung 3.5.: WS-BusinessActivtiy (eigene Darstellung)

Abb. 3.6 zeigt die gleiche Ausgangssituation, nur dass hier einer der Teilnehmer einen Fehler mittels `Fault` meldet, welcher zunächst mit `Faulted` quittiert wird, bevor entschieden

wird, was mit den restlichen Teilnehmern passieren soll. Dies hängt vom Koordinationstyp ab: Ist der Typ AtomicOutcome, so müssen alle anderen Teilnehmer dazu gebracht werden, die Transaktion rückgängig zu machen. Ist der Typ MixedOutcome, so können die anderen Teilnehmer erfolgreich abschließen. Im Beispiel ist der Typ AtomicOutcome und es werden diejenigen Teilnehmer, die bereits `Completed` gesendet haben, mittels `Compensate` dazu gebracht, eine Kompensationsaktion durchzuführen. Den anderen Teilnehmern wird `Cancel` geschickt, um die laufende Transaktion abzubrechen. Auch `Cancel` und `Compensate` müssen quittiert werden.

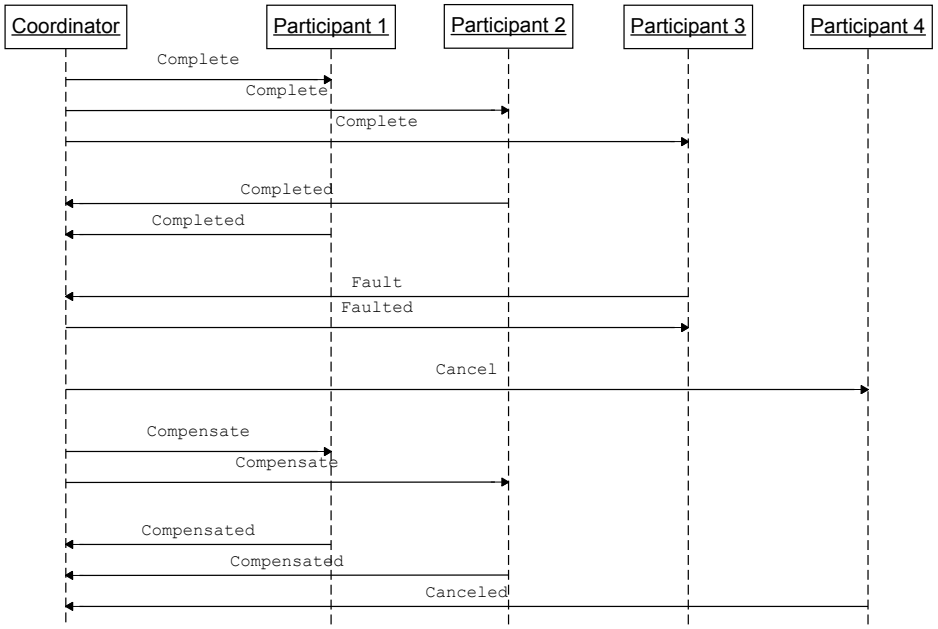

Abbildung 3.6.: WS-BusinessActivity mit Kompensation (eigene Darstellung)

Die Spezifikation legt nicht fest, wie Kompensation im einzelnen durchzuführen ist, sondern es wird lediglich gefordert, dass die beteiligten Web Services Kompensation beherrschen. Auch in dieser Arbeit werden keine Kompensations-Konzepte behandelt, sondern es wird für zu implementierende Web Services lediglich die Fähigkeit zur Kompensation angenommen.

Genau wie WS-Coordination sieht WS-BusinessActivity vor, dass alle Web-Service-Aufrufe asynchron implementiert werden, eine synchrone Verarbeitung ist optional.

Nach (Papazoglou, 2003) bilden WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity eine sinnvolle Ergänzung zu WS-BPEL. Web Service Orchestration kann so um transaktionale Fähigkeiten erweitert werden. In (Tai u. a., 2004) wird dazu ein konkreter Ansatz und ein Prototyp vorgestellt.

3.4.3. Web Service Composite Application Framework

2003 wurden von Arjuna, Fujitsu, IONA, Oracle sowie Sun drei weitere Spezifikationen zu Kontextmanagement (Web Services Context, WS-CTX, (WS-CTX, 2003)), Koordination (Web Services Coordination Framework, WS-CF, (WS-CF, 2003)) sowie Transaktionsmanagement (Web Services Transaction Management, WS-TXM, (WS-TXM, 2003)) veröffentlicht, die zusammengefasst als Web Service Composite Application Framework (WS-CAF) bekannt sind. WS-CAF ist wesentlich umfangreicher und detaillierter als WS-Coordination, die grundlegenden Konzepte sind aber vergleichbar.

WS-CAF ist eine generische Lösung für jegliche Art von verteilten Aktivitäten. Sie ist in drei Schichten aufgeteilt, die wie in Abb. 3.7 gezeigt in die Web Services Schichtenarchitektur einzuordnen sind.

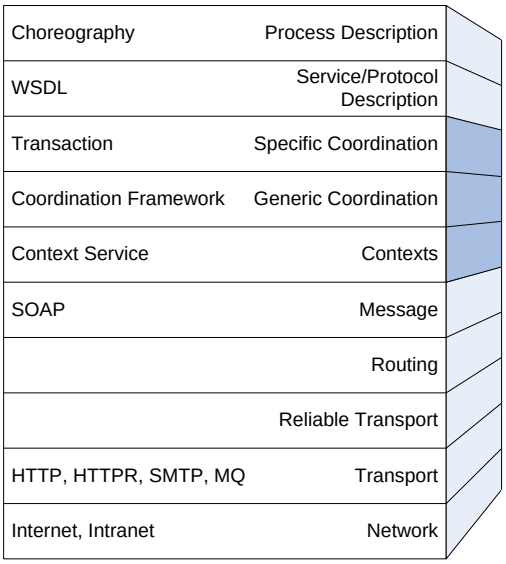

Abbildung 3.7.: WS-CAF Schichtenmodell (WS-CTX, 2003, S. 12)

An dieser Stelle werden nur die Hauptunterschiede zu WS-Coordination genannt. Zum genaueren Vergleich werden die drei Schichten in Anhang A.1 beschrieben.

Im Gegensatz zu WS-Coordination werden Kontext und Koordination im WS-CAF getrennt behandelt. Der Autor merkt hier an, dass verteilte Aktivitäten immer in irgendeiner Form koordiniert werden müssen. Dazu muss ohnehin Kontextinformation ausgetauscht werden, d. h. WS-CTX wird nicht „stand-alone“ ohne eine Spezifikation zur Koordination wie WS-CF benutzt werden. Es ist daher (an anderer Stelle) kritisch zu überprüfen, welche anderen Anwendungen von WS-CTX es geben könnte, die diese Trennung und die damit einhergehende Verkomplizierung rechtfertigen.

WS-CAF spezifiziert und unterscheidet explizit geschachtelte Aktivitäten, Koordinationen (Interpositionen) und Transaktionen (Nested Transactions). Dieses Merkmal macht WS-CAF flexibler einsetzbar als WS-Coordination, führt aber ebenso zu einer Verkomplizierung der Spezifikation. Weiterhin werden Status- und Fehlernachrichten detaillierter spezifiziert als in WS-Coordination.

3.4.4. Ähnlichkeit der Konzepte

Wie eingangs bereits angedeutet, sind sich die Grundkonzepte der vorgestellten Ansätze BTP, WS-Coordination und WS-CAF sehr ähnlich. Alle drei basieren auf einer zentralen Instanz (Koordinator), die beliebig viele Teilnehmer koordiniert, welche sich zu diesem Zwecke zentral registrieren, sobald sie aufgerufen werden. Registrierung- und Terminierungsprotokolle sind immer nur für die Kommunikation zwischen dem Koordinator und genau einem Teilnehmer definiert. Dieses Modell wird sowohl für atomare als auch für langlebige Transaktionen verwendet.

Die beschriebenen Architekturen ähneln dem X/Open DTP Modell für ACID-Transaktionen (s. 3.2.1). Ein Web Service übernimmt hier die Rolle der Applikation, die die Transaktion initiiert und koordiniert, die weiteren beteiligten Web Services nehmen die Rolle der Ressourcenmanager ein. Die Kommunikation zwischen Web Service und Aktivierungs-Service beinhaltet Elemente des TX-Protokolls (z. B. Transaktion starten), die Registrierung und die Abwicklung des Terminierungsprotokolls beinhalten Elemente des XA-Protokolls.

Keine der Spezifikationen definiert ein sprachgebundenes API. JSR 156 (JSR-156, 2005) spezifiziert nun das „Java API for XML Transactions“, kurz JAXTX, ein einheitliches Java-API zum Zugriff auf verschiedene Transaktionsumgebungen, wie z. B. JTA, J2EE Activity Service (s. o.), aber auch die in den vorangehenden drei Abschnitten erwähnten Spezifikationen für Web-Service-Transaktionen. Eine Beschreibung dieses Ansatzes ist in (Little u. a., 2004, Kap. 10) zu finden. Es werden generische Operationen zur Definition von Demarkationspunkten (Transaktionsbeginn und -ende), zur Registrierung von Teilnehmern und zur Propagierung von Transaktionsinformationen (Kontext) über das Netz definiert. Neben dem API werden weitere architektonische Komponenten generisch spezifiziert, z. B. Transaktionsdienst, Teilnehmer und Kontext.

JAXTX stellt also eine gemeinsame Architektur für verschiedene Implementierungen von Transaktionsdiensten zur Verfügung, so dass über eine einheitliche Schnittstelle auf diese zugegriffen werden kann. Die Tatsache, dass dies möglich ist, zeigt noch einmal die Ähnlichkeit der vorgestellten Ansätze. Da im Zeitraum der Erstellung dieser Arbeit und insb. der Entwicklung der Software JAXTX noch in Bearbeitung durch den Java Community Process war und keine weitere Dokumentation und Bibliotheken o. Ä. zur Verfügung standen, konnte es für diese Arbeit nicht verwendet werden.

3.4.5. Schlussfolgerung

Sämtliche beschriebenen Spezifikationen sind nicht standardisiert. Nach (Little, 2003b) hat sich das BTP nicht durchgesetzt. WS-CAF wurde seit den ursprünglichen Versionen (s. 3.4.3) zwar bei OASIS weiterentwickelt, aber die größeren Aktivitäten sind bei WS-Coordination zu beobachten (s. a. Anhang A.3).

WS-Coordination und WS-AtomicTransaction werden bereits von IBM mit dem IBM WebSphere Application Server 6.0 (WAS60-AT, 2006) unterstützt. Sun unterstützt beide Spezifikationen im Project Tango (Carr, 2006) und Microsoft in der Windows Communication Foundation (MS-WCF, 2006).

Mit Apache Kandula (Kandula, 2006) existiert außerdem ein Open-Source-Projekt zur Schaffung einer Web-Service-Transaktions-Middleware auf Basis von Apache Axis (Axis, 2006) und Axis 2 (Axis2, 2006). Hier liegt der Schwerpunkt bisher nur auf atomaren Transaktionen mit WS-AtomicTransaction, dem Mapping des Transaktionskontexts auf lokale JTA-Transaktionen sowie der Interoperabilität mit verschiedenen J2EE-Transaktionsmanagern. Die Unterstützung von WS-BusinessActivity wird vom Kandula-Team angekündigt, jedoch ohne Nennung von Einzelheiten.

WS-Coordination wird also bereits unterstützt. Da WS-Coordination und WS-BusinessActivtiy zudem wesentlich kürzer und einfacher gehalten sind als die entsprechenden WS-CAF Spezifikationen, werden für diese Arbeit WS-Coordination und WS-BusinessActivity verwendet.

3.5. Mobile Web Services mit J2ME

3.5.1. Mobile Web Service Stacks

Sowohl das .NET Compact Framework von Microsoft als auch verschiedene Java 2 Micro Edition (J2ME) Profile sind auf einer Vielzahl von mobilen Geräten verfügbar und bilden nach (Grimm u. a., 2004) als „virtuelle Maschinen“ ein Mittel, um die Heterogenität verschiedener Hardware-Plattformen zu umgehen.

Für das .NET Compact Framework von Microsoft existiert nach (Schrörs, 2005, 3.1.2, 3.1.4, 3.2.2 und 4.3.1) mit der .NET Compact Framework Mobile Web Server Architecture eine ausreichende Lösung, während die in (Schrörs, 2005, 3.1.3 und 3.2.1) untersuchte J2ME-basierte Lösung IBM Mobile SOAP Server u. a. wegen der benötigten Infrastruktur und den Lizenzbedingungen als nur für spezielle Zwecke brauchbar eingestuft wird. In (Schrörs, 2005) wurde daher im Rahmen einer Diplomarbeit ein Web-Service-Framework für die

J2ME-Plattform entwickelt. Unterstützt wird dabei die Konfiguration „Connected Limited Device Configuration“ CLDC 1.1 (JSR-139, 2003) mit dem Profil „Mobile Information Device Profile“ MIDP 2.0 (JSR-118a, 2002). Für Web Services werden SOAP 1.2 (SOAP, 2003) mit RPC-Nachrichtenstil und SOAP-Encoding unterstützt. In einer Erweiterung der Diplomarbeit (Schrörs, 2006a) werden weiterhin das „Literal“-Encoding, der “Document“-Nachrichtenstil als „Wrapped“-Variante mit Bestimmung der auszuführenden Operation anhand des Root-Elements des XML-Dokuments, die Serialisierung und Deserialisierung von Aufrufparametern und SOAP-Headern gemäß JAX-RPC Spezifikation (JSR-101, 2003) sowie asynchrone Kommunikation (One-Way Messages) unterstützt (Schrörs, 2006b, Anhang A).

Darüber hinaus existiert unter dem Namen „Fast Web Services“ (Sandoz u. a., 2003) ein von Sun Microsystems initiiertes Projekt, welches das Problem des relativ großen, auf der Lesbarkeit von XML beruhenden Overheads von SOAP-Dokumenten adressiert. Dieser Overhead kann insbesondere auf ressourcenbeschränkten Geräten zu Performance-Engpässen beim Serialisieren und Deserialisieren der Nachrichten führen. Bei Fast Web Services kommen anstelle von XML redundanzvermeidende Formate wie „Fast Infoset“ (Sandoz u. a., 2004), binäre Formate wie ASN.1 (ASN1, 2006) oder Kombinationen aus beiden zum Einsatz. Es geht hier lediglich um das Format der serialisierten Daten.

Seit einiger Zeit findet eine Ausprägung des Architekturstils REST (Representational State Transfer, (Fielding, 2000)) Verbreitung für Web-Service-ähnliche Anwendungen. REST wird für die möglichst einfache und effiziente Kommunikation zwischen verteilten Systemen mittels XML über HTTP verwendet. Es wird jedoch nicht festgelegt, wie Daten zu serialisieren sind, und es werden zwingend HTTP als zugrunde liegendes Protokoll sowie die dazugehörigen Architekturen (z. B. Web Server) vorausgesetzt.

Für einführende Informationen zu J2ME siehe z. B. (J2ME, 2006) oder (Li und Knudsen, 2005), zu Web Services siehe z. B. (Zimmermann u. a., 2003/2005).

3.5.2. Besondere Problemstellung Mobilität

Bei der Realisierung von Anwendungen auf mobilen Geräten sind im Gegensatz zu stationären Systemen folgende Rahmenbedingungen zu beachten (s. a. (Dolan, 2004)):

- Offline-Situationen: Mobile Geräte sind unter Umständen nicht ständig online.
- Kommunikationsstörungen: Wenn ein mobiles Gerät online ist, kann es zu Verbindungsabbrüchen durch das Verschwinden eines Partners kommen. Dies kann mit oder ohne Ankündigung passieren. Es ist zu ermitteln, von welcher Kommunikationsschicht ein unangekündigtes Verschwinden entdeckt und von welcher Schicht dieses behandelt werden soll. In jedem Fall muss die Anwendungslogik sowie das Transaktionsmanagement diese Fälle entsprechend behandeln.

- Bandbreite: Die verfügbare Bandbreite variiert und ist unter Umständen nicht so hoch wie in LANs und WLANs. Die Datenlaufzeiten variieren je nach benutztem Netzwerk.
- Kosten: Die Verbindungskosten sind unter Umständen hoch (GPRS, UMTS) und werden unterschiedlich (Zeit, Daten) berechnet. Es ist daher empfehlenswert, sich auf WLAN- bzw. Bluetooth-Verbindungen zu konzentrieren. (Nicht im Rahmen dieser Arbeit behandelt.)
- Sicherheit: Die Risiken des Diebstahls von Geräten und des Abhörens drahtloser Kommunikation sind größer als bei stationären, drahtgebundenen Systemen. (Nicht im Rahmen dieser Arbeit behandelt.)

Weiterhin sind bei der Anwendungsentwicklung für kleine mobile Geräte im Gegensatz zu „großen“ mobilen Geräten wie z. B. Laptops, begrenzte Ressourcen hinsichtlich UI, Rechenleistung, Speicher und Batteriekapazität zu beachten. J2ME mit CLDC/MIDP trägt dem dadurch Rechnung, dass nur eine sehr begrenzte Untermenge der Programmiersprache Java zur Verfügung steht (siehe (Li und Knudsen, 2005)). Die in (Schrörs, 2006a) benutzten Implementierungen „kXML 2“ und „kDOM“ (kXML2, 2006) bieten „Lightweight“-Versionen üblicher XML-Standards und -Spezifikationen. Für den Entwurf eines SOAP-APIs wurde „kSOAP 2“ (kSOAP2, 2006) als Vorbild benutzt, eine Implementierung, die ebenfalls mit sehr knappen Ressourcen auskommt.

3.6. Frameworks und Entwurfsmuster

3.6.1. Entwurfsmuster für Frameworks

Entwurfsmuster und Frameworks sind nach (Jacobsen u. a., 1997) Beispiele für Abstraktion auf Architektur-Ebene. Entwurfsmuster haben dabei einen höheren Abstraktionsgrad als Frameworks und sind unabhängig vom Anwendungsgebiet während Frameworks abhängig vom jeweiligen Anwendungsgebiet sind. Entwurfsmuster sind feingranular, Frameworks sind grobgranular. Siehe dazu auch (Schmidt und Buschmann, 2003).

Wie in 2.4 bereits erwähnt wurde, kann der Einsatz von Entwurfsmustern die Framework-Entwicklung genau wie die Anwendungsentwicklung erheblich vereinfachen.

In der Framework-Entwicklung spielen besonders Generalisierung und Parametrisierung eine wichtige Rolle. Generalisierung ist Voraussetzung für die Wiederverwendbarkeit des Frameworks für viele Anwendungen. Entwurfsmuster in der Framework-Entwicklung konzentrieren sich auf abstrakte und generische Konzepte, die z. B. aus Dekomposition von Klassen

und Strukturen einer speziellen Anwendung gewonnen werden können. Typisch – und grundlegend notwendig – ist hierbei die Aufteilung verschiedener Aspekte von Funktionalität auf einzelne Klassen.

Mittels Parametrisierung wird gesteuert, welche Framework-Klassen angepasst bzw. erweitert werden können bzw. müssen und wie das geschehen muss. Man kann ein Framework auch als parametrisierte Repräsentation von Anwendungen eines bestimmten Anwendungsklasse betrachten. Ein Parameter repräsentiert dabei eine anpassbare bzw. erweiterbare Komponente des Frameworks. In „Whitebox-Frameworks“ definiert der Benutzer neue Komponenten auf Basis existierender (abstrakter) Komponenten, in „Blackbox-Frameworks“ werden existierende (konkrete) Komponenten alleine über ihre Interfaces benutzt. Es kommen daher unterschiedliche Arten von Entwurfsmustern in Whitebox- und Blackbox-Frameworks vor.

Das Framework aus (Schrörs, 2006a) ist hauptsächlich ein Whitebox-Framework. Die Web-Service-Implementierungsklassen sowie benutzerdefinierte Datentypen werden von (abstrakten) Framework-Klassen abgeleitet. Es wird sich zeigen, dass das Transaktions-Framework das bestehende Framework um Whitebox-Komponenten und Blackbox-Komponenten erweitert.

Für die Entwicklung speziell von Frameworks können neben Entwurfsmustern zusätzlich Metamuster (Jacobsen u. a., 1997) nützlich sein. Sie beschreiben, wie Klassen von Entwurfsmustern in Frameworks zum Tragen kommen können. Der Fokus liegt dabei auf der anwendungsunabhängigen Konstruktion von Frameworks. Metamuster helfen bei der Spezifikation der Parameter eines Frameworks, unterschiedliche Metamuster spezifizieren dabei unterschiedliche Formen der Parametrisierung. Dabei geht es vor allem darum, wie die Anknüpfungspunkte der Anwendungslogik an die Framework-Logik in Form von aufrufenden Methoden (Template-Methode) und aufgerufenen Methoden (Hook-Methode) zu entwerfen und innerhalb der Klassenstruktur der zu verwendenden Entwurfsmuster zu platzieren sind.[5]

Folgende Mechanismen werden nach (Jacobsen u. a., 1997) von Metamustern in unterschiedlicher Art und Weise kombiniert:

1. Template- und Hook-Methoden
2. Vererbung und Referenzierung
3. Vereinigung vs. Aufteilung von Funktionalität

[5] Metamuster beschreiben allgemeine Implementierungstechniken, die auf Entwurfsmuster anwendbar sind. Sie beschreiben nicht Entwurfsmuster auf einer Meta-Ebene.

Es wird zwischen zwei Kategorien von Metamustern unterschieden. Die erste Kategorie enthält zwei Metamuster, welche die Terminologie der Template- und Hook-Methoden definieren sowie gewisse Prinzipien festlegen:

Template-Hook: Dieses Metamuster basiert darauf, dass bestimmte Methoden (Hook-Methoden) von anderen Methoden (Template-Methoden) aufgerufen werden, um gewisse Aufgaben zu erfüllen. Die Hook-Methoden eines Frameworks können bzw. müssen (falls sie als abstrakt deklariert wurden) von spezifischen Anwendungen überschrieben werden. Ein solcher Mechanismus wird auch als „Inversion of Control“ (Schmidt und Buschmann, 2003) bezeichnet. Nahezu alle Entwurfsmuster (z. B. aus (Gamma u. a., 1995)) lassen sich nach diesem Muster für die Frameworkentwicklung verwenden.

Narrow Inheritance Principle: Dieses Prinzip besagt im Wesentlichen, dass möglichst wenige Hook-Methoden zur Abarbeitung einer Aufgabe verwendet werden sollten, dass aber trotzdem größtmögliche Flexibilität gewährleistet sein sollte. Mehr Hook-Methoden erhöhen zwar die Flexibilität des Frameworks, aber auch den Aufwand der Anpassung für eine bestimmte Anwendung. Es ist schwierig, ein objektives Maß für dieses Prinzip festzulegen, daher sollte bei der Framework-Entwicklung jedes verwendete Muster kritisch daraufhin geprüft werden.

Die zweite Kategorie enthält sieben Metamuster zur Komposition von Klassen, die beschreiben wie Template- und Hook-Methoden auf die Klassen eines Frameworks bzw. eines Entwurfsmusters aufgeteilt sind und wie diese Klassen zueinander in Beziehung stehen:

Unification: Template- und Hook-Methode befinden sich in derselben Klasse. Für jede Variation der Hook-Methode muss eine neue Subklasse implementiert werden. Die Hook-Methode kann nicht zur Laufzeit verändert werden. Beispiele aus (Gamma u. a., 1995): Factory Method, Template Method.

1:1 Connection: Template- und Hook-Klasse (diese Klassen enthalten die enstprechenden Methoden) sind unterschiedlich und nicht Teil derselben Vererbungshierarchie. Dieses Muster erlaubt Änderungen der Hook-Methoden zur Laufzeit, da unterschiedliche Instanzen von Subklassen der Hook-Klasse mit dem Template-Objekt assoziiert werden können. Beispiele aus (Gamma u. a., 1995): Builder, Prototype, Bridge, Command, Strategy, State.

1:N Connection: Dieses Muster ist mit dem Muster „1:1 Connection“ vergleichbar, außer dass ein Template-Objekt eine Kollektion von Hook-Objekten referenzieren kann.

1:1 Recursive Connection: Die Template-Klasse ist eine Subklasse der Hook-Klasse. Die Template- und Hook-Methoden haben überlicherweise denselben Namen, wobei die

Template-Methode die Hook-Methode überschreibt, diese jedoch auch aufruft. Auf diese Weise wird eine Kette von Aufrufen von unten (Subklasse) nach oben (Superklassen) in der Vererbungshierarchie erzeugt. Beispiel aus (Gamma u. a., 1995): Chain of Responsibility, auch bekannt als Interceptor ((Schmidt u. a., 2000), (Zimmermann u. a., 2003/2005, 3.3)).

1:N Recursive Connection: Dieses Muster ist mit dem Muster „1:1 Recursive Connection“ vergleichbar, außer dass ein Template-Objekt eine Kollektion von Hook-Klassen referenzieren kann, so dass die Aufrufkette innerhalb einer Baumstruktur stattfindet. Beispiele aus (Gamma u. a., 1995): Composite, Interpreter.

1:1 Recursive Unification: Template- und Hook-Methode sind dieselbe Methode einer Klasse, wobei ein Objekt dieser Klasse ein anderes Objekt der Klasse referenziert. Wie im Fall von „1:1 Recursive Connection“ kann so eine Kette von Aufrufen zwischen Objekten erzeugt werden, nur dass die Objekte alle von derselben Klasse sind.

1:N Recursive Unification: Entsprechend „1:N Recursive Connection“.

Diese Metamuster können bei der Entwicklung eines Code-Generators hilfreich sein, welcher etwa die anwendungsspezifischen Klassen für transaktionale Web Services (in Form von Web-Service-Interface- und -Implementierungs-Klassen) generieren könnte. Die Web-Service-Implementierungsklassen müssten von Framework-Klassen abgeleitet werden, wobei gewisse Methoden überschrieben werden.

3.6.2. Abgrenzung: Middleware

Middleware ist nach (Schmidt und Buschmann, 2003) Software, die die Wiederverwendbarkeit fördert, indem sie Standardlösungen für verbreitete Aufgaben zur Verfügung stellt. Transaktionsmanagement ist eine solche Aufgabe. Frameworks und Entwurfsmuster erleichtern die Entwicklung von Middleware und von darauf basierenden Anwendungen.

Die Framework-Klassen sowie generierter Code (s. o.) bilden also eine Transaktions-Middleware für mobile Web Services. Die Middleware stellt dem Entwickler der Anwendungslogik APIs und Standardkomponenten zur Verfügung, die die Komplexität des Frameworks verbergen und es ihm ermöglichen, sich auf die Anwendungslogik zu konzentrieren.

3.7. Fazit

Transaktionsmanagement für Web Services muss konzeptionell ähnlich aussehen wie Transaktionsmanagement in Komponentenumgebungen wie CORBA und J2EE. Den beteiligten

Applikationen müssen Interfaces zur Verfügung gestellt werden, über die Transaktionen gestartet und beendet werden können und die Status der Transaktion und der beteiligten Web Services abgefragt werden können. Der Transaktionskontext muss bei allen transaktionalen Aufrufen übergeben werden. An der Transaktion teilnehmende Systeme müssen sich registrieren, damit nach erfolgter Abarbeitung Terminierungsprotokolle bzw. nach Systemausfällen Wiederherstellungs-Protokolle abgearbeitet werden können. Die Registrierungs- und die Terminierungsprotokolle müssen von allen teilnehmenden Web Services verstanden und korrekt implementiert werden.

Das zu entwickelnde Transaktions-Framework muss diese Anforderungen erfüllen. Dabei werden im Rahmen dieser Arbeit nur langlebige Transaktionen unterstützt, die nicht die Atomicity- und Isolation-Eigenschaften besitzen, sondern Datenkonsistenz durch den Einsatz von Kompensation so gut wie möglich zu wahren versuchen. Die Fähigkeit zur Kompensation wird für die transaktionalen Web Services vorausgesetzt. Sie ist vom Entwickler der Geschäftslogik bei der Implementierung entsprechend umzusetzen.

Die langlebigen Transaktionen sollen gemäß der Spezifikationen WS-Coordination und WS-BusinessActivity abgewickelt werden.

Zu beachten ist dabei, dass gemäß dieser Spezifikationen die gesamte Kommunikation asynchron ablaufen muss. Es muss daher nach jedem Aufruf der aktuelle Zustand des aufrufenden Systems persistiert werden. Außerdem müssen sämtliche eintreffenden Aufrufe mit dem richtigen persistierten Zustand korreliert und ggf. gemäß aktuellem Transaktionsstatus synchronisiert werden.

Auf kleinen mobilen Geräten müssen die an die begrenzten Ressourcen angepassten Laufzeitumgebungen benutzt werden. Im Rahmen von (Schrörs, 2005) und (Schrörs, 2006a) wurde bereits ein Framework für mobile Web Services mit J2ME, kXML 2 und kDOM realisiert.

Bei der Weiterentwicklung dieses Frameworks sollten spezielle funktionale und nichtfunktionale Anforderungen von Frameworks berücksichtigt werden. Ein Mittel, dies zu erreichen, ist die Verwendung von Entwurfsmustern bzw. Metamustern.

4. Analyse

Das in (Schrörs, 2005) entwickelte und vom gleichen Autor erweiterte (Schrörs, 2006a) Framework für mobile Web Services soll um die Unterstützung transaktionaler Web Services erweitert werden. Im Folgenden werden dafür anhand der existierenden Spezifikationen, der Arbeit von Schrörs und allgemeiner Betrachtungen funktionale und nichtfunktionale Anforderungen formuliert und dementsprechend zu erstellende Komponenten und deren APIs erarbeitet.

Interaktions- und Klassen- und Zustandsdiagramme sind dabei in einer an die „Unified Modeling Language“ (UML, s. (UML, 2006)) angelehnten Notation gehalten. Bei Diagrammen ohne Quellenangabe handelt es sich in diesem Kapitel um eigene Darstellungen.

Es wird *nicht* mehr auf die grundlegenden Anforderungen an Frameworks für mobile Web Services (Anwendungsfälle Web-Service-Entwicklung und -Kommunikation, SOAP-Nachrichtenaustausch, SOAP-Nachrichtenformat, SOAP-Rollen, Fehlerfälle (SOAP-Faults), Serialisierung/Deserialisierung, Transport-Protokoll) eingegangen, da diese in (Schrörs, 2005, Kap. 4) behandelt werden und im dort entwickelten Protoypen in einem für diese Arbeit ausreichenden Ausmaß realisiert wurden (s. a. (Schrörs, 2006b)).

4.1. Funktionale Anforderungen

Funktionalen Anforderungen betreffen im Falle der Abwicklung von verteilten Transaktionen (unabhängig davon ob atomar oder langlebig) die Kommunikation zwischen Koordinator und Teilnehmer sowie die Transaktionslogik auf beiden Seiten. Die Protokolle, die von den als geeignet ausgewählten Spezifikationen WS-Coordination und WS-BusinessActivity (s. 3.4.2) spezifiziert werden, müssen eingehalten werden (4.1.2). Das zu entwickelnde Framework soll dabei dem Entwickler der Anwendungslogik möglichst viel dieser Verantwortung abnehmen (4.1.1), weshalb genau zu ermitteln ist, welche generischen Aufgaben das Framework übernehmen kann und welche speziellen Aufgaben die Anwendungslogik übernehmen muss. Protokollnachrichten müssen zur richtigen Zeit sowie zu aktueller Rolle und aktuellem Zustand der beteiligten Systeme passend gesendet werden und es muss auf eingehende Nachrichten (Ereignisse) ebenso passend reagiert werden, falls das Protokoll es erfordert

(4.1.3). Weitere Anforderungen betreffen Kommunikation (4.1.4, 4.1.5), Persistenz (4.1.6), sowie Kompensationsfähigkeit (4.1.7).

4.1.1. Framework-Logik und Anwendungslogik

Bei der folgenden Betrachtung von funktionalen Anforderungen an das Transaktions-Framework muss zwischen Framework-Logik und Anwendungslogik unterschieden werden.

Die Framework-Logik besteht – wie sich im Folgenden zeigen wird – aus generischen Framework-Klassen und -Interfaces, die eine API zur Transaktionssteuerung anbieten und generische Aufgaben des Transaktionsmanagements übernehmen, sowie aus spezifischen Teilen, die mittels Codegenerierung als Web-Service-Implementierung erzeugt werden (s. 4.4).

Die Anwendungslogik enthält die Geschäftslogik der transaktionalen Web Services in Form von transaktionalen Operationen. Sie hat – wie sich ebenfalls im Folgenden herausstellen wird – auch Verantwortlichkeiten bezüglich der Einhaltung des WS-BusinessActivity-Protokolls.

Die Framework-Logik bildet eine Middleware (s. 3.6.2) und läuft zwischen der Logik des Web-Service-Frameworks und der Anwendungslogik ab (Abb. 4.1).

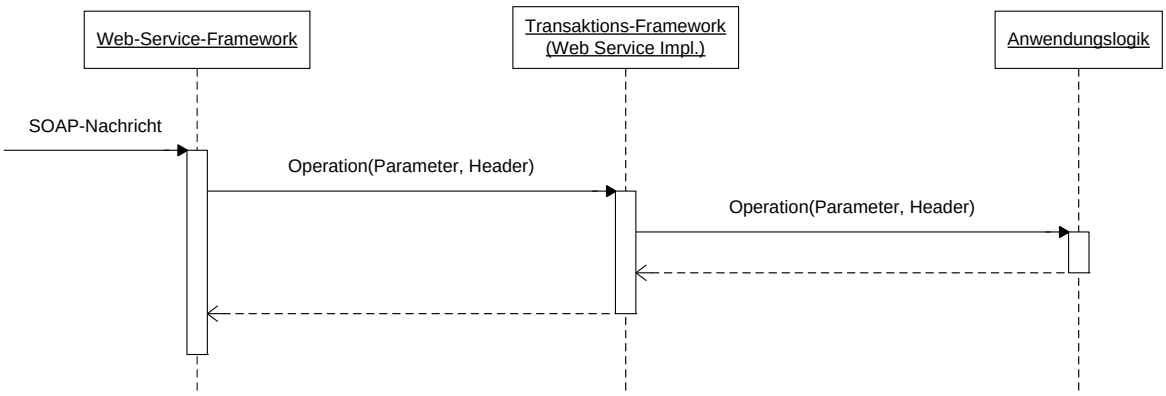

Abbildung 4.1.: Framework-Logik und Anwendungslogik (eigene Darstellung)

4.1.2. WS-Coordination und WS-BusinessActivity als Zustandsautomat

In Abschnitt 3.2.5 wurde bereits angedeutet, dass beim Transaktionsmanagement die Abwicklung von Terminierungsprotokollen immer zwischen Koordinator-Teilnehmer-Paaren ab-

läuft. Dies trifft auch auf WS-Coordination und WS-BusinessActivity zu. Die Protokolle können als Zustandsautomaten beschrieben werden. Die Zustände der Protokolle sind dabei jeweils aus Sicht des Koordinators und des Teilnehmers zu behandeln. Die unten aufgezählten Nachrichten bzw. Ereignisse verändern den Zustand entsprechend. WS-Coordination besteht dabei wie in 3.4.2.1 beschrieben aus Aktivierung (d. h. Starten der Transaktion) und Registrierung von Teilnehmern. In (WS-BA, 2005, Appendix A) werden für die beiden Protokolltypen von WS-BusinessActivity mögliche Zustände, Zustandsübergänge und notwendige Aktionen beim Empfangen und Senden von Nachrichten jeweils aus Koordinator- und Teilnehmersicht ausführlich beschrieben. In 4.1.3 wird im Rahmen der Zusammenstellung möglicher Transaktionsmuster noch einmal näher darauf eingegangen.

In den folgenden Abschnitten werden Nachrichten (aus Sicht des Senders) bzw. Ereignisse (aus Sicht des Empfängers) und Zustände der beiden Protokolle beschrieben.

4.1.2.1. Nachrichten in WS-Coordination

WS-Coordination definiert folgende Nachrichten für Aktivierung und Registrierung:

CreateCoordinationContext: Aufforderung des Koordinators an den Aktivierungs-Service (der Teil des Koordinators sein kann), eine Transaktion durch die Erstellung eines Transaktionskontextes zu starten. Die Aktivierungsanfrage enthält dabei den gewünschten Koordinationstyp.

CreateCoordinationContextResponse: Antwort des Aktivierungs-Service an den Koordinator. Der erstellte Kontext wird als Datenstruktur gemäß dem in (WS-C, 2005) referenzierten XML-Schema an den Koordinator übertragen.

Register: Registrierung eines Teilnehmers, gesendet vom Teilnehmer an den Registrierungs-Service (kann ebenfalls Teil des Koordinators sein), nachdem der Koordinator oder ein anderer bereits registrierter Teilnehmer eine Operation des Teilnehmers aufgerufen hat. Die Adresse des Registrierungs-Services wird dabei im Transaktionskontext übertragen. Die Registrierungsanfrage enthält dabei den gewünschten Protokolltyp.

RegisterResponse: Antwort des Registrierungs-Services an den Teilnehmer. Dieser geht nun in den Zustand `REGISTERED` bzw. `ACTIVE` (s. u.) über und führt die ursprünglich aufgerufene Operation aus.

4.1.2.2. Nachrichten in WS-BusinessActivity

WS-BusinessActivity definiert folgende Nachrichten für die Terminierung:

Complete: Aufforderung des Koordinators an einen Teilnehmer zum Beenden aller Operationen im Rahmen der aktuellen Transaktion.

Completed: Meldung eines Teilnehmers an den Koordinator über das Beenden aller Operationen im Rahmen der aktuellen Transaktion.

Close: Benachrichtigung vom Koordinator an einen Teilnehmer darüber, dass die Transaktion beendet wird und keine weiteren Nachrichten gesendet werden.

Closed: Bestätigung des Teilnehmers an den Koordinator über das Beenden der Transaktion.

Cancel: Aufforderung des Koordinators an einen Teilnehmer zum Abbrechen aller Operationen im Rahmen der aktuellen Transaktion.

Canceled: Bestätigung des Teilnehmers an den Koordinator über den erfolgten Abbruch aller Operationen im Rahmen der aktuellen Transaktion.

Compensate: Aufforderung des Koordinators an einen Teilnehmer zur Kompensation der bereits im Rahmen der aktuellen Transaktion abgeschlossenen Operationen.

Compensated: Bestätigung des Teilnehmers an den Koordinator über die erfolgten Kompensation der bereits im Rahmen der Operation abgeschossenen Operationen.

Exit: Meldung eines Teilnehmers an den Koordinator darüber, dass der Teilnehmer kontrolliert aus der Transaktion aussteigt.

Exited: Bestätigung des Koordinators an einen Teilnehmer über dessen kontrolliertes Aussteigen aus der Transaktion. Damit ist die Transaktion für diesen Teilnehmer beendet.

Fault: Meldung eines Teilnehmers an den Koordinator über einen Fehler in einer Operation des Teilnehmers. Der Fehler wird dabei gemäß dem in (WS-BA, 2005) referenzierten XML-Schema übertragen.

Faulted: Bestätigung des Koordinators an einen Teilnehmer über einen von diesem zuvor gemeldeten Fehler. Damit ist die Transaktion für diesen Teilnehmer beendet.

GetStatus: Anfrage des Koordinators an einen Teilnehmer oder eines Teilnehmers an den Koordinator nach dem Status des Protokolls aus der Sicht des angefragten Systems.

Status: Statusmeldung nach vorheriger Abfrage durch `GetStatus`. Der Protokollstatus wird dabei gemäß dem in (WS-BA, 2005) referenzierten XML-Schema übertragen.

4.1.2.3. Zustände in WS-Coordination

WS-Coordination definiert folgende Zustände während der Aktivierung und Registrierung. Die Zustandsnamen sind dabei in der Spezifikation nicht explizit zu finden, sondern wurden sinngemäß gewählt:

INIT: Der (potentielle) Teilnehmer nimmt noch nicht an der Transaktion teil (d. h. innerhalb der Transaktion wurde noch keine Operation des Teilnehmers aufgerufen). Die Aktivierung der Transaktion durch den Koordinator läuft ab, wenn alle (potentiellen) Teilnehmer sich noch im Zustand `INIT` befinden.

REGISTERING: Es wurde das erste Mal innerhalb der Transaktion eine Operation des Teilnehmers aufgerufen, woraufhin dieser sich durch Senden von `Register` beim Registrierungs-Service registriert. Alle weiteren in dieser Zeit empfangenen Operationsaufrufe müssen zwischengespeichert werden, bis die Registrierung abgeschlossen ist.

REGISTERED: Es wurde vom Registrierungs-Service die Nachricht `RegisterResponse` empfangen. Der Teilnehmer führt nun die ursprünglich aufgerufene und alle inzwischen aufgerufenen Operationen aus. Dieser Zustand ist gleichbedeutend mit dem Zustand `ACTIVE` des WS-BusinessAcitvity-Protokolls (s. u.).

4.1.2.4. Zustände in WS-BusinessActivity

WS-BusinessActivity definiert folgende Zustände vor und während der Terminierung:

ACTIVE: Teilnehmer ist gemäß WS-Coordination registriert und führt Operationen innerhalb der Transaktion aus.

COMPLETING: Der Teilnehmer beendet gerade alle Operationen im Rahmen der aktuellen Transaktion (nach Senden von `Complete` von Koordinator an Teilnehmer, nur für Protokolltyp CoordinatorCompletion).

COMPLETED: Der Teilnehmer hat alle Operationen im Rahmen der aktuellen Transaktion abgeschlossen (nach Senden von `Completed` von Teilnehmer an Koordinator).

CLOSING: Der Teilnehmer schließt die Transaktion gerade (erfolgreich) ab.

CANCELING: Der Teilnehmer bricht die Transaktion gerade ab (nach Empfang von `Cancel` aus Teilnehmersicht).

CANCELING_ACTIVE: Der Teilnehmer bricht die Transaktion gerade ab (nach Senden von `Cancel` im Zustand `ACTIVE` aus Koordinatorsicht).

CANCELING_COMPLETING: Der Teilnehmer bricht die Transaktion gerade ab(nach Senden von `Cancel` im Zustand `COMPLETING` aus Koordinatorsicht)

COMPENSATING: Der Teilnehmer kompensiert gerade die bereits abgeschlossenen Operationen (nach Senden von `Closed` von Teilnehmer an Koordinator und `Compensate` von Koordinator an Teilnehmer).

EXITING: Der Teilnehmer verlässt die Transaktion gerade geplant (nach Senden von `Exit` von Teilnehmer an Koordinator).

FAULTING_ACTIVE: Der Teilnehmer befindet sich in einem Fehlerzustand (nach Senden von `Fault` von Teilnehmer an Koordinator im Zustand `ACTIVE` oder `COMPLETING`).

FAULTING_COMPENSATING: Der Teilnehmer befindet sich in einem Fehlerzustand (nach Senden von `Fault` von Teilnehmer an Koordinator im Zustand `COMPENSATING`).

FAULTING: Der Teilnehmer befindet sich in einem allgemeinen Fehlerzustand (nach wiederholtem Fehler).

ENDED: Teilnehmer hat die Transaktion verlassen (Folgezustand von `CLOSING`, `FAULTING` oder `EXITING` nach entsprechender Bestätigung).

4.1.2.5. Zusammenfassung

Beispielhafte Interaktionsdiagramme für WS-Coordination und WS-BusinessActivity wurden bereits in Abb. 3.4, Abb. 3.5 und Abb. 3.6 gezeigt. Alle möglichen Zustandsübergänge von WS-BusinessActivity werden noch einmal für den Koordinationstyp ParticipantCompletion in Abb. 4.2 und für den Koordinationstyp CoordinatorCompletion in Abb. 4.3 zusammengefasst.

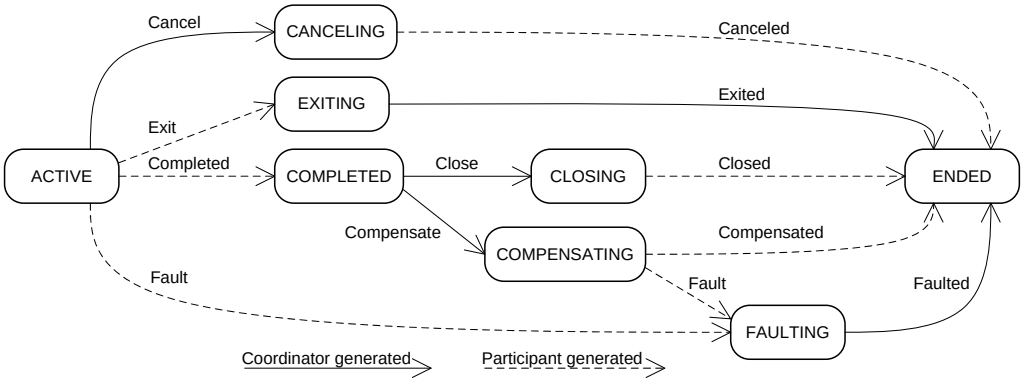

Abbildung 4.2.: Zustandsübergänge für ParticipantCompletion (WS-BA, 2005)

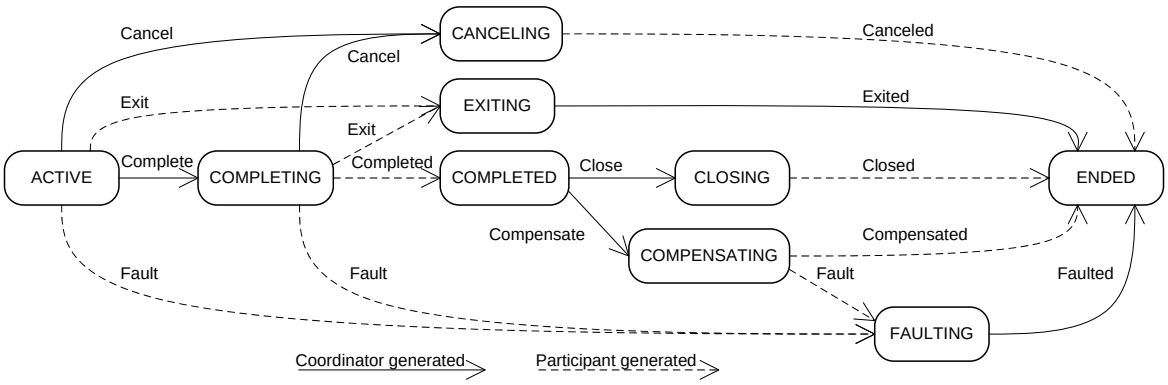

Abbildung 4.3.: Zustandsübergänge für CoordinatorCompletion (WS-BA, 2005)

4.1.3. Anwendungsfälle

Nachdem die einzelnen Protokollnachrichten und die möglichen Zustandsübergänge im vorigen Abschnitt beschrieben wurden, muss noch geklärt werden, welche Verantwortungen Framework- und Anwendungslogik dabei haben. Damit das Framework dem Anwendungsentwickler so viel Arbeit wie möglich abnehmen kann, sollte dem Framework soviel Verantwortung wie möglich gegeben werden. Dies trägt auch zur Vermeidung von Fehlern bei.

Es geht also um die Semantik der Protokollnachrichten, jeweils auf Framework- und auf Anwendungsseite. Die Semantik variiert dabei mit den Anwendungsfällen, für die das Framework benutzt wird.

Als „Anwendungsfälle“ werden hier für das Transaktionsframework die acht in Tabelle 4.1 aufgeführten Kombinationen aus Koordinationstyp, Protokolltyp sowie Rolle des Services betrachtet, wobei die Bezeichnung der Anwendungsfälle aus den Anfangsbuchstaben der englischen Bezeichnungen von Koordinationstyp (AtomicOutcome, MixedOutcome), Protokolltyp (ParticipantCompletion, CoordinatorCompletion) und Service-Rolle (Coordinator, Participant) zusammengesetzt wird. Je nach Kombination ergeben sich für Framework und Anwendungslogik unterschiedliche Verantwortlichkeiten hinsichtlich der Einhaltung der Protokolle WS-Coordination und WS-BusinessActivity.

Eine ausführliche Beschreibung des Protokollablaufs mit Unterscheidungen für die einzelnen Anwendungsfälle, Nachrichten und Zustandsübergänge, die für die Implementierung nützlich ist, findet sich in Anhang A.2. Hier genügt eine Zusammenfassung der wesentlichen Punkte. Bei der Verarbeitung der Protokollnachrichten wird wie in 4.1.1 erklärt zunächst die Framework-Logik ausgeführt, welche dann veranlasst, dass die zugehörige Anwendungslogik ausgeführt wird, sofern keine Fehler auftreten. Für den Entwurf des Zusammenspiels dieser beiden Schichten empfiehlt es sich, Meta-Patterns wie in 3.6.1 erklärt zu benutzen (siehe

Koordinationstyp	Protokolltyp	Service-Rolle	Bezeichnung
AtomicOutcome	ParticipantCompletion	Koordinator	APC
AtomicOutcome	ParticipantCompletion	Teilnehmer	APP
AtomicOutcome	CoordinatorCompletion	Koordinator	ACC
AtomicOutcome	CoordinatorCompletion	Teilnehmer	ACP
MixedOutcome	ParticipantCompletion	Koordinator	MPC
MixedOutcome	ParticipantCompletion	Teilnehmer	MPP
MixedOutcome	CoordinatorCompletion	Koordinator	MCC
MixedOutcome	CoordinatorCompletion	Teilnehmer	MCP

Tabelle 4.1.: Anwendungsfälle

auch das entsprechende Kapitel zum Entwurf von Web-Service-Implementierungsklassen 5.2.5).

Prinzipiell muss zwischen Operationen des Koordinators und von den mit dem Koordinator zusammenhängenden Aktivierungs- und Registrierungs-Services und Operationen der Teilnehmer unterschieden werden. Bis auf `GetStatus` und `Status` sind diese Mengen von Operationen disjunkt.

Die Aktivierung der Transaktion und die Registrierung der Teilnehmer (WS-Coordination) kann und sollte komplett von der Framework-Logik übernommen werden, sobald eine Operation des Koordinators bzw. eines Teilnehmers aufgerufen wird. Bei der Terminierung (WS-BusinessActivity) hat die Anwendungslogik dagegen auch einige Verantwortlichkeiten.

Die Aufgabe des Koordinators legt dabei den Koordinationstyp fest. Der Koordinationstyp wird dem Aktivierungs-Service als Parameter übergeben und entscheidet bei der Terminierung, ob alle Teilnehmer erfolgreich abschließen müssen oder ob auch eine Teilmenge reicht, um die Transaktion erfolgreich zu beenden.

Die Aufgabe jedes Teilnehmers legt zusammen mit dem Koordinator bzw. dessen Aufgabe den Protokolltyp fest. Der Protokolltyp wird dem Registrierungs-Service als Parameter übergeben und entscheidet bei der Terminierung, ob der Teilnehmer eine Aufforderung zum Beenden der Operationen im Rahmen der aktuellen Transaktion braucht oder ob der Teilnehmer das Ende der Bearbeitung selbst feststellen und melden kann.

Ist der Koordinationstyp AtomicOutcome, so kann und sollte die Framework-Logik des Koordinators (Anwendungsfälle APC, ACC) die Transaktion abschließen, und zwar erfolgreich mittels `Close`, wenn alle Teilnehmer `Completed` gesendet haben, und nicht erfolgreich mittels `Cancel` an Teilnehmer im Zustand `ACTIVE` bzw. `Compensate` an Teilnehmer im Zustand `COMPLETE`, sobald ein Teilnehmer `Fault` gesendet hat.

Ist der Koordinationstyp MixedOutcome, so muss die Anwendungslogik des Koordinators (Anwendungsfälle MPC, MCC) die Transaktion abschließen. Dies kann zu jedem beliebigen Zeitpunkt erfolgreich oder nicht erfolgreich geschehen. Der Gesamtstatus ist dazu nach jedem Empfang von `Completed`, `Exit`, `Fault` und `Status` in der Anwendungslogik zu überprüfen und die Transaktion ggf. erfolgreich mittels `Complete` bzw. nicht erfolgreich mittels `Cancel` und `Compensate` abzuschließen.

Ist der Protokolltyp für einen Teilnehmer ParticipantCompletion, so darf von der Anwendungslogik des Koordinators (Anwendungsfälle APC, MPC) nicht `Complete` des Teilnehmers aufgerufen werden, sondern die Anwendungslogik des Teilnehmers (Anwendungsfälle APP, MPP) muss zum gegebenen Zeitpunkt mittels `Completed` dem Koordinator das Ende der Verarbeitung melden. Ist der Protokolltyp dagegen CoordinatorCompletion, so muss die Anwendungslogik des Koordinators (Anwendungsfälle ACC, MCC) mittels `Complete` den Teilnehmer zum Beenden der Verarbeitung auffordern, was die Framework-Logik des Teilnehmers (Anwendungsfälle ACP, MCP) mit `Completed` bestätigt.

Unabhängig von Koordinations- und Protokolltyp dürfen auf Teilnehmerseite nur im Zustand `ACTIVE` weitere Teilnehmer aufgerufen werden, da ansonsten davon ausgegangen werden muss, dass nicht bekannt ist, ob der Koordinator die Transaktion gerade abschließt oder nicht. Auf Koordinatorseite können jederzeit weitere Teilnehmer aufgerufen werden, solange sichergestellt ist, dass das Terminierungsprotokoll mit diesen Teilnehmern irgendwann bis zum Status `ENDED` abläuft.

Im Beispiel des Freigabemanagers (s. 1.2) wäre als Protokolltyp ParticipantCompletion verwendbar: Der Freigabe-Web-Service sendet `Completed` an den Koordinator, sobald die Freigabeanfrage vom Benutzer bearbeitet wurde.

Die Framework-Logik muss beim Empfangen von Nachrichten prüfen, ob die Nachricht im aktuellen Zustand des Protokolls (aus lokaler Sicht) gültig ist. Ist dies nicht der Fall, so muss ein Fehler protokolliert werden und der Aufrufer über den Fehler informiert werden, falls möglich.

Es sind weiterhin Dienste vorstellbar, die an einer langlebigen Transaktion als Teilnehmer teilnehmen, selbst aber eine untergeordnete neue langlebige Transaktion als Koordinator starten, im Rahmen derer weitere Teilnehmer aufgerufen werden. Dazu müsste die transaktionale Operation eines Koordinators mit einem Transaktionskontext aufgerufen werden, als handele es sich um den Aufruf eines Teilnehmers. Die Operation startet dann die untergeordnete Transaktion durch Erzeugung eines neuen Kontexts. Das Callback zum ursprünglichen aufrufenden Web Service würde dann erst nach Beendigung der untergeordneten Transaktion ausgeführt werden.

Auf ähnliche Weise könnten atomare Transaktionen mit WS-AtomicTransaction von Teilnehmern an langlebigen Transaktionen ausgeführt werden. Solche Web Services wären dann gleichzeitig Koordinatoren für die atomaren Transaktionen.

Derartige geschachtelte Transaktionen werden von den WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity Spezifikationen nicht berücksichtigt. Sollen sie dennoch realisiert werden, so müssen die Protokolle entsprechend proprietär erweitert werden. Es wäre dann auch zu prüfen, ob die dadurch eingeführte Komplexität auch für kleine mobile Geräte annehmbar ist. Dies würde jedoch den Rahmen dieser Arbeit sprengen.

4.1.4. Propagierung des Transaktionskontexts

Das Framework muss Funktionen zur Verfügung stellen, die es erlauben, andere Web Services innerhalb eines Transaktionskontextes aufzurufen, d. h. der Transaktionskontext muss automatisch als SOAP-Header an den aufzurufenden Web Service übertragen werden. Listing 4.1 zeigt die Darstellung des Transaktionskontexts als SOAP-Header-Element `CoordinationContext` mit den für diese Arbeit wesentlichen Kindelementen `Identifier`, `CoordinationType` sowie `RegistrationService`.

Listing 4.1: Transaktionskontext gemäß (WS-C, 2005)

```
<soap:envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
 <soap:Header>
  <wscoor:CoordinationContext soap:mustUnderstand="true"
    xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor">
   <wscoor:Identifier>
    http://www.informatik.haw-hamburg.de/ws/#8237123801
   </wscoor:Identifier>
   <wscoor:CoordinationType>
    http://schemas.xmlsoap.org/ws/2004/10/wsba/AtomicOutcome
   </wscoor:CoordinationType>
   <wscoor:RegistrationService>
    <wsa:Address
      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
     http://coordinator.domain.com:8080/service
    </wsa:Address>
   </wscoor:RegistrationService>
  </wscoor:CoordinationContext>
 </soap:Header>
 ...
 <soap:Body> <!-- Example: Call to approval manager service -->
  <app:Approve xmlns:app="...">
   ...
  </app:Approve>
 </soap:Body>
</soap:envelope>
```

4.1.5. Asynchroner Nachrichtenaustausch und Korrelierung

Die Spezifikationen WS-Coordination und WS-BusinessActivtity sehen als bevorzugtes Aufrufmodell für alle Operationen asynchrone Aufrufe vor. Optional können auch synchrone Aufrufe angeboten werden, die dann je einen Operationsaufruf (als Request) sowie ein Callback (als Response) zusammenfassen. Um Workflows mit Benutzerinteraktionen und anderen potentiell lange andauernden Operationen abzubilden sowie um eine möglichst lose Kopplung der beteiligten Dienste zu erreichen, eignen sich asynchrone Aufrufe jedoch besser als synchrone, da dann nicht blockierend auf Antworten und Quittungen gewartet wird. Beim Einsatz von Web-Services auf mobilen Geräten können weitere Bedingungen hinzukommen (s. 3.5.2), die den Einsatz von asynchroner Kommunikation rechtfertigen. Das Ziel ist dabei, dass keine Systemressourcen durch das Warten auf eine Antwort blockiert werden und dass keine Inkonsistenzen dadurch entstehen, dass während des Wartens Fehler auftreten (z. B. ein Verbindungsabbruch).

In dieser Arbeit wird daher mit den asynchronen Ausprägungen der WS-Coordination- und WS-BusinessActivity-APIs gearbeitet. Infolgedessen müssen die Aufrufe zwischen den beteiligten Systemen entsprechend vom Framework korreliert und ggf. synchronisiert werden.

Die Spezifikationen WS-Coordination und WS-BusinessActivity sehen zur Korrelierung die Verwendung von WS-Addressing (WS-Addr, 2004) SOAP-Headern vor. Für die Korrelierung von Nachrichten ist es ausreichend, einen eingeschränkten Satz der Spezifikation zu verwenden. Die Einschränkungen betreffen den Datentyp `EndpointReferenceType` sowie einige Header.

Für den Datentyp `EndpointReferenceType` wird nur das `Address`-Element benutzt. Es enthält die URL des Services. Dies betrifft die Header `From`, `ReplyTo` und `FaultTo`, sowie auch an anderer Stelle verwendete XML-Elemente dieses Typs, z. B. `CoordinationContext/RegistrationService`.

Ein Beispiel für einen Header zeigt Listing 4.2.

Listing 4.2: WS-Addressing From-Header gemäß (WS-Addr, 2004)

```
1 <wsa:From xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
2   <wsa:Address>http://server.domain.de:8080/services/ExampleService</wsa:Address>
3 </wsa:From>
```

Beim Empfang einer Nachricht müssen WS-Addressing SOAP-Header der Anwendung zugänglich gemacht werden. Beim Senden einer Nachricht müssen WS-Addressing SOAP-Header automatisch eingefügt oder vom Aufrufer einem entsprechenden Call-API übergeben werden. Für die Behandlung der Header gibt es Unterschiede zwischen synchronem und asynchronem Nachrichtenaustausch.

Hier wird lediglich der asynchrone Fall betrachtet, so dass sich folgende Bedeutung der Header ergibt:

To: Der `To`-Header enthält die logische Zieladresse des Aufrufs als URI. Dieser Header wird von Gateways benötigt, um den Aufruf weiterzuleiten (s. 4.2.1.2).

Action: Der `Action`-Header enthält die logische Bezeichnung der auszuführende Operation. Da das ursprüngliche Framework die Operation aus dem Root-Element des SOAP-Bodies bestimmt („Wrapped"-Stil), wird dieser Header nicht benötigt.

MessageId: Dieser Header enthält eine global eindeutige Identifikation für die Nachricht.

RelatesTo: Dieser Header enthält die `MessageId` der Nachricht, auf die sich mit der aktuellen Nachricht bezogen wird. Einschränkung: Der Wert des Attributes `RelationshipType` enstpricht immer dem Defaultwert `Reply`, wird also nicht angegeben.

From: Der Absender der aktuellen Nachricht wird als `From`-Header mitgesendet. Dieser Header wird nur benötigt, falls `ReplyTo` nicht angegeben ist. Für diese Arbeit reicht es aus, nur das `Address`-Element zu benutzen (s. o.).

ReplyTo: Der `ReplyTo`-Header enthält Adresse des Web Services, an den eine eventuelle Antwort auf die aktuelle Nachricht geschickt wird (Callback). Es wird dabei angenommen, dass der Anwendungslogik bekannt ist, welche Nachricht an diesen Web Service geschickt werden muss (d. h. welche Operation aufgerufen werden muss). Auch hier reicht die Verwendung des `Address`-Elements zunächst aus.

FaultTo: An den vom `FaultTo`-Header angegebenen Endpunkt werden Fehlernachrichten geschickt, wenn der Empfänger der aktuellen Nachricht bei der Verarbeitung in einen Fehlerzustand gerät. Hier ist ebenfalls das `Address`-Element ausreichend.

Beim Empfang einer Nachricht, also dem Ausführen einer transaktionalen Web-Service-Operation, müssen die Header dem Framework sowie der Anwendungslogik zur Verfügung gestellt werden. Das Framework sollte benötigte Header beim Senden von Nachrichten automatisch generieren, falls möglich. Zum Beispiel können `MessageId` mit einer generierten ID und `From` mit dem Endpunkt des aktuell ausgeführten Web Services jeder ausgehenden Nachricht hinzugefügt werden.

Es ist anzumerken, dass diese Art der Korrelierung für jegliche Art von asynchronem Nachrichtenaustausch sinnvoll ist und nicht nur für Koordination und Transaktionsmanagement.

4.1.6. Persistenz

Eine Persistenzschicht für verteilte Transaktionen muss die Daten vorhalten, die zur Abwicklung der Transaktion über mehrere Aufrufe hinweg benötigt werden. Im Falle von langlebigen

Transaktionen müssen die Daten auch über das Ausschalten des Systems bzw. das Beenden der Web-Service-Laufzeitumgebung hinweg erhalten bleiben.

Auf Seiten des Koordinators müssen über mehrere Aufrufe innerhalb einer Transaktion folgende Daten über die Transaktions-ID (CoordinationContext/Identifier) zugreifbar bleiben:

- Die aufzurufende Operation sowie die Aufrufparameter beim Aufruf einer transaktionalen Operation, für die Dauer der Erzeugung des Transaktionskontextes (bis zum Empfang von `CreateCoordinationContextResponse`).
- Der gesamte Kontext (Transaktions-ID und Koordinationstyp sowie die logische Adresse des Registrierungs-Services)
- Logische Adresse, Protokolltyp und Protokollstatus für jeden registrierten Teilnehmer aus Koordinatorsicht

Auf Seiten der Teilnehmer müssen folgende Daten für die gesamte Transaktion zugreifbar bleiben:

- Aufzurufende Operationen sowie deren Aufrufparameter nach dem ersten Aufruf einer Operation innerhalb einer Transaktion für die Dauer der Registrierung (bis zum Empfang von `RegisterResponse`).
- Der gesamte Kontext, um ihn an weitere Teilnehmer zu propagieren.
- Logische Adresse des Koordinators, Protokolltyp und Protokollstatus aus Teilnehmersicht.

Auf beiden Seiten müssen alle relevanten Informationen für Anwendungs-Timer (s. 4.2.1.1) gespeichert werden. Dies sind neben der absoluten Zeit des Timeouts der Wert des `MessageId` WS-Addressing-Headers und eine Referenz auf den aufrufenden Web Service.

Es ist davon auszugehen, dass auch die Anwendungslogik Daten über mehrere asynchrone Operationsaufrufe hinweg speichern muss. Hier ist es dem Entwickler überlassen, wie dies geschieht.

Für die Datenspeicherung sind folgende Möglichkeiten denkbar:

- Flüchtiges Speichern im Arbeitsspeicher (z. B. in einer Hashtable innerhalb der Java Virtual Machine). Es muss dann sichergestellt werden, dass der Web Service Container ständig läuft. Dies ist im Falle von mobilen Geräten insbesondere für lange andauernde Aktionen nicht empfehlenswert. Wenn sichergestellt werden kann, dass das Erzeugen des Transaktionskontextes und die Registrierung schnell und zuverlässig funktionieren, können Operationen und Parameter auf diese Weise zwischengespeichert werden.

- Persistentes Speichern in nichtflüchtigem Speicher (z. B. mittels MIDP RecordStores, s. (Li und Knudsen, 2005, Kap. 8)). Es muss dann ein Mapping der Java-Objekte auf das Speicherformat stattfinden.

Um eine Wiederherstellung zu ermöglichen, müssen alle Informationen nicht-flüchtig abgelegt werden, die für die in 4.2.1.4 genannten Schritte benötigt werden. Dies sind alle o. g. Informationen, denn es kann zwischen den einzelnen Operationen jederzeit ein Crash stattfinden.

Auf kleinen mobilen Geräten sind auch nichtfunktionale Besonderheiten des nicht-flüchtigen Speichers zu beachten, die in 4.2.2 erwähnt werden.

4.1.7. Kompensationsfähigkeit

Der Entwickler der Anwendungslogik muss dafür sorgen bzw. beachten, dass sämtliche an langlebigen Transaktionen teilnehmenden Web Services „kompensationsbasiertes Transaktionsmanagement“ beherrschen. Ressourcen, auf die im Rahmen einer Transaktion zugegriffen wird (z. B. Datenbanktabellen) dürfen nicht für die gesamte Dauer der Transaktion (d. h. bis zum Empfang von `Close`) gesperrt bleiben. Vielmehr muss jede Aktion zu einer Transaktion protokolliert werden, um sie ggf. wieder rückgängig machen zu können (Empfang von `Compensate` nach dem Empfang von `Complete` bzw. dem Senden von `Completed`, aber vor dem Empfang von `Close`). Die Aspekte einer derartigen Kompensationsfähigkeit liegen außerhalb des Rahmens dieser Arbeit, es wird hier lediglich diese Fähigkeit als vorhanden angenommen.

4.2. Nichtfunktionale Anforderungen

Nach (Zimmermann u. a., 2003/2005, 3.5) sind bei der Entwicklung von dienstorientierten, Web-Service-basierten Lösungen folgende wichtige nichtfunktionale Anforderungen zu beachten:

1. Robustheit (4.2.1)
2. Leistungsfähigkeit (Perfomance, 4.2.2)
3. Skalierbarkeit (4.2.3)
4. Verfügbarkeit (4.2.4)
5. Portierbarkeit (4.2.5)

Diese Anforderungen sind bis auf Fehlertoleranz, einem Unterpunkt von Robustheit (s. u.), für diese Arbeit nur unter dem Gesichtspunkt interessant, dass sie auf mobile Umgebungen angewendet werden müssen. Konkret ist hier die in 3.5.1 beschriebene J2ME-Umgebung gemeint. Fehlertoleranz ist im Hinblick auf Transaktionen besonders wichtig, daher werden die einzusetzenden Spezifikationen in 4.2.1 daraufhin untersucht und entsprechende konkretere Anforderungen erarbeitet. Insbesondere werden dabei die für diese Arbeit relevanten Problemstellungen kleiner, mobiler Geräte (s. 3.5.2) betrachtet.

Für die Frameworkentwicklung sind nach (Schrörs, 2005, 4.2.4) weiterhin folgende Anforderungen zu betrachten:

6. Erweiterbarkeit (4.2.6)
7. Benutzbarkeit (4.2.7)
8. Vollständigkeit (4.2.8)

4.2.1. Robustheit

Unter Robustheit bzw. Zuverlässigkeit werden nach (Zimmermann u. a., 2003/2005, 3.5.4) Fehlertoleranz bzw. effiziente und effektive Fehlerbehandlungsmechansimen, Wartbarkeit sowie die Abwesenheit von schweren Softwarefehlern zusammengefasst.

Absolute Fehlerlosigkeit kann im Rahmen dieser Arbeit sicher nicht erwartet werden. Wartbarkeit soll durch erweiterbaren, leicht verständlichen Code realisiert werden. Ein weiterer Aspekt der Wartbarkeit, nämlich die Anforderung, dass das System problemlos neu gestartet werden kann, wird im Rahmen von Fehlertoleranz und Kompensation von Crash-Fehlern unten in 4.2.1.4 behandelt.

Ein fehlertolerantes System muss nach (Tanenbaum und van Steen, 2002, 7.1.1)[6] u.a. folgende wichtige Eigenschaften haben:

- Verfügbarkeit: Das System arbeitet in einem bestimmten Anteil einer beliebigen Zeitspanne zu jedem Zeitpunkt korrekt.
- Zuverlässigkeit: Das System arbeitet für eine Zeitspanne bestimmter Länge ohne Unterbrechung korrekt.
- Sicherheit (im Sinne von engl. Safety): Wenn das System für eine bestimmte Zeit ausfällt, hat dies keine gravierenden Folgen (z. B. Verlust kritischer Daten oder Inkonsistenzen).

[6] Die Klassifizierungen von nichtfunktionalen Anforderungen unterscheiden sich in (Zimmermann u. a., 2003/2005) und (Tanenbaum und van Steen, 2002) ein wenig, was aber für diese Arbeit nicht von Bedeutung ist.

- Wiederherstellbarkeit: Wenn das System nicht funktioniert, kann der Fehler leicht bzw. automatisch behoben werden.

Im Falle mobiler Systeme, auf denen mobile Web Services laufen und die damit eine Server-Rolle spielen, muss aus den in 3.5.2 aufgeführten Gründen von beeinträchtigter Verfügbarkeit und Zuverlässigkeit ausgegangen werden. Es ist daher umso wichtiger, dass die Eigenschaften Sicherheit und Wiederherstellbarkeit betont werden.

Dabei müssen für das Transaktions-Framework zwei Aspekte betrachtet werden:

1. Kompensation von Fehlern bei der Übertragung von Nachrichten (Fehler beim Senden sowie Auslassungsfehler und Zeitüberschreitungsfehler nach (Tanenbaum und van Steen, 2002, 7.1.2) aus Sicht von Sender und Empfänger).
2. Kompensation von Zeiten, in denen ein Gerät ausgeschaltet oder sich im „Stand by"-Modus befindet, sowie von Systemausfällen (Crash-Fehler nach (Tanenbaum und van Steen, 2002, 7.1.2) aus Sicht des ausgefallen Systems).

4.2.1.1. Übertragungsfehler

Übertragungsfehler lassen sich wie folgt unterteilen (vgl. (Tanenbaum und van Steen, 2002, 7.1.2)).

- Sendefehler treten beim Senden einer Nachricht auf. Der Absender wird direkt über den Fehler informiert und es ist sicher, dass der Empfänger die Nachricht nicht erhalten hat.
- Auslassungsfehler bezeichnen den Verlust von Nachrichten. Der Absender kann nicht direkt informiert werden, allerdings kann indirekt ein Zeitüberschreitungsfehler auftreten, da keine Antwort erfolgt. Es ist beim Absender nicht bekannt, ob der Empfänger die Nachricht erhalten und verarbeitet hat.
- Zeitüberschreitungsfehler bezeichnen das Überschreiten einer bestimmten Wartezeit beim Warten auf die Antwort zu einem Aufruf. Die Ursache können Sendefehler (beim Empfänger der Ursprünglichen Nachricht) oder Auslassungsfehler (sowohl die Ursprüngliche Nachricht als auch die Antwort darauf können verloren gehen) sein. Es ist nicht bekannt, ob der Empfänger die Ursprüngliche Nachricht erhalten und verarbeitet hat.

Bei der Behandlung von Fehlern ist zwischen dem reinen Nachrichtentransport und der Anwendung zu unterscheiden. Durch die Verwendung eines asynchronen Kommunikationsmodells auf Anwendungsebene wird bereits die Tatsache adressiert, dass die Verarbeitung

empfangener Nachrichten lange dauern kann. Dies kann nicht nur durch die Art des Workflows bedingt sein, sondern auch durch eingeschränkte Rechenleistung auf Seiten des mobilen Gerätes. Es gilt nun, zusätzlich Fehler bei der Übertragung von Nachrichten zu kompensieren. Dazu muss das zugrunde liegende Kommunikationsprotokoll betrachtet werden.

Wird ein asynchrones Protokoll (z. B. UDP) verwendet, so lassen sich Aufrufe und Nachrichtenempfang direkt von der Anwendung auf das Kommunikationsprotokoll übertragen.

Wird ein synchrones Protokoll verwendet (z. B. TCP), so gehört zu einem Aufruf auch immer eine Bestätigung bzw. eine Antwort. Zur Unterstützung von asynchroner Kommunikation auf Anwendungsebende sollte der Empfang von Nachrichten positiv bestätigt werden bevor die Verarbeitung beginnt. Die eigentliche Antwort wird dann asynchron als Nachricht in umgekehrter Richtung gesendet.

Das zu erweiternde Framework basiert auf HTTP über TCP/IP und WLAN. HTTP über TCP ist synchron, und um asynchrone Anwendungen darauf aufzubauen, wird das Empfangen von Nachrichten zunächst automatisch mit „200 OK“ und einer leeren Nachricht bestätigt.

Sendefehler können für synchrone Transportprotokolle weiter unterteilt werden, so dass man zu folgender Liste direkt vom Absender einer Nachricht beobachtbarer Fehler kommt:

1. Konfigurationsproblem: Es tritt ein Fehler auf, der nicht mit dem Sendevorgang zusammenhängt, sondern mit der Vorbereitung (z. B. Fehler beim Herausfinden der physikalischen Zieladresse für den Aufruf).
2. Konnektivitätsproblem: Es tritt ein Fehler beim tatächlichen Absenden der Nachricht auf.
3. Negative Bestätigung: Die Nachricht wird zwar versendet, aber der Empfänger meldet einen Fehler (z. B. HTTP „500 ...“). Die Nachricht wurde in diesem Fall nicht an die Anwendungslogik des Empfängers ausgeliefert.
4. Keine Bestätigung (Zeitüberschreitung): Es wird keine synchrone Antwort empfangen und es ist daher nicht bekannt, ob die Nachricht beim Empfänger verarbeitet wurde oder nicht.

4.2.1.2. Wiederholungen bei Übertragungsfehlern

In bestimmten Fällen kann es sinnvoll sein, einen gescheiterten Übertragungsversuch zu wiederholen. Insbesondere im Falle von kurzzeitigen Empfangsstörungen durch Access-Point-Handovers (WLAN), Zellwechsel (GSM, GPRS, UMTS), o. Ä., die manchmal trotz Einsatz entsprechender Protokolle (z. B. IP Mobility Support nach RFC 2002) zu lange dauern, um erfolgreich vor höher liegenden Netzwerkschichten verborgen zu werden, kann schon

der nächste Versuch wieder erfolgreich sein. Es ist die Aufgabe des Frameworks, solche Wiederholungen möglichst konfigurierbar und transparent für die Anwendungslogik zu realisieren.

Im Falle von Fehlerart 1 (z. B. bei Problemen beim Herausfinden der physikalischen Adresse des Empfängers, hervorgerufen durch Konfigurationsprobleme des Service-Repositories bzw. der Service-Registry) kann direkt ein Fehler zurückgemeldet werden, da eine Wiederholung des Sendeversuchs keine anderen Ergebnisse bringen würde.

Treten Fehler der Arten 2, 3 oder 4 auf, so könnte das Senden bis zu einer konfigurierbaren maximalen Anzahl von Versuchen wiederholt werden. Im Falle von HTTP-Zeitüberschreitungen kann dies z. B. eine Zeit von einigen Sekunden bis Minuten in Anspruch nehmen.

Deuten die Fehler darauf hin, dass der Empfänger offline ist, so könnten weiterhin fest installierte Gateways benutzt werden, welche immer online sind. Die Zieladresse des Aufrufs wird wie üblich im `To`-Header übertragen, der Aufruf jedoch an ein Gateway gesendet, welches die Nachricht puffert und innerhalb eines konfigurierbaren Zeitraums versucht, die Nachricht auszuliefern. Dem Absender wird vom Gateway der erfolgreiche Empfang der Nachricht (synchron) signalisiert, so dass angenommen werden kann, das die Nachricht garantiert beim Empfänger ausgeliefert werden wird.[7]

Bei jeder Wiederholung des Sendeversuchs können erneut Fehler auftreten. Dies führt unabhängig davon, ob ein Gateway benutzt wird oder nicht, letztendlich dazu, dass irgendwann nicht mehr bekannt ist, ob die Nachricht vom Empfänger gar nicht, einmal, oder mehrmals empfangen und verarbeitet wurde.

Wünschenswert wäre dagegen ein „Exactly Once“- oder zumindest ein „At Least Once“-Auslieferungsmodell (Tanenbaum und van Steen, 2002, 7.3). Ansätze wie WS-ReliableMessaging (WS-RM, 2005) und WS-Reliability (WS-Rel, 2004) versuchen, auf der Anwendungsschicht ein „Exactly Once“-Auslieferungsmodell umzusetzen, in dem eine Nachricht maximal einmal ausgeliefert wird und im Falle der Nichtauslieferung mindestens eine der beiden Seiten eine entsprechende Rückmeldung erhält. Diese Ansätze hier zu betrachten würde allerdings zu weit führen.

Es ist daher zu prüfen, ob die Protokolle WS-Coordination und WS-BusinessActivity robust genug sind, um dieses Problem zu kompensieren. WS-Coordination kennt nur zwei Anwendungsfälle: Aktivierung und Registrierung. Bei der Aktivierung, der Erzeugung des Transaktionskontexts, kann einfach der erste Kontext verwendet werden, der vom Aktivierungs-Service auf die Anfrage `CreateCoordinationContext` erzeugt und mit `CreateCoordinationContextResponse` zurückgeschickt wird. Duplizierte Antworten können

[7] Da der To-Header eine logische Zieladresse enthält, kann auf diese Weise auch die Verantwortung für die Ermittlung der physikalischen Adresse an das Gateway delegiert werden.

verworfen werden. Bleibt die Antwort dagegen aus, so muss das Framework nach einer bestimmten Zeit den Aktivierungsversuch wiederholen oder einen Fehler melden.

Bei der Registrierung eines Teilnehmers mittels `Register` kann der Teilnehmer nicht fortfahren bis `RegisterResponse` vom Registrierungs-Service empfangen wird. Ein Teilnehmer kann laut Spezifikation im Rahmen eines Kontexts nur einmal für ein Protokoll registriert sein. Für duplizierte `Register`-Nachrichten ist der SOAP-Fault `wscoor:AlreadyRegistered` vorgesehen, der vom Teilnehmer ignoriert werden kann, wenn bereits `RegisterResponse` empfangen wurde. Bleibt die Antwort jedoch für bestimmte Zeit aus, so muss das Framework den Registrierungsversuch wiederholen. Eine duplizierte `RegisterResponse`-Nachricht kann vom Teilnehmer ignoriert werden. Der Koordinationszustand zwischen Koordinator und Teilnehmer kann sich also von `INIT` nur über den Zwischenzustand `REGISTERING` auf `REGISTERED` ändern.

Im Falle von WS-BusinessActivity findet ein Statuswechsel ebenfalls immer nur über den Umweg eines Zwischenzustands statt, in dem auf eine Quittung gewartet wird (z.B. `ACTIVE` → `Complete` → `COMPLETING` → `Completed` → `COMPLETED`). Wenn nun eine Seite sicher ist, dass eine Nachricht nicht ausgeliefert wurde, so kann aus dem Zwischenzustand wieder in den ursprünglichen Zustand übergegangen werden und die Nachricht noch einmal gesendet werden. Im Falle einer doppelten Auslieferung sieht das Protokoll in allen Status vor, dass auf eine Nachricht folgende gleichartige Nachrichten ignoriert werden und der Status nicht verändert wird.

Abb. 4.4 illustriert die unterschiedlichen Fälle. Links oben wurde eine Antwort nicht ausgeliefert. Der Absender wurde (auf Transportebene) unmittelbar über die gescheiterte Auslieferung informiert und kann noch rechtzeitig wiederholen. Links unten wird eine Nachricht scheinbar ausgeliefert, aber auf der Gegenseite nicht verarbeitet. Der Zustand wird nicht geändert. Der Koordinator erhält einen Timeout (auf Anwendungsebene), woraufhin der Sendeversuch wiederholt wird.

Rechts oben wurde eine Nachricht des Koordinators scheinbar nicht ausgeliefert, da die Bestätigung (auf Transportebene) verloren gegangen ist. Der Teilnehmer verarbeitet die Nachricht jedoch, schickt eine entsprechende Antwort und ändert den Zustand. Eine in diesem Zustand empfangene Wiederholung der ursprünglichen Nachricht führt zu einem wiederholten Senden der Antwort. Auf der Seite des Koordinators wird die wiederholte Antwort dann ignoriert, sofern sie in dem Zustand eintrifft, der durch das Empfangen des ersten Versuchs herbeigeführt wurde (`COMPLETED`). Ist die Verarbeitung jedoch schon weiter fortgeschritten (`CLOSING`, rechts unten) und trifft die zweite Antwort erst dann ein, so wird sicherheitshalber die `Close`-Nachricht wiederholt (gestrichelte Pfeile). Auf Seiten des Empfängers wird mit der Wiederholung bei zu spätem Eintreffen genauso verfahren (würde das wiederholte `Close` im Zustand `CLOSING` empfangen, so würde es ignoriert werden).

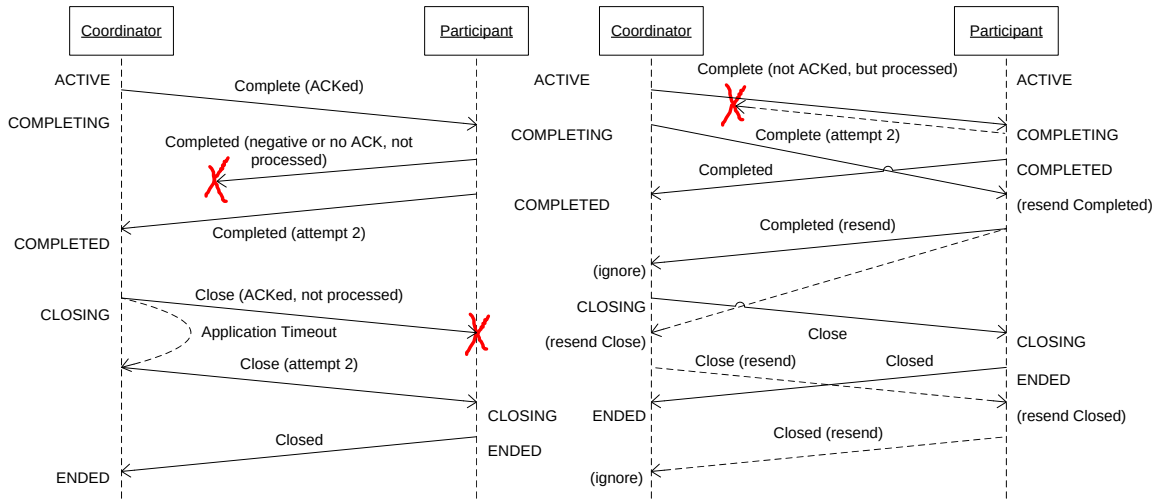

Abbildung 4.4.: Verhalten von WS-BusinessActivity bei Übertragungsfehlern (eigene Darst.)

Die Untersuchung zeigt, dass duplizierte Nachrichten nicht zu Inkonsistenzen führen, da je nach Zustand ein Duplikat entweder ignoriert werden kann oder erneut eine Quittung bzw. die vor Empfang des Duplikats gesendete Nachricht gesendet werden kann. Für den Verlust von Nachrichten gibt es dagegen kritische Phasen sowohl auf Koordinator- als auch auf Teilnehmerseite. Dies sind die Zustände, in denen auf Quittungen gewartet wird (`COMPLETING`, `CLOSING`, `CANCELING` und `COMPENSATING` auf Koordinatorseite, `EXITING` und `FAULTING_...` auf Teilnehmerseite). Hier muss, wie auf der linken Seite in Abb. 4.4 gezeigt, vom Framework mit Timeouts gearbeitet werden. Für jeden asynchronen Aufruf, der quittiert werden muss, muss ein Timer gestartet werden. Bei Ablauf des Timers vor Eintreffen der Quittung (Timeout) wird entschieden wie weiter verfahren werden soll.

Unter Einsatz von Timern sind WS-Coordination und WS-BusinessActivity robust genug, um vereinzelte Übertragungsfehler und Störungen zu kompensieren. Ein entsprechendes Timer-API wird in 4.3.2 spezifiziert. Ein Timer kommt zum Einsatz, sobald die Übertragung der Nachricht erfolgreich war und dann auf eine entsprechende asynchrone Antwort gewartet wird. Da die jeweiligen Operationen von langer Dauer sein können, insbesondere wenn mobile Empfänger länger offline sind und Gateways (s. o.) eingesetzt werden, sind die Timeout-Zeiten entsprechend zu wählen.

Es ist anzumerken, dass die Korrelierungsmechanismen von WS-Addressing (`MessageId`- und `RelatesTo`-Header) nicht unbedingt zur Erkennung von Fehlern benötigt werden, da dazu die Kombination aus aktuellem Protokollzustand und empfangener Nachricht ausreichend ist. Das hat den Vorteil, dass nicht unbedingt eine Nachrichtenhistorie geführt werden muss.

4.2.1.3. Zeitüberschreitungen auf Anwendungsebene

Auch eine auf dem Transaktions-Framework aufbauende Anwendung muss Zeitüberschreitungen behandeln. Dies betrifft sowohl die Framework-Logik als auch die Anwendungslogik. Die fünfte Fehlerart ist also:

5. Keine asynchrone Antwort auf eine Nachricht mit den WS-Addressing SOAP-Headern `MessageId` und `ReplyTo`: Die Nachricht wurde zwar ausgeliefert, aber es ist nicht bekannt, bis zu welchem Schritt sie beim Empfänger verarbeitet wurde.

Alle asynchronen Aufrufe (mit `MessageId` und `ReplyTo` WS-Addressing SOAP-Headern), auf die eine Antwort durch ein asynchrones Callback (mit `RelatesTo`-Header) erwartet wird, müssen daher mit einem Timeout-Mechanismus arbeiten, der nach einer frei wählbaren Zeitspanne signalisiert, dass die ursprüngliche Nachricht zwar ausgeliefert wurde, aber bisher keine Antwort empfangen wurde. Die Zeitspanne sollte ausreichend groß gewählt werden, um Verzögerungen beim Senden (s. o.) zu berücksichtigen. Die Zeitspanne ist aber letztendlich anwendungsabhängig, je nach Art des ausgeführten Workflow-Schrittes kann sie unter Umständen einige Stunden, Tage, Wochen oder sogar Monate betragen. Wenn kein Timeout auftritt, muss beim Empfang der Nachricht mit `RelatesTo`-Header der Timer für diese `MessageId` deaktiviert werden. Timeouts können auch benutzt werden, um regelmäßig zu prüfen, ob schon ein Ergebnis vorliegt.

Der Geschäftsprozess-Service (Koordinator) im Freigabemanager-Beispiel könnte die Zeitspanne so wählen, dass ein Timeout beim Überschreiten der Frist für die Freigabeentscheidung durch den Freigabemanager ausgelöst wird. Die Anwendungslogik könnte dann z. B. einen alternativen Prozess anstoßen oder die Anfrage wiederholen.

In WS-Coordination ist ein SOAP-Fault `wscoor:NoActivity` vorgesehen, der im Falle eines Timeouts vom Koordinator an den Teilnehmer gesendet werden kann. Der umgekehrte Fall ist nicht vorgesehen, ist aber auch nicht so wichtig, da der Koordinator die steuernde Logik ausführen sollte. Im Falle des WS-BusinessActivity-Protokolls sind die Teilnehmernachrichten, auf die der Koordinator reagieren muss lediglich `Exit` und `Fault`. In beiden Fällen ist ein Timeout beim Warten auf die Quittung `Exited` bzw. `Faulted` nicht kritisch, da das Protokoll nach der Quittung in jedem Fall beendet wäre (s. 4.1.2.5).

Die notwendige Timerverwaltung kann man sich als anwendungsunabhängige Komponente des Frameworks vorstellen. Sie wird im Rahmen dieser Arbeit lediglich skizziert (s. 4.3.2, 5.2.2). Die Anforderung besteht darin, dass beim asynchronen Aufrufen eines Web Services ein Timer gestartet wird, der im Falle von Anwendungs-Timeouts (dies umfasst auch Timeouts für Nachrichten, die von der Framework-Logik versendet wurden) die entsprechenden Callback-Methoden auf einem beim Aufruf angegebenen Interface aufruft.

Mit einem Timer müssen folgende Informationen gespeichert werden, um die o. g. Schritte zu ermöglichen: Die Zeit des Timeouts, die `MessageId` des Aufrufs sowie das im Fehlerfall aufzurufende Callback-Interface. Die `MessageId` muss dabei auch an die Framework-Logik bzw. die Anwendungslogik zurückgegeben werden, damit dort die für die weitere Verarbeitung oder für einen Timeout notwendigen Informationen zugeordnet werden können.

4.2.1.4. Crash-Fehler

Das Transaktions-Framework sollte die Wiederherstellung (Recovery) nach einem Crash unterstützen: Startet ein System nach einem Crash neu und sind noch Transaktionsinformationen im persistenten Speicher, so müssen diese ausgewertet werden und je nach Rolle des Systems (Koordinator, Teilnehmer) verfahren werden. Als Crash wird dabei auch das vom Benutzer gewollte Ausschalten des Geräts sowie ein gewollter oder ungewollter Übergang in einen „Stand by“-Modus zwecks Stromsparens bezeichnet.

Wiederherstellungsfähigkeit ist nicht nur ein Fehlertoleranzkriterium, sondern fällt auch unter Wartbarkeit.

Die Implementierung der Wiederherstellung ist eine umfangreiche Aufgabe. Sie kann im Rahmen dieser Arbeit daher nur skizziert und im Entwurf angedeutet werden:

Beim Neustart der transaktionalen Web Services muss zunächst mittels `GetStatus`-Nachrichten für vorhandene Transaktionen geprüft werden, in welchem Status sich das Protokoll befindet. Ein Koordinator muss dabei alle Teilnehmer befragen, ein Teilnehmer nur den Koordinator. Die daraufhin empfangenen `Status`-Nachrichten müssen ausgewertet werden. Für jedes Koordinator-Teilnehmer-Paar muss der Status aus der Sicht von Koordinator und Teilnehmer gleich sein. Aufgrund der o. g. Eigenschaft der Protokolle, dass immer mit Quittungen gearbeitet wird, können sich die Status von Koordinator und Teilnehmer nicht um mehr als einen Schritt unterscheiden. Der Status wird dann bei dem Partner, der die Wiederherstellung durchführt, angepasst.

Auch für die Geschäftsanwendung kann es erforderlich sein, dass nach der Wiederherstellung des Transaktionsstatus eine Wiederherstellung stattfindet. Im Laufe dieser könnte entschieden werden, dass eine weitere Verarbeitung nicht möglich ist. Es müssten dann entsprechende Nachrichten (`Fault`, `Cancel`, `Compensate`) an den Koordinator bzw. alle Teilnehmer geschickt werden.

Weiterhin müssen alle nichtabgelaufenen Anwendungs-Timer neu gestartet werden und für abgelaufene Timer die entsprechenden Callbacks aufgerufen werden.

Die Wiederherstellung kann über spezielle Recovery-Interface-Methoden der Framework- und der Anwendungslogik geschehen, die aufgerufen werden sobald die Web-Service-Umgebung (wieder) gestartet ist.

4.2.2. Leistungsfähigkeit

Nach (Zimmermann u. a., 2003/2005, 3.5.1) ist SOAP der kritische Punkt bei Web Services Perfomance. Die Verarbeitung von XML-Nachrichten durch generische Laufzeitinfrastrukturen wie SOAP generieren einen nicht zu vernachlässigenden Overhead, der Web Services für gewissen Anwendungsklassen (z. B. Real-Time-Anwendungen) untauglich macht. Auf kleinen mobilen Geräten mit begrenzter Rechen- und Speicherkapazität ist umso mehr auf eine effiziente XML-Verarbeitung gemäß SOAP und den darauf aufbauenden Spezifikationen zu achten.

Das in (Schrörs, 2005) und (Schrörs, 2006a) entwickelte Framework benutzt bereits derartig schlanke Bibliotheken. Weiterhin sollten nur die nötigsten Elemente der in WS-Coordination, WS-BusinessActivity und WS-Addressing definierten Datentypen verwendet werden (s. a. 4.1.4, 4.1.5), welche ohnehin nicht komplex sind. So bestehen die meisten der WS-BusinessActivity-Nachrichten nur aus einem Element.

Aufgrund der Tatsache, dass viele mobile Geräte über Akkus betrieben werden, sollte die von mobilen Web Services und damit auch vom Transaktions-Framework genutzte Rechenleistung so gering wie möglich ausfallen. Dies vermindert das Risiko von Ausfällen.

Der nichtflüchtige Speicher kann je nach Gerätetyp unterschiedliche Leistungsdaten (Typ, Größe, Zugriffszeit) aufweisen. In der Regel muss von einer geringen Größe ausgegangen werden. Die Zugriffszeiten scheinen zudem von Gerät zu Gerät zu variieren, einen inoffiziellen Benchmark enthält (MIDP-Bench, 2006). Es ist empfiehlt sich daher, nur absolut notwendige Informationen zu speichern.

4.2.3. Skalierbarkeit

Es ist davon auszugehen, dass Anwendungen auf kleinen mobilen Geräten aufgrund beschränkter Leistungsfähigkeit nicht gut skalieren werden. Übermäßig viele gleichzeitige Zugriffe auf mobile Web Services zu fordern, scheint daher nicht angemessen. Ein neuerer PDA oder ein Notebook mit WLAN-Anbindung wird einige gleichzeitige Verbindungen verkraften. Ein Mobiltelefon wird weit weniger gleichzeitige Zugriffe verarbeiten können, weshalb die geringere Bandbreite von Bluetooth-, GPRS- und UMTS-Netzwerken gegenüber WLAN nicht so ins Gewicht fällt.

Die unterschiedlichen Bandbreiten der einzelnen Netzwerke führen zu unterschiedlich langen Datenlaufzeiten. Dies muss ebenfalls bei der Formulierung von Skalierbarkeitsanforderungen beachtet werden.

Bezüglich der Batteriekapazität kleiner mobiler Geräte gilt das gleiche wie oben bereits für Leistungsfähigkeit erwähnt.

Optimimierungspotential bietet hier das Prinzip des „Web Service Scopes“ (Zimmermann u. a., 2003/2005, 3.5.2). Wenn ein Web Service reentrant ist (d. h. die Implementierung gleichzeitig thread-safe innerhalb mehrerer (Transaktions-)Kontexte ausgeführt werden kann), dann kann für die gesamte Applikation eine Instanz der Web-Service-Implementierung benutzt werden, ansonsten muss für jeden Request eine neue Instanz benutzt werden. Eine Zwischenlösung bilden Session-basierte Web Services, bei der eine Instanz für die Dauer einer Session für mehrere Aufrufe verwendet wird. Eine Session könnte z. B. mit einer Transaktions-ID assoziiert sein. Sessions werden jedoch nicht von der SOAP-Spezifikation behandelt. Im Hinblick auf beschränkte Speicherkapazität und Prozessorleistung sind Sessions mit Vorsicht zu behandeln, da sie (d. h. die zugehörigen Instanzen der Web-Service-Implementierungen) persistent gespeichert bzw. serialisiert werden müssen, um Crash-Fehler (s. o.) zu überdauern.

Mobile Web Services und damit auch das Transaktions-Framework sollten daher reentrant ausgelegt werden. Die benutzte Instanz einer Web-Service-Implementierung kann nach Crash-Fehlern einfach neu erzeugt werden. Dies bedeutet, dass Request-spezifische Daten wie z. B. die WS-Coordination- und WS-Addressing SOAP-Header bei jedem Methodenaufruf der Implementierung als Parameter übergeben werden müssen.

4.2.4. Verfügbarkeit

Verfügbarkeit ist einerseits ein Aspekt von Fehlertoleranz, welche in Abschnitt 4.2.1 behandelt wurde, wenn damit gemeint ist, wie oft und wie lange ein System funktionsfähig ist. Andererseits kann mit Verfügbarkeit auch gemeint sein, auf welchen Plattformen Software (sinnvoll) einsetzbar und lauffähig ist, was wiederum von der Portierbarkeit (s. 4.2.5) abhängt. Die Rahmenbedingungen für diese Arbeit setzen J2ME mit CLDC 1.1 und MIDP 2.0 voraus (s. 3.5.1), eine Laufzeitumgebung, die sehr verbreitet auf mobilen Geräten zu finden ist. Auch andere J2ME-Konfigurationen und -Profile können mit Einschränkungen benutzt werden, siehe dazu Abschnitt 6.1.

4.2.5. Portierbarkeit

Portierbarkeit betrifft im Falle von mobilen Web Services die Fähigkeit, auf verschiedenen Systemen eingesetzt werden zu können, ohne dass sie neu implementiert werden müssen. Transaktionale mobile Web Services, wie sie in dieser Arbeit betrachtet werden, benötigen auf anderen Systemen identische Konfigurationen, also J2ME mit der richtigen Konfiguration und dem richtigen Profil sowie das erweiterte Web Services Framework aus (Schrörs, 2006a) (s. 3.5.1) und die in dieser Arbeit beschriebenen Erweiterungen.

4.2.6. Erweiterbarkeit

Erweiterbarkeit betrifft zwei Aspekte: Das Framework sollte zum Einen erweiterbar bezüglich anderer Protokolle für langlebige Transaktionen und anderer Transaktionstypen (z. B. ACID-Transaktionen mit WS-AtomicTransaction) sein. Zum Anderen ist eine gute Erweiterbarkeit des Frameworks um Anwendungskomponenten, d. h. eine möglichst einfache Implementierung von Web Services mit Hilfe des Frameworks, gefordert.

4.2.7. Benutzbarkeit

Nach (Schrörs, 2005) kann die Benutzbarkeit eines Systems durch geeignete Werkzeuge und intuitiv verständliche Schnittstellen gefördert werden. Ein Beispiel hierfür sind Code-Generatoren zur Erstellung von Grundgerüsten für Anwendungen. Im Falle von Web Services sind dies Stubs auf Client-Seite, Skeletons auf Server-Seite sowie Klassen zur Serialisierung und Deserialisierung von Datentypen.

Um kompatibel zu WS-Coordination und WS-BusinessActivity zu sein, benötigen Web Services einerseits gewisse Protokolloperationen in ihrem Interface, andererseits für jede Operation der Anwendungslogik bestimmten Code, der zur Einhaltung der Protokolle beiträgt. Dieser Code kann unter Angabe der notwendigen Parameter größtenteils generiert werden. Dies wird in 4.4 beschrieben.

4.2.8. Vollständigkeit

Um produktiv eingesetzt werden zu können, muss ein System vollständig bezüglich des von ihm zu erfüllenden Zwecks sein. Im Falle des Transaktions-Frameworks müssen also die Spezifikationen WS-Coordination und WS-BusinessActivity in allen Einzelheiten unterstützt bzw. eingehalten werden. Dem Entwickler der Anwendungslogik müssen entsprechende Komponenten und Einsprungpunkte (d. h. Möglichkeiten zum Eingreifen ins Protokoll) zur Verfügung gestellt werden. Die Vollständigkeit dient hier vor allem der Interoperabilität von verteilten Systemen.

4.3. APIs

Aufgrund der Überlegungen in den vorangehenden Abschnitten werden im Folgenden notwendige APIs in Form von konzeptionellen Interface- und Klassendiagrammen dargestellt.

4.3.1. WS-C & WS-BA API

Das Framework sollte dem Entwickler der Geschäftslogik möglichst viel Arbeit bei der Erzeugung von Transaktionen, der Registrierung von Teilnehmern sowie der Einhaltung der Terminierungsprotokolle abnehmen. Das Transaktionsmanagement sollte auch möglichst transparent für die Anwendungslogik stattfinden. Es hat sich allerdings gezeigt, dass im Falle von WS-BusinessActivity die Anwendungslogik für die Einhaltung der Protokolle mitverantwortlich sein muss (s. 4.1.3). Dazu muss dem Entwickler die Möglichkeit gegeben werden, im Rahmen der Anwendungslogik auf die Ereignisse im Rahmen der WS-BusinessActivity Terminierungsprotokolle zu reagieren. Dem Entwickler müssen also die Operationen des Protokolls zur Verfügung stehen.

Abb. 4.5 zeigt die benötigten Interfaces. Auf Koordinator-Seite dient das Interface `Coordinator` dazu, Zugriff auf alle registrierten Teilnehmer zu ermöglichen. Ein einzelner Teilnehmer wird über das Interface `BACParticipant` angesprochen, welches die Operationen enthält, die vom Koordinator an eine Teilnehmer gesendet werden können. Außerdem kann neben Service-Endpoint und -Name auch der registrierte Protokolltyp sowie der Status des Protokolls zwischen Koordinator und Teilnehmer aus Koordinatorsicht abgefragt werden. Über das Interface `BACoordinatorListener` wird die Anwendungslogik vom Transaktionsframework über Protokollnachrichten von einzelnen Teilnehmern informiert.

Auf Teilnehmerseite stehen mit dem Interface `BAPParticipant` die Operationen zur Verfügung, die an den Koordinator gesendet werden können. Der Status des Protokolls ist aus Teilnehmersicht verfügbar. Zusätzlich kann der Koordinationsstatus abgefragt werden, um (im Falle von asynchron eintreffenden Operationsaufrufen innerhalb derselben Transaktion) feststellen zu können, ob ein Teilnehmer gerade erst aufgerufen wurde, die Registrierung beim Koordinator gerade läuft oder ob der Teilnehmer bereits registriert ist. Über das Interface `BAParticipantListener` wird die Anwendungslogik über Protokollnachrichten des Koordinators informiert.

Die Interfaces mit dem Prefix `BA` erweitern dabei die Interfaces, die die Funktionalität von WS-Coordination (Aktivierung, Registrierung) zur Verfügung stellen, um die Funktionalität von WS-BusinessActivity. Der Rückgabewert der Protokolloperationen, auf die eine Antwort erwartet wird (`complete()`, `close()`, `cancel()`, `compensate()`, `exit()`, `fault()` und `getStatus()`), ist die für die Nachricht verwendete `MessageId` (s. 4.1.5). Für Protokollnachrichten, die Antworten auf eine vorangehende Nachricht (Anfrage) sind, muss als Parameter `relatesTo` die `MessageId` der Anfrage übergeben werden. Die Methode `completed()` zum Senden der `Complete`-Protokollnachricht an den Koordinator wird dabei im Falle des Protokolltyps ParticipantCompletion ohne Parameter, d. h. nicht als Antwort auf eine `Complete`-Nachricht benutzt, im Falle von CoordinatorCompletion dagegen mit der `MessageId` der `Complete`-Nachricht als Parameter

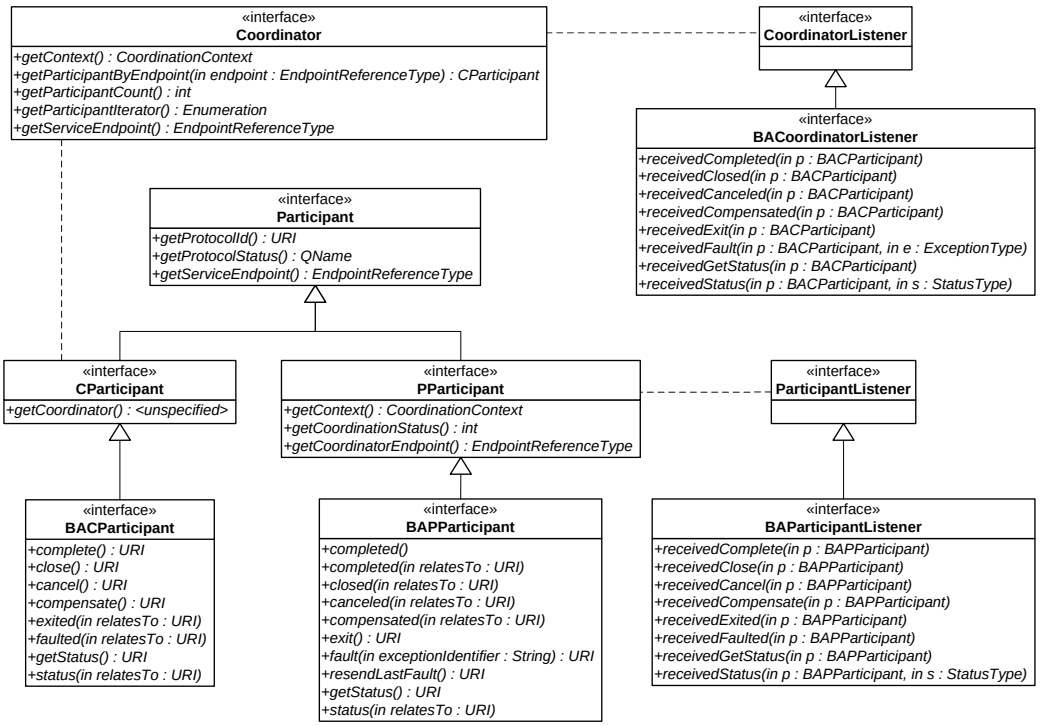

Abbildung 4.5.: WS-BusinessActivity API

`relatesTo`. Die Framework-Logik ist dafür verantwortlich, dass die jeweils richtige Methode benutzt wird.

4.3.2. Timer API

Die in Java2 seit der Version J2SE 1.3 sowie in MIDP verfügbaren Klassen `java.util.Timer` und `java.util.TimerTask` können für die Umsetzung von Timeout-Mechanismen herangezogen werden. Ein Timer wird gestartet, indem man eine `Timer`-Instanz erzeugt und dieser über eine ihrer `schedule()`-Methoden die Zeit des Timeouts sowie eine Instanz der Klasse `TimerTask` übergibt, deren `run()`-Methode bei Ablauf des Timers ausgeführt wird (s. (Java API, 2006)).

Abb. 4.6 veranschaulicht, wie eine Web-Service-Implementierung über einen Timeout benachrichtigt werden kann. Beim asynchronen Aufruf eines anderen Web Services wird für die `MessageId` der Nachricht ein Timer gestartet. Trifft vor Ablauf des Timers eine Nachricht ein, die diese ID als `RelatesTo`-Header enthält, so wird der Timer deaktiviert. Trifft eine solche Nachricht nicht vor Ablauf des Timers ein, so wird der aufrufende Web Service wie beschrieben benachrichtigt.

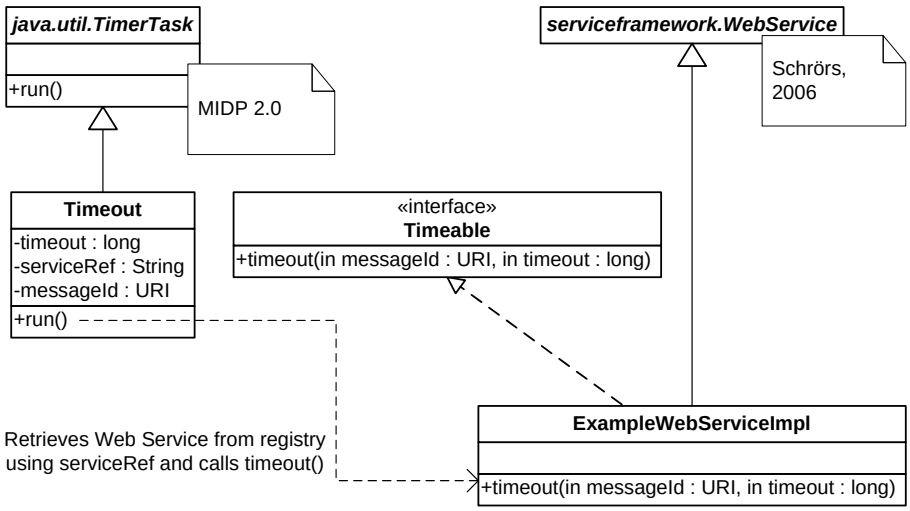

Abbildung 4.6.: Timer API

Es wird kein Callback-Interface bzw. Callback-Objekt anstelle der Web-Service-Referenz vorgesehen, da die Timer-Informationen persistent gespeichert werden müssen. Ein Callback-Objekt müsste dazu komplett serialisiert werden. Nur die Referenz auf den Web Service innerhalb seines SOAP-Servers zu speichern ist dagegen wesentlich einfacher.

4.3.3. WS-Addressing und Call API

Zum Aufrufen von Web Services wird in (Schrörs, 2006a) ein API ähnlich der Klasse `Call` aus JAX-RPC (JSR-101, 2003) bereitgestellt. Ein Objekt der Klasse `soap.Call` wird über Setter-Methoden mit den notwendigen Informationen versorgt, bevor schließlich eine Methode zur Ausführung des Aufrufs aufgerufen wird. Um dem Aufruf die benötigten SOAP-Header für WS-Addressing und WS-Coordination möglichst einfach hinzufügen zu können, sollten zwei Erweiterungen dieser Klasse wie in Abb. 4.7 angedeutet bereitgestellt werden.

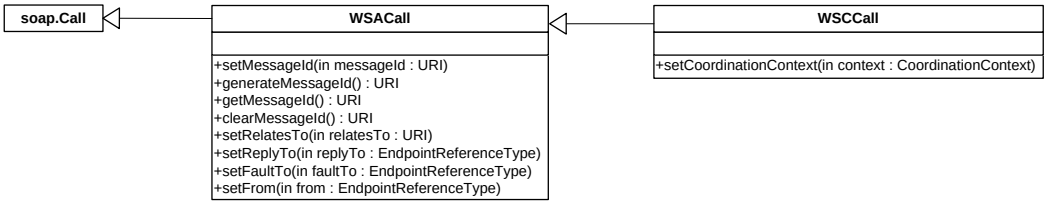

Abbildung 4.7.: Call API

4.3.4. Persistenz API

Um die in 4.1.6 aufgelisteten Informationen zwischen asynchronen Aufrufen zwischenspeichern zu können, sollte ein möglichst einfaches API zum Einsatz kommen. Es genügt, beliebige Informationen unter eindeutigen IDs ablegen zu können. Implementierungen können, wie in Abb. 4.8 angedeutet, flüchtigen oder nicht-flüchtigen Speicher nutzen.

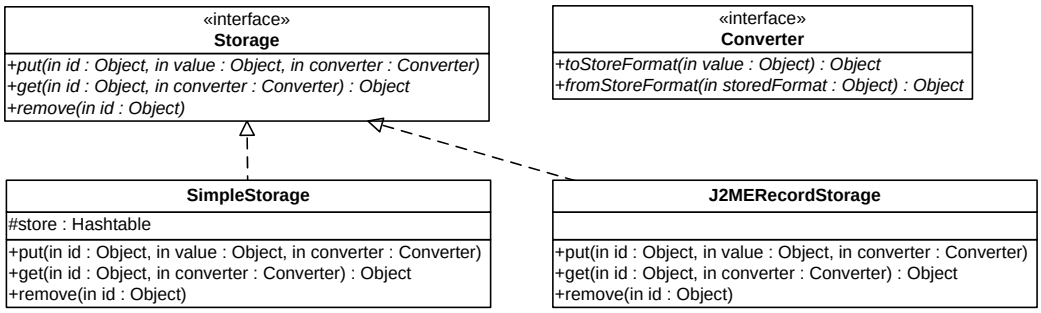

Abbildung 4.8.: Persistenz API

`SimpleStorage` verwendet einfach eine `java.util.Hashtable`, `J2MERecordStorage` würde MIDP RecordStores (s. (Li und Knudsen, 2005, Kap. 8)) benutzen. Um die verschiedenen Entitäten in das Speicherformat umzuwandeln und aus dem Speicherformat wieder zurück zu wandeln, werden Konvertierungsobjekte (Implementierungen von `Converter`) benötigt. Generische bzw. automatische Konvertierungsmechanismen wie etwa Objektserialisierung werden hier aufgrund ihrer Komplexität bewusst nicht vorgesehen. Prinzipiell könnten die `Storage`-Implementierungen und Konverter für die Top-Level-Entitäten (Operation, Coordinator, Participant, Timer) jedoch beliebige in der jeweiligen Laufzeitumgebung verfügbare Persistenz-Mechanismen verwenden.

4.3.5. Web Service Interfaces

Transaktionale Web Services müssen die Protokolle WS-Coordination und WS-BusinessActivity einhalten. Dazu benötigen die Web Services die entsprechenden Operationen in ihren Interfaces. Für die Operationen der WS-Coordination Aktivierungs- und Registrierungs-Services können entweder zentrale Services benutzt werden oder die Operationen können in die Interfaces der einzelnen Services aufgenommen werden, so dass jeder Koordinator seinen eigenen Aktivierungs- und Registrierungs-Service betreibt und jeder Teilnehmer seinen eigenen lokalen Registrierungs-Service, der dann den Registrierungs-Service des zugehörigen Koordinators aufruft. Der Einfachheit halber sollten die Operationen in die Interfaces der einzelnen Services aufgenommen werden (s. a. 5.2.5). Eine allgemeine

Diskussion über die Aggregation von Web Services findet sich in (Khalaf und Leymann, 2003).

Abb. 4.9 zeigt mögliche Interfaces für einen koordinierenen und einen teilnehmenden Web Service. Dabei entsprechen die Klassenhierarchieebenen den Spezifikationen WS-Coordination (`CoordinatorService`, `ParticipantService`), WS-BusinessActivity (`BA...`) sowie der Anwendungslogik.

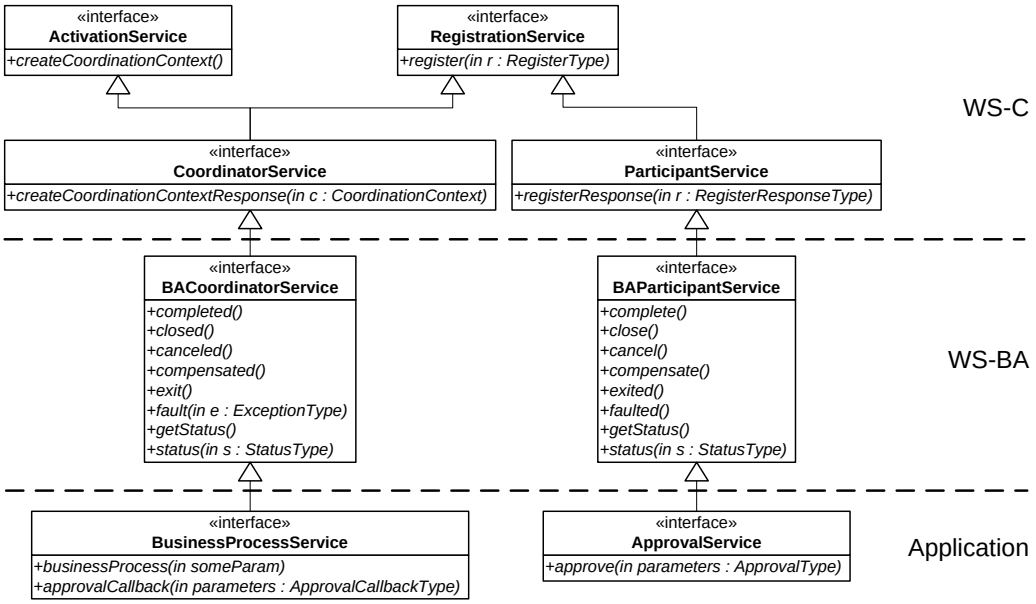

Abbildung 4.9.: Aufbau der Interfaces transaktionaler Web Services

Bezogen auf das Beispiel des Freigabe-Services (s. 1.2) startet die Operation `businessProcess()` des Koordinators eine Transaktion für einen Geschäftsprozess durch Aufrufen des Aktivierungs-Service (`createCoordinationContext()`). Im Rahmen des Prozesses wird die Operationen `approve()` des Freigabe-Services vom Koordinator mit einem gültigen Transaktionskontext aufgerufen und benutzt den Registrierungs-Service (`register()`), um sich für die Transaktion beim Koordinator anzumelden. Die Operation `approvalCallback()` des Koordinators wird von einer teilnehmenden Instanz des Freigabe-Services aufgerufen, um das Ergebnis einer Operation zurückzumelden. Über die Operationen der WS-BusinessActivity Spezifikation wird dann zum Abschluss des Prozess das Terminierungsprotokoll durchgeführt.

Weitere Beispiele für die Interaktion zwischen Koordinator und Teilnehmer sind in Abb. 3.4, Abb. 3.5 und Abb. 3.6 in 3.4.2 dargestellt.

4.4. Code-Generierung

Ein Teil des zur Implementierung von Web Services benötigten Codes dient dazu, die Implementierung in das jeweils benutzte Framework einzubinden. Für jede Web-Service-Operation muss in der Regel eine entsprechende Methode vorhanden sein. In (Schrörs, 2005, 6.1) und (Schrörs, 2006c) wird beschrieben, welche Methoden darüber hinaus vom Basis-Framework für mobile Web Services benötigt werden. Die Funktionalität dieser zusätzlichen Methoden ist dabei für jeden Web Service grundsätzlich gleich, Unterschiede finden sich nur bei den benutzten Operationsnamen und Datentypen.

Web Services, ihre Operationen und die von ihnen benutzten Datentypen werden üblicherweise durch Dokumente in der Web Service Description Language WSDL (WSDL-1.1, 2001) beschrieben. Die darin enthaltenen Informationen können nun genutzt werden, um folgende Artefakte des Implementierungscodes zu generieren.

- Klassen für komplexe Datentypen können aus dem `<types>`-Abschnitt des WSDL-Dokuments generiert werden.
- Für jede Operation des Web Services können Methodenrümpfe generiert werden, die vom Anwendungsentwickler mit Geschäftslogik zu füllen sind.
- Vom jeweiligen Framework verlangte besondere Methoden können unter Einbeziehung der Information über Operationen und Datentypen generiert werden.

Apache Axis (Axis, 2006) enthält z. B. mit `WSDL2Java` einen solchen Code-Generator.

Im Falle transaktionaler Web Services unter Benutzung von WS-BusinessActivity lässt sich aus den funktionalen Anforderungen (s. 4.1) weitere Funktionalität ableiten, die jede Web-Service-Implementierung bereitstellen muss:

- Operationen der Aktivierungs- und Registrierungs-Services (WS-Coordination) auf Koordinatorseite,
- Callback-Operationen des Aktivierungs- und Registrierungsservices auf Koordinator- wie auf Teilnehmerseite,
- die Operationen des WS-BusinessActivity-Protokolls auf Koordinator- und auf Teilnehmerseite, sowie
- evtl. benötigte Hilfsfunktionen.

Dabei muss allen Operationen Zugriff auf das WS-C & WS-BA API (s. 4.3.1) gegeben werden, was zusätzlichen Code für jede Geschäfts-Operation erfordert. Die benötigte Funktionalität und entsprechende Methoden werden vollständig in 5.2.5 beschrieben.

Für eine möglichst umfassende Code-Generierung werden zusätzlich zum WSDL-Dokument (Operationen und Datentypen) noch folgende Parameter für jede Operation benötigt:

- Die Rolle des Web-Services (Koordinator oder Teilnehmer),
- der Koordinationstyp,[8]
- der Protokolltyp,
- die Art der Operation (Transaktionsstart oder bereits mit Transaktionskontext aufzurufende Operation), sowie
- der Web-Service-Scope (s. 4.2.3).

Als Besonderheit speziell für den Einsatz mobiler, asynchroner Web-Services werden weiterhin Informationen über Timeouts für Web-Service-Aufrufe und ggf. über zu benutzende Gateways benötigt (s. 4.2.1). Dies dient der Robustheit einer aus verteilten, mobilen Web-Services bestehenden Anwendung.

Ein Code-Generator sollte aus der Web-Service-Beschreibung (WSDL-Dokument) sowie den genannten Zusatzinformationen folgende Artefakte erstellen:

- Ein vollständiges, um die Operationen von WS-Coordination und WS-BusinessActivity erweitertes WSDL-Dokument zur Benutzung von Web-Service-Clients auf anderen Systemen und mit anderen Frameworks.[9]
- Implementierungs-Code für jede Geschäftsoperation, der für die Einhaltung der Protokolle und Fehlerbehandlung gemäß der Spezifikationen (s. 4.1.2) zuständig ist, sowie
- Methodenrümpfe für die Anwendungslogik, die außer der Signatur keinen weiteren generierten Code enthalten.

Das Ziel der Code-Generierung sollte es sein, dem Entwickler der Geschäftslogik soviel Arbeit wie möglich hinsichtlich der Teilnahme an langlebigen Transaktionen abzunehmen. Transaktions- und Geschäftslogik sollten, soweit im Rahmen von WS-BusinessActivity möglich, vollständig voneinander getrennt werden.

Schließlich kann der Generierung von Code aus der Web-Service-Beschreibung (WSDL) die Generierung des WSDL-Dokuments aus einem Java-Interface vorgelagert sein. Hier ist

[8] Die WS-BusinessActivity-Spezifikation beschreibt, wie für Web-Service-Operationen mit Hilfe von WS-Policy (WS-P, 2006) im WSDL-Dokument angegeben werden kann, dass sie nur im Rahmen einer Transaktion mit bestimmten Koordinationstyp aufgerufen werden dürfen. Der Generator könnte diese Information auswerten.

[9] Hier könnte die in WSDL 2.0 (WSDL-2, 2006) eingeführte Interface-Vererbung nützlich sein: Die für Koordinator und Teilnehmer immer benötigten Operationen könnten in Basis-Interfaces zusammengefasst werden, von denen ein konkreter Web Service dann erbt.

zu beachten, dass aus einem Interface mit Geschäftsmethoden im WSDL-Dokument Operationen für diese Methoden als auch die von WS-Coordination und WS-BusinessActivity benötigen Operationen generiert werden müssen. Die benötigten Parameter für die Transaktionslogik können dabei z. B. im Java-Interface als Annotationen vorhanden sein, aus denen dann die zusätzlich zum WSDL-Dokument benötigten Informationen (s. o.) in für den im Anschluss zu benutzenden Code-Generator geeigneter Art und Weise erzeugt werden.

4.5. Fazit

Das zu entwerfende System ist eine Erweiterung des Frameworks aus (Schrörs, 2005) und (Schrörs, 2006a) um eine Bibliothek von Interfaces und Klassen gemäß 4.3, die Funktionalität und Datentypen zur Unterstützung von langlebigen Transaktionen für Web-Services bereitstellen. Dazu gehören:

- Ein API und die dazugehörige Implementierung zum Senden und Empfangen von Nachrichten gemäß WS-Coordination und WS-BusinessActivity,
- ein Timeout-Mechanismus,
- ein erweitertes Aufruf-API,
- ein einfacher Persistenz-Mechanismus sowie
- Basisklassen für die Implementierung von Web-Service-Interfaces und -Implementierungsklassen, die bereits allgemeine Funktionalität des Frameworks bereitstellen.

Ein weiterer zu entwerfender Teil des Systems ist ein Code-Generator, der für jeden zu implementierenden transaktionalen Web Service aus Angaben über die Geschäftsoperationen (z. B. als Interface oder WSDL-Dokument), die Rolle des Dienstes, die Transaktionalität der Operationen und die Art der Transaktionen Artefakte für die Implementierung der Anwendungslogik erstellt. Aus Zeitgründen wird die Codegenerierung nicht detailliert entworfen und umgesetzt, sondern nur angedeutet.

Bezüglich der nichtfunktionalen Anforderungen werden im Rahmen dieser Arbeit spezielle Komponenten nur zur Förderung der Robustheit (s. 4.2.1) entworfen. Es wird außerdem darauf geachtet, das System so zu entwerfen, dass bezüglich der übrigen nichtfunktionalen Anwendungen eine effiziente, schlanke Implementierung möglich ist, um die Einschränkungen mobiler J2ME-Umgebungen so weit wie möglich zu kompensieren.

Wenn möglich, werden vorhandene Komponenten des ursprünglichen Frameworks (Schrörs, 2006a) erweitert oder wiederverwendet und nur in Ausnahmefällen verändert.

5. Entwurf

Nachdem im vorangehenden Kapitel die Spezifikationen WS-Coordination und WS-BusinessActivity untersucht und darauf aufbauend funktionale und nichtfunktionale Anforderungen formuliert wurden, werden in diesem Kapitel Muster und Techniken ermittelt bzw. erarbeitet, um die Anforderungen umzusetzen und die ebenfalls in der Analyse beschriebenen APIs zu implementieren.

Es wird zunächst ein Überblick über die Architektur des zu entwickelnden Frameworks gegeben (5.1), in dem auch noch einmal die wesentlichen Komponenten des zugrunde liegenden Frameworks (Schrörs, 2006a) kurz beschrieben werden. Dann werden die neuen Komponenten ausführlich beschrieben (5.2). Es werden Muster zur Umsetzung der Spezifikationen, Anforderungen und APIs aus Kapitel 4 ausgewählt und die mögliche Umsetzung durch Klassendiagramme und Interaktionsdiagramme verdeutlicht. Das Exception-Handling wird als allgemeiner, komponentenübergreifender Aspekt in 5.3 diskutiert.

Alle Diagramme in diesem Kapitel sind dabei eigene Darstellungen und sind wie schon in Kapitel 4 in einer an die UML angelehnten Notation gehalten.

Nicht weiter eingegangen wird auf all jene grundlegenden Themen im Zusammenhang mit der Implementierung von Web Services, die in (Schrörs, 2005) und (Schrörs, 2006c) behandelt werden. Dazu gehören Transport (HTTP), XML, SOAP (SOAP, 2003), SAAJ (SAAJ, 2005), Web-Service-Calls, JAX-RPC (JSR-101, 2003), verschiedene Aufruf-Styles und -Encodings (das Transaktions-Framework verlangt aufgrund der Verwendung von WS-Coordination „Document/Literal“), Interaktionsmuster (das Transaktions-Framework verlangt „One Way“-Nachrichten, also asynchrone Interaktion zwischen Web Services) sowie Serialisierung und Deserialisierung von Datentypen.

5.1. Architektur

Die in (Schrörs, 2005, Kap. 5) beschriebene Architektur einer Web Service Middleware für mobile Endgeräte wird um einige allgemeine sowie transaktionsspezifische Komponenten erweitert. Die Zielplattform ist dabei wie in 3.5.1 vorgegeben J2ME in der Ausprägung CLDC 1.1 und MIDP 2.0.

Das zu erweiternde Framework ist im Komponentendiagramm in Abb. 5.1 als „(Schrörs, 2006a)“ gekennzeichnet, da es bereits einige Erweiterungen gegenüber der Version aus 2005 enthält (insbesondere „Document/Literal“-Nachrichtenstil und „One Way“-Nachrichten).

Die allgemeinen Erweiterungen sind unter „Extensions“ zusammengefasst, die transaktionsspezifischen unter „Transactions“.

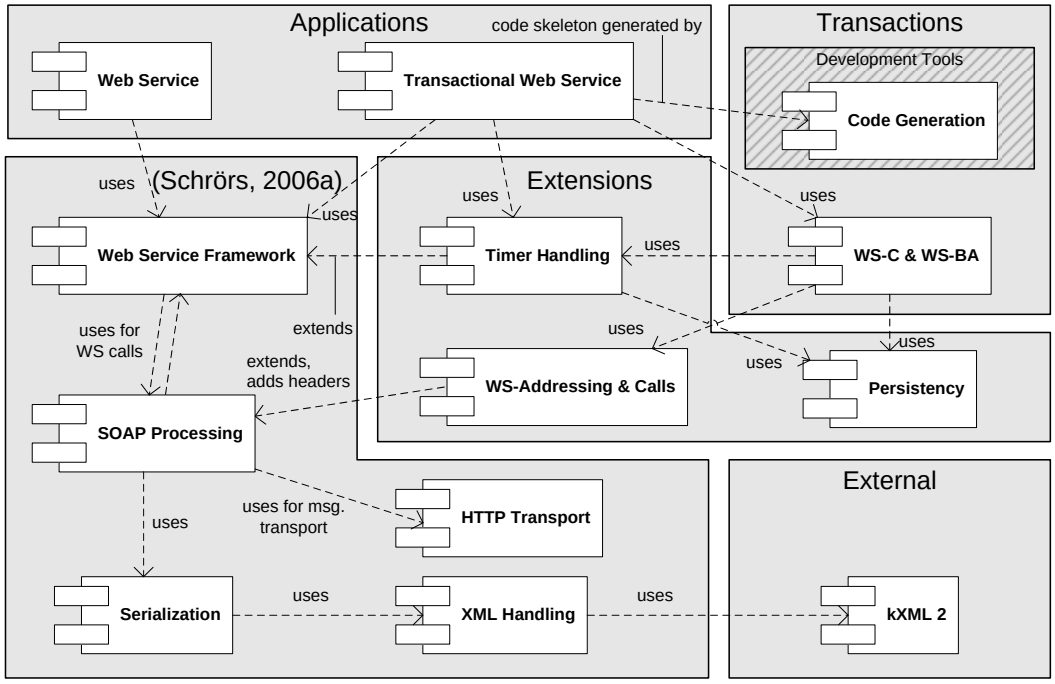

Abbildung 5.1.: Komponentendiagramm

Es folgt eine kurze Beschreibung aller Komponenten. Die erweiterten bzw. neuen Komponenten werden dann im nächsten Abschnitt 5.2 ausführlich beschrieben.

kXML 2: Diese externe Bibliothek (kXML2, 2006) ist die Grundlage für das XML-Handling im ursprünglichen Framework. Sie enthält auch den verwendeten XML-Parser.

XML-Handling: Diese Komponente stellt Mittel zur Behandlung von XML-Dokumenten zur Verfügung.

Serialization: Diese Komponente übernimmt die Serialisierung und Deserialisierung von Nachrichten und Nachrichtenteilen.

SOAP Processing: Diese Komponente, die im Folgenden auch SOAP-Prozessor genannt wird, ist für die Abarbeitung von Service-Aufrufen gemäß SOAP und die Bindung von SOAP an Transportprotokolle (hier nur HTTP) zuständig. Darüber hinaus wird eine an

das „SOAP with Attachments API for Java“ (SAAJ, 2005) angelehnte Bibliothek von APIs und Werkzeugen zum Erzeugen und Analysieren von SOAP-XML-Dokumenten zur Verfügung gestellt.

HTTP Transport: Senden und Empfangen von HTTP-Nachrichten wird von dieser Komponente übernommen.

Web Service Framework: Dieses grundlegende Framework stellt Klassen und Interfaces zur Implementierung von Web Services zur Verfügung.

WS-C & WS-BA: Diese Komponente stellt das in 4.3.1 geforderte API zur Verfügung und stellt Basisklassen zur Implementierung transaktionaler Web Services passend zu den in 4.3.5 beschriebenen Interfaces bereit, welche die vom Web Service Framework (s. o.) bereitgestellten Klassen erweitern. Entwurf: 5.2.1 und 5.2.5.

Timer Handling: Diese Komponente verwaltet die Timer (s. 4.3.2) für Transaktions- und Anwendungslogik. Entwurf: 5.2.2.

WS-Addressing & Calls: Diese Komponente erweitert das bereits vorhandene Call-API um WS-Addressing (s. 4.3.3) und Aufrufstrategien (z. B. Wiederholungen im Fehlerfall, s. 4.2.1.2). Entwurf: 5.2.3.

Persistency: Diese Komponente stellt das in 4.3.4 beschriebene einfache, erweiterbare Persistenz-Framework zur Verfügung, welches von den Basisklassen transaktionaler Web Services genutzt wird. Entwurf: 5.2.4.

Code Generation: Die in 4.4 skizzierte Funktionalität zur Generierung von Code für transaktionale Web Services wird durch diese Komponente zur Verfügung gestellt. Siehe 5.2.6.

Web Service: Nicht-transaktionale mobile Web Services bauen nach wie vor auf dem ursprünglichen Framework auf.

Transactional Web Serivce: Transaktionale mobile Web Services benutzen neben dem ursprünglichen Framework auch die Komponenten des Transaktions-Frameworks. In 5.2.5.4 wird beschrieben, wie das Freigabemanager-Beispiel implementiert werden kann.

5.2. Beschreibung der Komponenten

In diesem Abschnitt werden die neuen Komponenten sowie Erweiterungen der bestehenden Komponenten genau beschrieben. Es werden sowohl die statischen Struktur der Komponenten (Klassen und Relationen) als auch dynamisches Verhalten (Interaktionen) beschrieben.

Wenn möglich und sinnvoll, werden bewährte Entwurfsmuster verwendet, einschließlich der in 3.6.1 beschriebenen (Meta-)Muster für Frameworks.

Neben der Erfüllung der funktionalen Anforderungen steht von den nichtfunktionalen Anforderungen hier die Erweiterbarkeit (s. 4.2.6) im Vordergrund. Die im Zuge von Robustheitsanforderungen (s. 4.2.1) beschriebenen benötigten Funktionen werden ebenfalls an geeigneter Stelle aufgegriffen. Auf Benutzbarkeit (s. 4.2.7) wird im Rahmen der Beschreibung des Code-Generators eingegangen. Alle weiteren nichtfunktionalen Anforderungen können im Rahmen der zur Verfügung stehenden Zeit hier nicht ausführlich behandelt werden.

5.2.1. WS-C & WS-BA (I) – API Implementierung

Die zentrale Komponente des Transaktions-Frameworks besteht aus zwei Teilen. Der erste Teil implementiert das API aus 4.3.1 und wird hier beschrieben. Der zweite Teil bildet die Basis für die Implementierung transaktionaler Web Services. Dieser zweite Teil benötigt neben dem WS-BA API auch die in den folgenden Abschnitten beschriebenen Komponenten und wird daher weiter unten beschrieben (s. 5.2.5).

Das WS-C & WS-BA API wird sowohl vom Framework als auch von der Anwendungslogik benutzt. Da die Operationen von WS-Coordination transparent für die Anwendungslogik ablaufen können, sind für die Anwendungslogik lediglich die WS-BusinessActivity-Teile des APIs interessant. Die Framework-Logik benötigt jedoch darüber hinaus allgemeine WS-Coordination-spezifische, vom Terminierungsprotokoll unabhängige Funktionen, etwa zur Verwaltung der Teilnehmer auf Koordinatorseite.

Die Anforderungen an das API wurde u. a. durch Interfaces (Abb. 4.5) ausgedrückt. Die Realisierung dieses APIs in Java sollte daher diese Interfaces komplett getrennt von der Implementierung enthalten. Durch die Trennung kann die Implementierung später leichter verändert, erweitert oder ausgetauscht werden.

Die Interfaces sind bereits nach WS-Coordination und WS-BusinessActivity getrennt. Für die Vererbungsstruktur der Interfaces sollte es parallel eine identische Struktur von Implementierungsklassen geben. Dabei muss zwischen Koordinatorsicht und Teilnehmersicht unterschieden werden:

- Der Koordinator hat Zugriff auf das Koordinator-API (`Coordinator`).
- Der Koordinator hat über das Koordinator-API auch Zugriff auf das Teilnehmer-API aus Koordinatorsicht für alle registrierten Teilnehmer (`CParticipant`).
- Ein Teilnehmer hat nur Zugriff auf sein eigenes Teilnehmer-API aus Teilnehmersicht (`PParticipant`).

Im Folgenden werden diese drei API-Teile entworfen. Alle Teile haben gemeinsam, dass nur diejenigen Klassen nicht abstrakt sein sollten, die das verwendete Terminierungsprotokoll implementieren (hier also WS-BusinessActivity mit Prefix `BA`). Diese Struktur ist leicht erweiterbar: APIs für weitere auf WS-Coordination aufbauende Protokolle wie z. B. WS-AtomicTransaction können analog zu den `BA...`-Klassen implementiert werden.

Es gibt hier keine Anknüpfungspunkte für eine Erweiterung der Vererbungsstruktur (Template-Hook-Metamuster, s. 3.6.1) für konkrete Anwendungen, die Interfaces (und damit implizit auch die Implementierungsklassen) sind also von der Anwendung genau so zu benutzen wie sie hier entworfen werden.

Die Benutzung des APIs durch Web-Service-Implementierungsklassen wird in 5.2.5 beschrieben.

5.2.1.1. Koordinator

Abb. 5.2 zeigt eine mögliche Implementierung des Koordinator-APIs. Eine Koordinator-Instanz wird vom Framework nach der Aktivierung angelegt. Die konkrete Unterklasse wird vom Koordinationstyp bestimmt. Für die WS-BusinessActivity-Typen gibt es nur `BACoordinatorImpl`.

Die Methode `addParticipant(CParticipant)` des Koordinator-APIs wird vom Framework (Registrierungs-Service) bei der Registrierung von Teilnehmern aufgerufen.

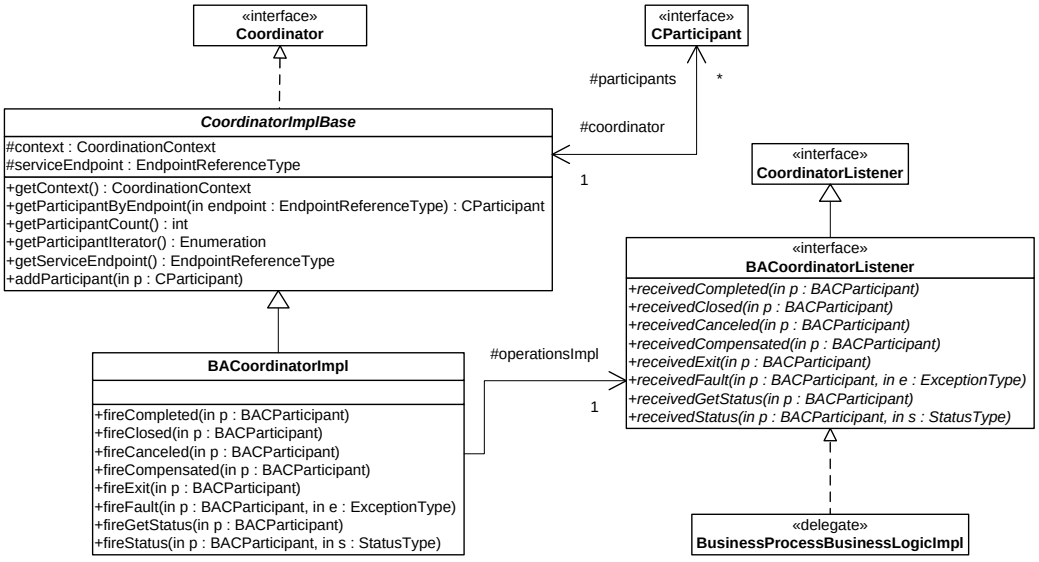

Abbildung 5.2.: Implementierung des Koordinator-APIs

Im Falle von WS-BusinessActivity wird der Koordinator-Instanz bei jedem Aufruf eine Referenz auf eine Implementierungsklasse der Anwendungslogik (s. 5.2.5) übergeben, welche das `BACoordinatorListener`-Interface implementiert. Auf diese Weise wird die Anwendungslogik über Protokollnachrichten einzelner Teilnehmer informiert. Dieser Listener-Mechanismus ist an das Observer-Muster (Gamma u. a., 1995) angelehnt. Da dieser Mechanismus bei anderen Protokollen anders funktionieren könnte, wird die Referenz nicht auf WS-Coordination-Ebene modelliert (z. B. ist WS-AtomicTransaction im IBM WebSphere Application Server 6.0 transparent für die J2EE-Anwendungslogik implementiert (WAS60-AT, 2006), weswegen dort keine Listener-Methoden benötigt werden würden).

Das Koordinatorobjekt sowie die registrierten Teilnehmerobjekte müssen über mehrere Aufrufe persistent gehalten werden (s. 5.2.4).

5.2.1.2. Teilnehmer aus Koordinatorsicht

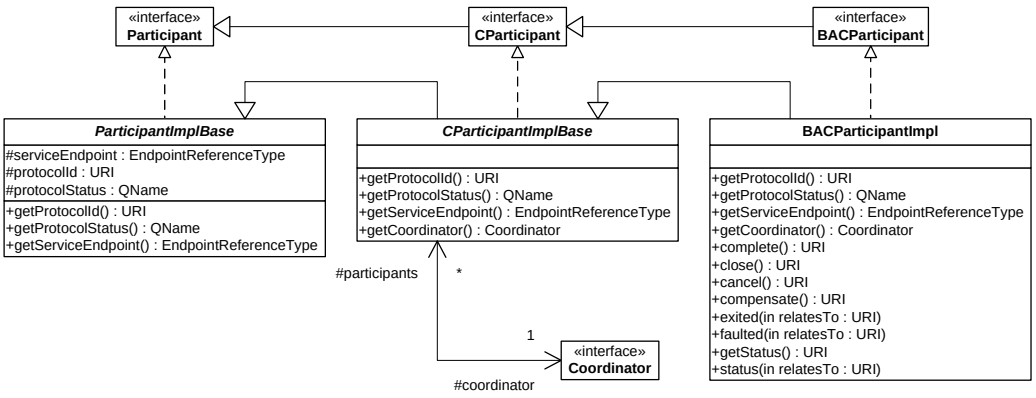

Abbildung 5.3.: Implementierung des Teilnehmer-APIs aus Koordinatorsicht

Abb. 5.3 zeigt eine mögliche Implementierung des Teilnehmer-APIs aus Koordinatorsicht. Der linke Teil (`Participant` und Implementierung) wird auch von der Implementierung aus Teilnehmersicht benutzt und enthält den aktuellen Protokollstatus (s. 4.1.2.4). Der WS-Coordination-Teil (Mitte) erlaubt zusätzlich den Zugriff auf das Koordinatorobjekt.

Der WS-BusinessActivity-Teil erlaubt dem Framework und der Anwendungslogik (über das Interface `BACoordinatorListener` in Abb. 5.2) auf Koordinatorseite, dem Teilnehmer Protokollnachrichten (`complete()` etc.) zu schicken. Das Framework überprüft dabei jeweils, ob eine Nachricht im aktuellen Zustand des verwendeten Protokolls tatsächlich gesendet werden kann.

5.2.1.3. Teilnehmer aus Teilnehmersicht

Abb. 5.4 zeigt schließlich eine mögliche Implementierung des Teilnehmer-APIs auf Teilnehmerseite. Der WS-Coordination-Teil erlaubt dem Framework und der Anwendungslogik hier nach der Registrierung den Zugriff auf den Endpunkt des Koordinators, so dass das Framework Protokollnachrichten an den Koordinator schicken kann und die Anwendungslogik Callback-Methoden der Koordinator-Anwendungslogik aufrufen kann. Neben dem Protokollstatus ist hier auch der Koordinationsstatus (s. 4.1.2.3) zugänglich.

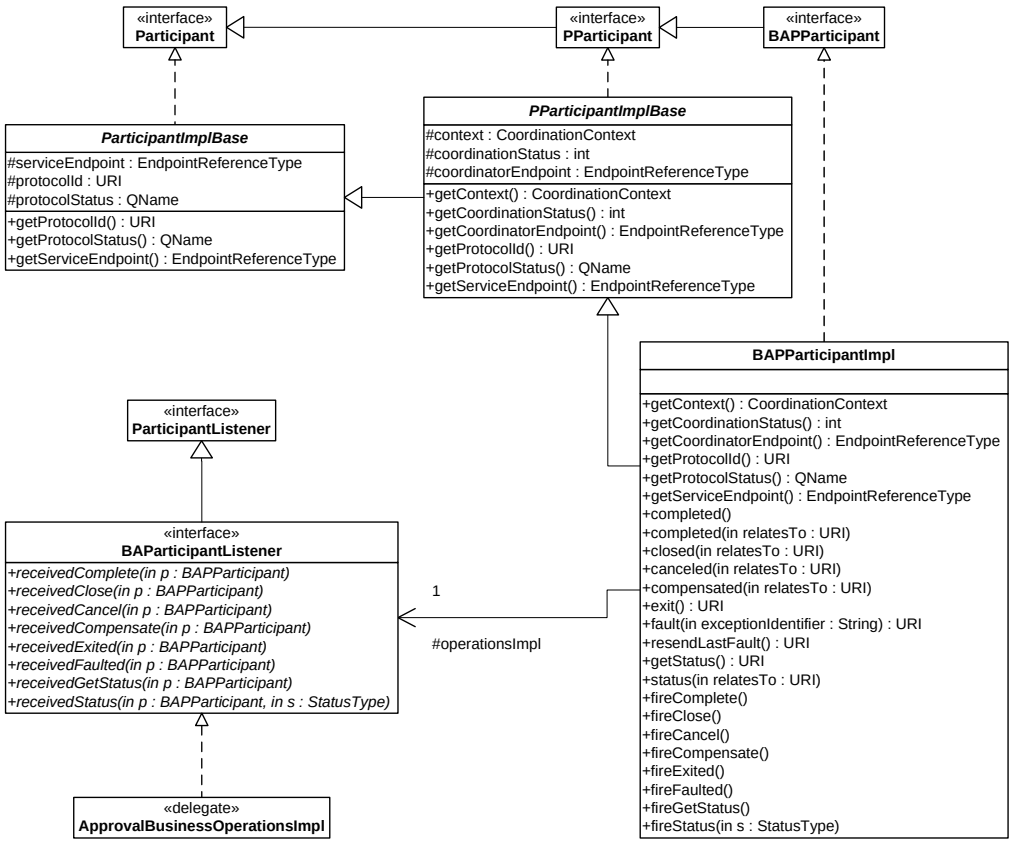

Abbildung 5.4.: Implementierung des Teilnehmer-APIs aus Teilnehmersicht

Analog zur Implementierung des Koordinator-APIs (s. 5.2.1.1) wird hier vom Framework für jeden Aufruf des Teilnehmers dem WS-BusinessActivity-Teil der Implementierung ein Listener-Objekt zur Benachrichtigung der Anwendungslogik über Protokollnachrichten des Koordinators übergeben (`BAParticipantListener`). Dieses sollte die Implementierungsklasse für die Anwendungslogik des Teilnehmers sein (s. 5.2.5).

Analog zur Implementierung des Teilnehmer-APIs aus Koordinatorsicht (s. 5.2.1.2) stehen Framework und Anwendungslogik Methoden zum Schicken von Protokollnachrichten an den

Koordinator (`completed()` etc.) zur Verfügung. Auch hier prüft die Framework-Logik, ob der jeweilige Aufruf im aktuellen Zustand des verwendeten Protokolls gesendet werden kann.

5.2.2. Timer-Handling

Die Komponente Timer-Handling unterstützt den in 4.2.1.3 geforderten Timeout-Mechanismus für asynchrone Web-Service-Aufrufe und implementiert das Timer-API (s. 4.3.2).

Wie die Komponente in das ursprüngliche Framework integriert werden kann, zeigt Abb. 5.5.

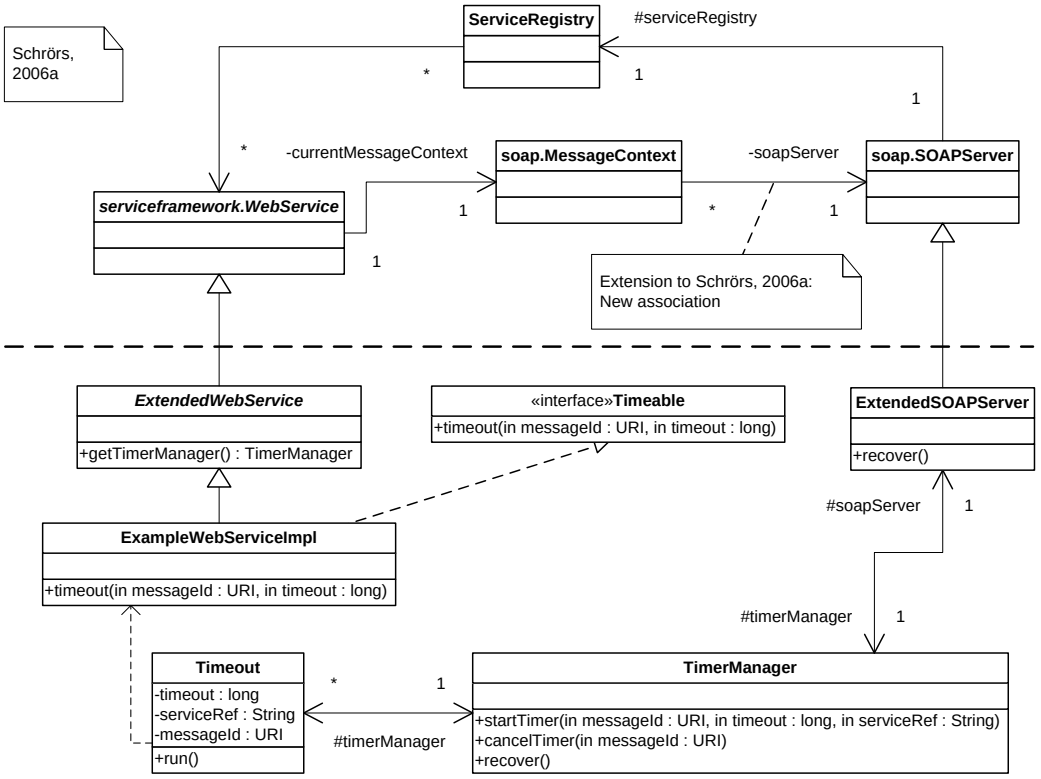

Abbildung 5.5.: Timer-Komponente

Die Klasse `SOAPServer` des ursprünglichen Frameworks wird als `ExtendedSOAPServer` um eine Instanzvariable `timerManager` erweitert, die beim Erzeugen einer Instanz mit einer neuen Instanz der Klasse `TimerManager` initialisiert wird.

Über diese Instanz der Klasse `TimerManager` können Timer gestartet und gestoppt werden. Die Methode `recover()` wird von der gleichnamigen Methode von

`ExtendedSOAPServer` aufgerufen, welche wiederum nach dem (Neu-)Start des SOAP-Servers aufgerufen werden sollte, um eine Wiederherstellung (s. 4.2.1.4) durchzuführen. `TimerManager` muss dazu persistent speichern, welche Timer zu welcher `MessageId` gerade laufen. Dazu kann die in 5.2.4 beschriebene Persistenzschicht verwendet werden.

Um es einer Instanz eines Web Services zu ermöglichen, den `TimerManager` des SOAP-Servers zu benutzen, der den aktuellen Aufruf empfangen hat (eine Web-Service-Implementierung kann bei mehreren SOAP-Servern registriert sein), muss die Klasse `MessageContext` des ursprünglichen Frameworks geändert werden, so dass die `SOAPServer`-Instanz zugreifbar ist, die den Kontext erzeugt hat. Über den aktuellen Aufrufkontext (`currentMessageContext` in `WebService`) kann dann auf den SOAP-Server und über diesen auf den `TimerManager` zugegriffen werden, sofern es sich um einen `ExtendedSOAPServer` handelt. Dafür wird die abstrakte Klasse `WebService` zu einer abstrakten Klasse `ExtendedWebService` erweitert, deren Methode `getTimerManager()` den aktuellen Timer-Manager zurück gibt. Das Sequenzdiagramm in Abb. 5.6 verdeutlicht die Zusammenhänge.

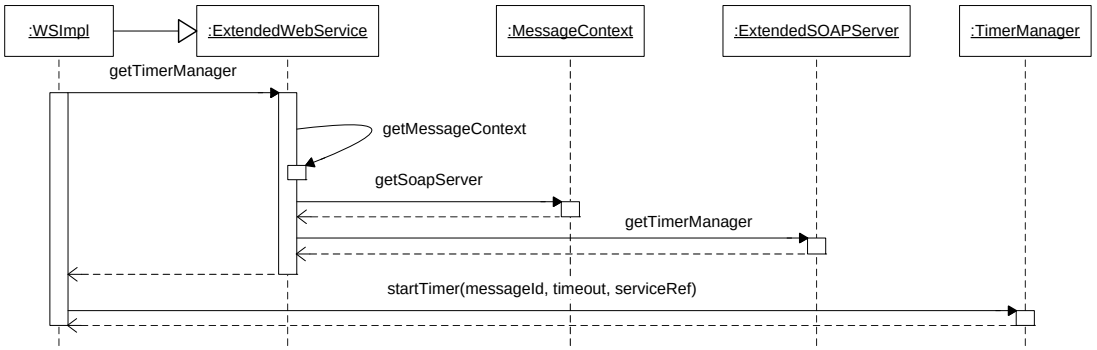

Abbildung 5.6.: Timer starten

Wie für einen asynchronen Web-Service-Aufruf automatisch ein Timer gestartet werden kann, wird im folgenden Abschnitt (5.2.3) beschrieben.

Im Falle eines Timeouts wird wie im Sequenzdiagramm in Abb. 5.7 verfahren. Von der Laufzeitumgebung wird automatisch die Methode `run()` der `Timeout`-Instanz aufgerufen, die zu dem abgelaufenen Timer gehört. Diese kann nun über den zugehörigen `TimerManager` auf die Service-Registry des SOAP-Servers zugreifen, um über die gespeicherte Referenz (`serviceRef`) den Service zu finden, der benachrichtigt werden soll. Implementiert dieser das `Timeable`-Interface, so kann die `timeout(...)`-Methode des Services aufgerufen werden. Dieser etwas umständliche Zugriff wird benutzt, da es einfacher und in der Komplexität planbarer ist, einen String persistent zu speichern, als z. B. Callback-Objekte für die Timeout-Benachrichtung zu serialisieren. Letzteres ist in J2ME mit CLDC 1.1 und MIDP 2.0 zudem nicht möglich.

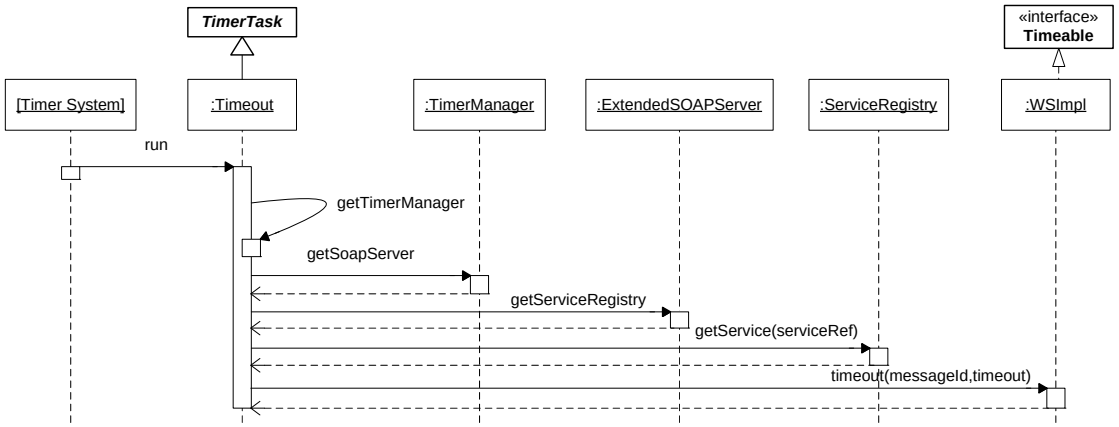

Abbildung 5.7.: Timeout

Das Timer-Handling sollte als abgeschlossene, nicht durch den Anwendungsentwickler zu erweiternde Komponente entworfen werden, die auf die Anforderungen der übrigen Framework-Komponenten abgestimmt ist. Dadurch wird sicher gestellt, dass die benötigte Funktionalität nicht durch den Anwendungsentwickler verändert (z. B. in einer Subklasse von `Timeout` überschrieben) werden kann und dass eine Implementierung schlank und effizient sein kann.

Die Timer-Implementierung der Laufzeitumgebung (MIDP 2.0) basiert auf einem abgewandelten 1:1-Connection-Metamuster (s. 3.6.1): Zur Laufzeit wird ein Callback (in Form der `Timeout`-Instanz) übergeben und die Laufzeitumgebung ruft dieses Callback nach Ablauf des Timers auf.

5.2.3. WS-Addressing und Calls

Die Komponente WS-Addressing & Calls sollte neben der in 4.3.3 beschriebenen Erweiterung der Klasse `soap.Call` aus (Schrörs, 2006a) auch einen Mechanismus zur Verfügung stellen, der es ermöglicht, einen Aufruf auf unterschiedliche Arten auszuführen, wobei es möglich sein sollte, dass der Aufruf und die Art des Aufrufs (die Aufrufstrategie) voneinander getrennt definiert werden. Dafür eignet sich das Strategy-Muster aus (Gamma u. a., 1995).

Eine Anwendung von Aufrufstrategien besteht in Aufrufen mit Wiederholung des Sendevorgangs im Fehlerfall, wie in 4.2.1.2 beschrieben. Die Implementierung der Strategie sorgt für die Wiederholungen, während das `Call`-Objekt selbst nur den eigentlichen Web-Service-Aufruf durchführt.

Das WS-BA API (s. 5.2.1) muss `Call`-Objekte benutzen, um die Protokollnachrichten zu verschicken. Implementiert man die entsprechenden Methoden (z. B. `complete()`,

`completed()`, `close()`) nun mit einem (zusätzlichen) Parameter vom Typ `CallStrategy`, in den dann von der Implementierung der Call „eingesetzt“ wird, so kann man von außen steuern, wie die Protokollnachrichten verschickt werden. Dies gilt genauso für die Basisklassen der Web-Service-Implementierung, die z. B. die Aktivierung und Registrierung ausführen (s. 5.2.5). Die zu verwendende Strategy-Implementierung kann schließlich als Parameter für einen Code-Generator (s. 4.4) zur Erzeugung der konkreten Web-Service-Implementierungsklassen dienen, aber auch eine dynamische Ermittlung der zu verwendenden Strategie zur Laufzeit ist vorstellbar.

Abb. 5.8 zeigt mögliche Implementierungen des Strategy-Musters.

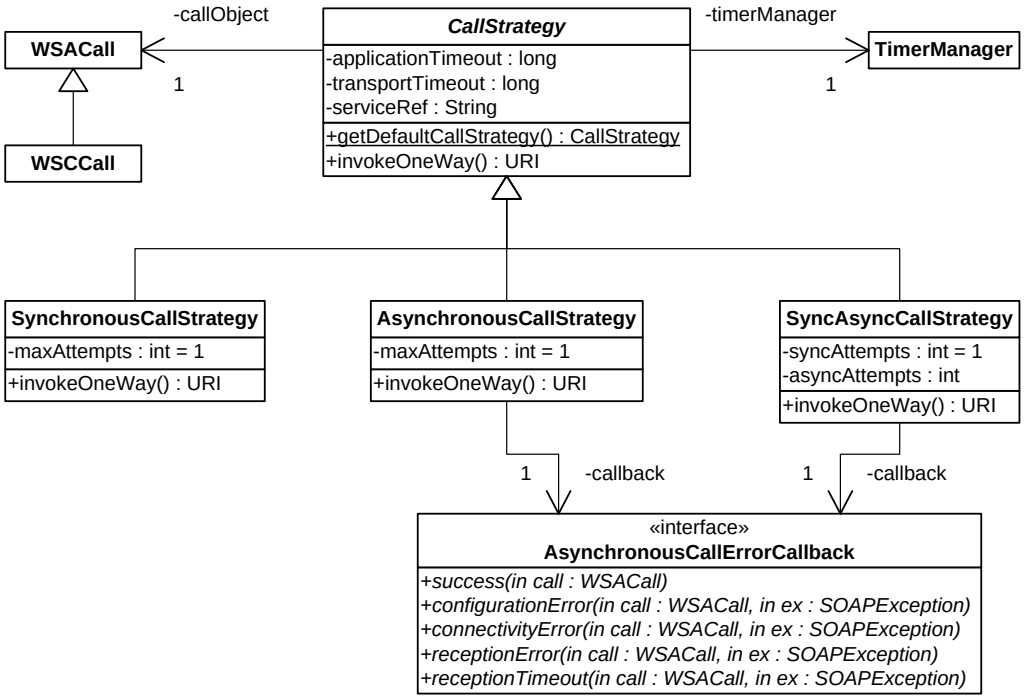

Abbildung 5.8.: Implementierung von Aufrufstrategien

Darüber hinaus sind auch Stretegy-Implementierungen für Aufrufe mobiler Empfängern mit langen Offline-Zeiten denkbar, die die Nachricht an ein Gateway (s. 4.2.1) übergeben, oder solche Implementierungen, die zunächst versuchen, die Nachricht direkt zu verschicken, bei einer bestimmten Anzahl von Fehlversuchen jedoch die Nachricht an ein Gateway übergeben. Auf diese Weise kann man die Benutzung von Gateways transparent für die Transaktions- und Anwendungslogik implementieren. Dem Absender des Aufrufs wird ggf. vom Gateway bestätigt, dass die Nachricht angenommen wurde. Das Gateway versucht dann die Nachricht auszuliefern. Erfordert der Aufruf eine asynchrone Antwort, sollte der Absender nun einen Timer (s. 5.2.2) starten, um festzulegen, wie lange auf die Antwort maximal gewartet werden soll.

Bei der Anwendung von Aufrufstrategien sorgt eine Instanz der `Call`-Erweiterung (`WSACall` oder die Subklasse `WSCCall`, s. 4.3.3) zunächst für die automatische Serialisierung von SOAP-Headern vor Ausführung des Aufrufs. Dazu muss die Erweiterung automatisch Custom Datatypes (s. (Schrörs, 2005, 5.2.2, 6.1) bzw. (Schrörs, 2006b, 3.6)) für `EndpointReferenceType` und `CoordinationContext` registrieren. Dies kann z. B. im Konstruktor geschehen. Die Datentypen werden benötigt, um die in WS-Addressing und WS-Coordination spezifizierten WS-Addressing-Header zu serialisieren.

Eine Strategie benutzt schließlich die `Call`-Instanz für den eigentlichen Aufruf. Das Sequenzdiagramm in Abb. 5.9 verdeutlicht den Vorgang.

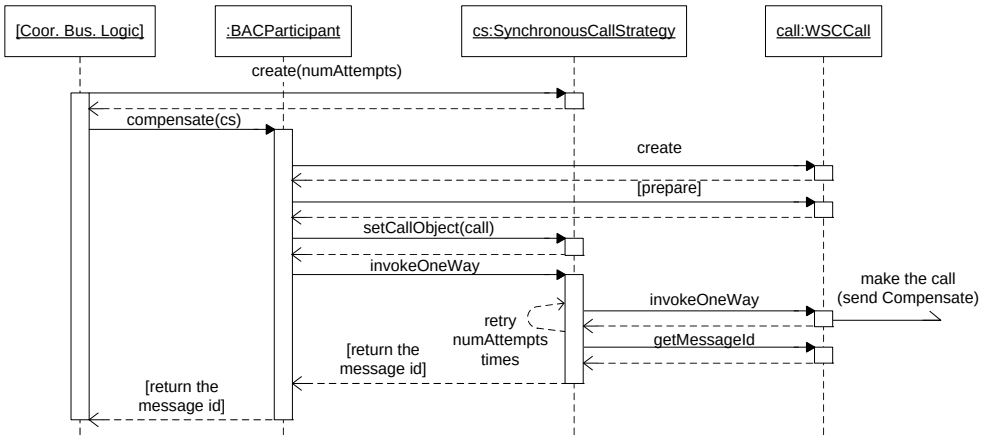

Abbildung 5.9.: Web-Service-Call mit Strategy-Muster

Listing 5.1 verdeutlicht die Funktionsweise von Aufrufstrategien anhand des Beispiels einer synchronen Strategie mit einem oder mehreren Sendeversuchen. Der eigentliche Aufruf erfolgt – möglicherweise mehrfach – innerhalb der Methode `invokeOneWay()` (Zeile 8).

Listing 5.1: SynchronousCallStrategy.invokeOneWay()

```
1  public URI invokeOneWay() throws SOAPException {
2          if (callObject == null) throw new SOAPException("en","No Call object");
3          URI messageId = null; SOAPException ex = null; int attempts = 0; boolean retry = true;
4          messageId = callObject.generateMessageId();
5          while (retry && attempts < maxAttemtps) {
6                  try {
7                          ex = null;
8                          callObject.invokeOneWay();
9                          retry = false;
10                 } catch (SOAPException e) {
11                         System.out.println("Call attempt " + (++attempts));
12                         e.printStackTrace(); ex = e;
13                 }
14         }
15         if (ex != null) throw ex;
16         if (applicationTimeout>0) timerManager.startTimer(messageId, applicationTimeout, serviceRef);
17         return messageId;
18 }
```

Der Code verdeutlicht weiterhin, wie nach einem erfolgreichen Aufruf ein Anwendungs-Timer gestartet wird (Zeile 16).

Wie in Abb. 5.8 angedeutet, sind auch Strategien denkbar, die die Sendeversuche ganz oder teilweise in einem separaten Thread durchführen und ein Callback-Objekt über das Ergebnis informieren, wobei die Fehlermethoden des Callbacks mit den Fehlerarten aus 4.2.1.1 korrespondieren. Ein solches asynchrones Senden erfordert allerdings eine aufwendige Anwendungslogik für den Fall, dass weitere Schritte vom Erfolg des Sendens abhängen. Es sollte daher gut überlegt werden, ob die maximale Dauer aller Sendeversuche (im schlimmsten Falle jedes Mal ein Timeout auf Transportebene, z. B. HTTP-Timeout) in synchroner Verarbeitung toleriert werden kann.

Aufrufstrategien sind sowohl für die Anwendungslogik als auch für die Frameworklogik benutzbar und insbesondere erweiterbar. Es können neue, beliebige Aufrufstrategien implementiert und benutzt werden. Erweiterungen von `WSCCall` für den eigentlichen Aufruf können auch benutzt werden, z. B. um weitere wichtige Datentypen automatisch zu registrieren.

Wie beim Timer-Mechanismus der Laufzeitumgebung kommt für die Aufrufstrategien das 1:1-Connection-Metamuster zur Anwendung. Das Strategy-Muster ist selbst ein Spezialfall davon (s. 3.6.1) und auch das angegebene Beispiel einer asynchronen Aufrufstrategie mit Callback-Objekt fällt unter diese Kategorie.

5.2.4. Persistenz

Für die Realisierung der in 4.3.4 beschriebenen Storage- und Konverter-Klassen eignet sich das Muster Abstract Factory aus (Gamma u. a., 1995). Eine Umsetzung ist in Abb. 5.10 angedeutet. `SimpleStorage` verwendet darüber hinaus das Singleton-Muster (Gamma u. a., 1995), um nur eine `Hashtable`-Instanz je Java Virtual Machine zu verwenden.

Konverter müssen für jeden zu speichernden Typ registriert werden. Beim Schreiben und Lesen von Objekten werden die Konverter dann benutzt, um ähnlich dem Builder-Muster aus (Gamma u. a., 1995) Objektstrukturen in das Speicherformat und zurück zu konvertieren. Abb. 5.11 verdeutlicht den Vorgang.

Für `SimpleStorageFactory` braucht aufgrund der Verwendung einer `Hashtable` nicht konvertiert werden. Natürlich stellt `SimpleStorageFactory` auch keinen über die Lebensdauer der Java Virtual Machine hinaus persistenten Speicher zur Verfügung und kann daher nur für Testzwecke eingesetzt werden. `J2MERecordStoreFactory` würde dagegen Konverter benötigen, die beim Schreiben Objekte in Byte-Arrays (`byte[]`) konvertieren und beim Lesen aus den Byte-Arrays wieder Objekte erzeugen.

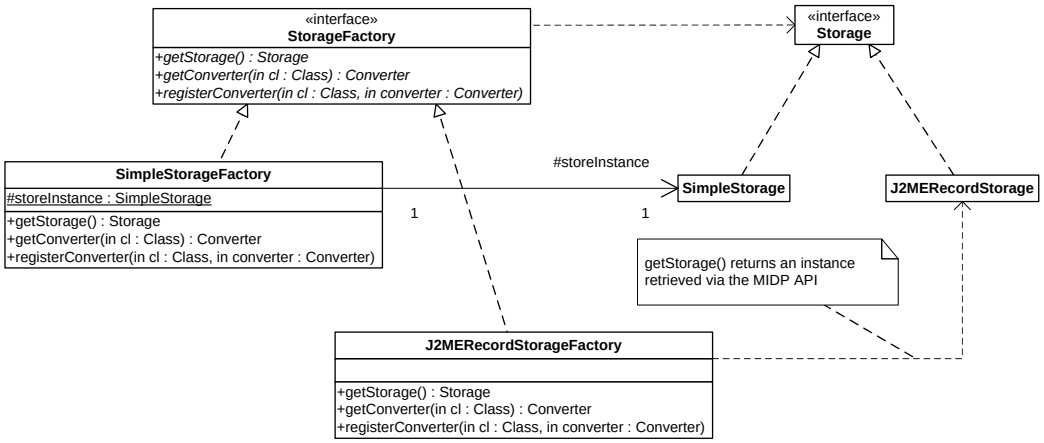

Abbildung 5.10.: Implementierung des Persistenz-APIs

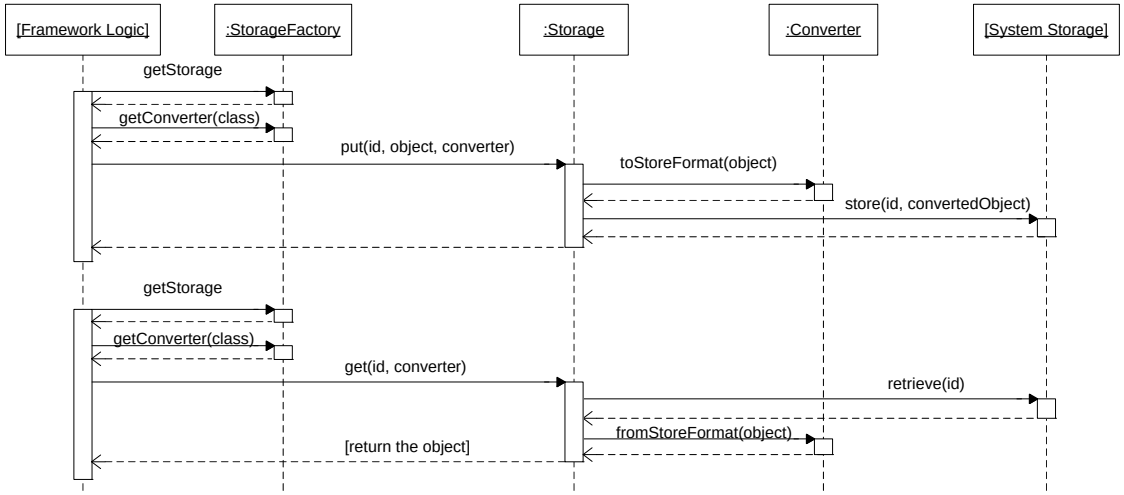

Abbildung 5.11.: Schreiben und Lesen persistenter Objekte

Konverter werden vom Transaktions-Framework für folgende Hauptdatentypen und für von diesen verwendete Hilfsdatentypen gebraucht:

Operation: Name, Parameter und SOAP-Header des Aufrufs für Operationen, die für die Dauer von Aktivierung und Registrierung zwischengespeichert werden müssen.

Coordinator: Alle relevanten Informationen auf Koordinatorseite, einschließlich Daten über die registrierten Teilnehmer (s. 4.1.6).

Participant: Alle relevanten Informationen auf Teilnehmerseite (s. 4.1.6).

Timer: Absoluter Timeout-Zeitpunkt, Service-Referenz und `MessageId` für jeden Anwendungs-Timer.

Abb. 5.12 zeigt, dass `StorageFactory` analog zu `TimerManager` (s. 5.2.2) an das ursprüngliche Framework anzubinden ist. Die `StorageFactory`- und `Storage`-Implementierungen kapseln die Komplexität der Persistenzschicht ähnlich wie im Entwurfsmuster DomainStore (Alur u. a., 2003), welches für J2EE-Persistenzimplementierungen verwendet werden kann.

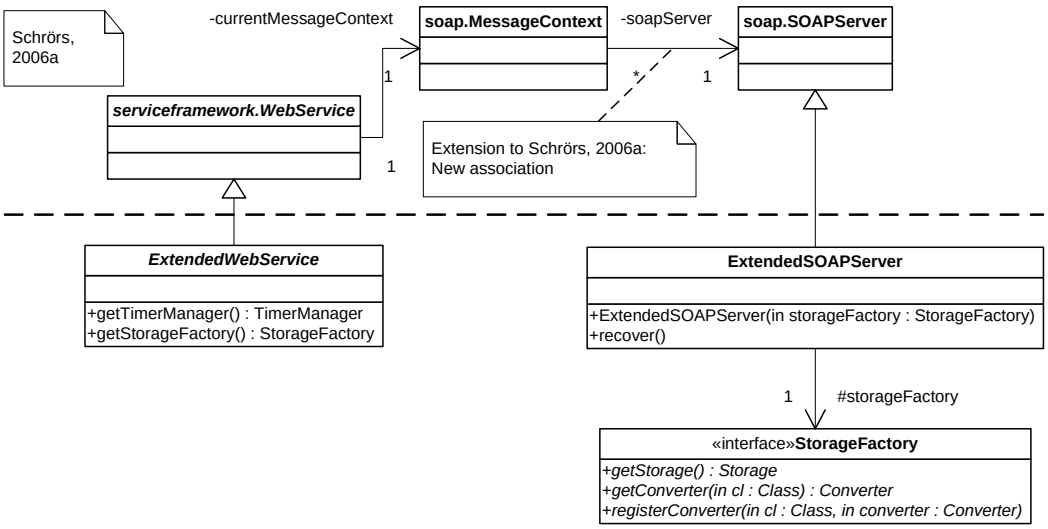

Abbildung 5.12.: Persistenz-Komponente

Um es einer Instanz eines Web Services (und damit auch der Transaktions-Logik) zu ermöglichen, die `StorageFactory` des SOAP-Servers zu benutzen, der den aktuellen Aufruf empfangen hat, wird die in 5.2.2 beschriebene Erweiterung der Klasse `MessageContext` benutzt, die die `SOAPServer`-Instanz zugreifbar macht, die den Kontext erzeugt hat. Über den aktuellen Aufrufkontext (`currentMessageContext`) kann dann auf den SOAP-Server und über diesen auf die `StorageFactory` zugegriffen werden, sofern es sich um einen `ExtendedSOAPServer` handelt (Abb. 5.13).

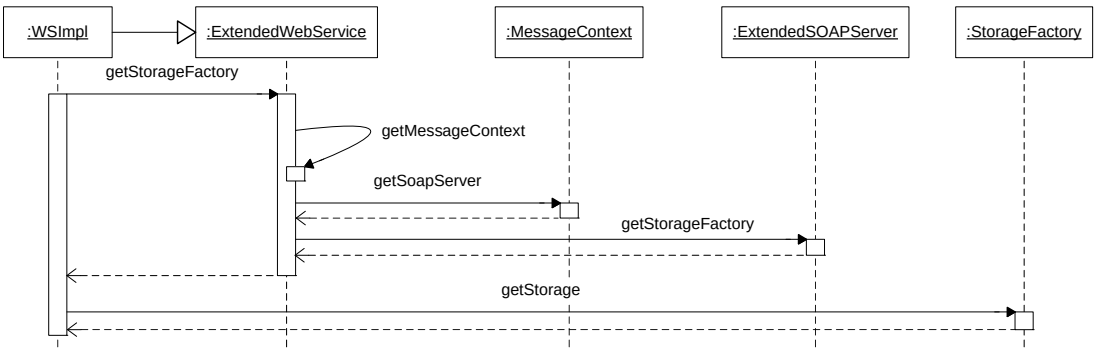

Abbildung 5.13.: StorageFactory holen

Die in 5.2.2 bereits eingeführte Klasse `ExtendedWebService` müsste dazu eine weitere Methode `getStorageFactory()` erhalten, welche die aktuelle `StorageFactory`-Instanz wie oben beschrieben ermittelt und zurück gibt.

Die vom SOAP-Server zu benutzende konkrete Instanz von `StorageFactory` muss dem Konstruktor von `ExtendedSOAPServer` übergeben werden, da verschiedene Implementierungen möglich sind. Die `StorageFactory`-Instanz ist somit Teil der Konfiguration des SOAP-Servers.

Die beschriebene Persistenzschicht wurde mit dem Ziel entworfen, die vom Transaktions-Framework benötigte Funktionalität, nämlich die Speicherung o. g. Objekttypen zu der aktuellen Transaktions-ID, möglichst schlank und effizient zur Verfügung zu stellen. Sie dient ausschließlich der Wiederherstellbarkeit wie in 4.2.1.4 beschrieben und ist nicht dafür geeignet, von der Anwendungslogik für die persistente Speicherung beliebiger Daten verwendet zu werden.

Hinsichtlich der konkreten Art der Speicherung ist die Komponente jedoch aufgrund der Verwendung der o. g. Entwurfsmuster einfach erweiterbar. So könnten z. B. JDO (Java Data Objects nach (JSR-12, 2003)) verwendet werden. Siehe hierzu (Parsons, 2005, Abschnitt 4) und (Braig und Gemkow, 2002). Für leistungsfähige Endgeräte mit CDC (Connected Device Configuration nach (JSR-218, 2005)) ist auch der Einsatz einer mobilen Datenbank denkbar. Für CLDC 1.1 und MIDP 2.0 ist zu beachten, dass keine Objektserialisierung möglich ist.

Für eine Erweiterung des Frameworks sind eine neue `StorageFactory`-Subklasse sowie dazu passende neue Konverter-Klassen für alle benötigten Entitäten zu entwickeln.

Das Prinzip der `StorageFactory`-Implementierungen entspricht dem Unification-Metamuster, das Prinzip der Konverter (Builder-Muster) entspricht dem 1:1-Connection- bzw. dem 1:N-Connection-Metamuster (s. 3.6.1).

5.2.5. WS-C & WS-BA (II) – Web Service Implementierung

Das Transaktions-Framework sollte möglichst viel Standardfunktionalität für transaktionale Web Services bereitstellen und möglichst viel Transaktionslogik abwickeln, gleichzeitig aber eine Interaktion mit der Anwendungslogik ermöglichen, wo es erforderlich oder wünschenswert ist. Folgende Ziele werden beim Entwurf verfolgt:

- Verlagerung von soviel Verantwortung für die Einhaltung der Protokolle wie möglich in das Framework.
- Erweiterbarkeit des Frameworks für andere auf WS-Coordination aufbauende Terminierungsprotokolle.

Die benötigte Funktionalität lässt sich auf zwei Weisen unterteilen:

1. Unterteilung nach Protokoll: Transaktionale Web Services müssen wie gewöhnliche Web Services gemäß den Vorgaben von (Schrörs, 2005) bzw. (Schrörs, 2006c) implementiert werden, um im SOAP-Prozessor des Frameworks zu funktionieren. Weiterhin ist zwischen Funktionalität für WS-Coordination und Funktionalität für WS-BusinessActivity zu unterscheiden und schließlich wird Funktionalität für die konkrete Anwendung benötigt. Umgekehrt betrachtet basiert die Anwendung auf WS-BusinessActivity, welches eine Spezialisierung (im Sinne einer speziellen Anwendung) von WS-Coordination ist. WS-Coordination ist wiederum eine Spezialisierung von SOAP bzw. Web Services mit asynchronen Operationen im Stil Document/Literal (s. 3.5.1).

2. Unterteilung nach allgemeiner und anwendungsspezifischer Funktionalität: Allgemeine Funktionalität kann komplett vom Framework implementiert werden. Anwendungsspezifische Funktionalität bezeichnet die Transaktionslogik, die in den Methoden konkreter Web Services abläuft, bevor die Anwendungslogik ausgeführt wird. Das besondere Problem der anwendungsspezifischen Funktionalität ist, dass die Web-Service-Methoden zum Entwicklungszeitpunkt des Frameworks nicht bekannt sind. Es gilt in diesem Fall, möglichst viel Funktionalität in Form von Framework-Methoden zu Verfügung zu stellen, welche dann vom Implementierungscode der Web-Service-Methoden aufgerufen werden.

In jedem dieser beiden Fälle eignet sich das Prinzip der Vererbung zum Erreichen der Ziele. Dies ist in (Schrörs, 2005) schon so vorgesehen. In den oberen Klassen (Superklassen) der Vererbungshierarchie finden sich demnach allgemein benötigte Operationen, weiter unten (in den Subklassen) speziellere.

Dabei kann von dem Template-Hook-Metamuster (s. 3.6.1) Gebrauch gemacht werden: Wo immer in einer Superklasse Funktionalität benötigt wird, die erst in einer Subklasse implementiert werden kann, wird dafür in der Superklasse eine abstrakte Hook-Methode deklariert, die in einer Subklasse implementiert werden muss.

Da für vom ursprünglichen Framework pro Web Service nur ein Objekt verwaltet wird, liegen Template- und Hook-Methoden immer in derselben Klassenhierarchie (Metamuster Unification).

Wo von Subklassen die Funktionalität der Superklassen erweitert wird, kommt als Ausprägung des Template-Hook-Metamusters entsprechend das Metamuster 1:1 Recursive Connection zum Einsatz, welches nichts anderes beschreibt, als dass eine Methode der Subklasse die gleichnamigen Methode der Superklasse aufruft und so eine Aufrufkette bildet bzw. fortsetzt. Geeignet ist z. B. die konkrete Ausprägung Chain of Responsibility (Gamma u. a., 1995).

Durch die konsequente Anwendung dieser Muster wird der Aufwand für eine Erweiterung des Frameworks zur Benutzung eines anderen Terminierungsprotokolls als WS-BusinessActivity minimiert.

Im Folgenden werden für jede Schicht der Unterteilung 1 oben allgemein und anwendungsspezifisch benötigte Funktionen gemäß Unterteilung 2 und eine mögliche Implementierungsstrategie detailliert beschrieben. An den für eine Erweiterung auf andere Terminierungsprotokolle relevanten Stellen wird darauf hingewiesen, was konkret dafür getan werden müsste.

5.2.5.1. SOAP

Die Implementierung eines Web Services wird immer von `serviceframework.WebService` abgeleitet. Diese Klasse muss zur Nutzung der Timer- und Persistenzkomponenten wie oben beschrieben erweitert werden (`ExtendedWebService`). Auf der gleichen Ebene kann auch eine Funktion zur Deserialisierung der SOAP-Header implementiert werden, ein Feature, das in (Schrörs, 2006a) noch fehlt.

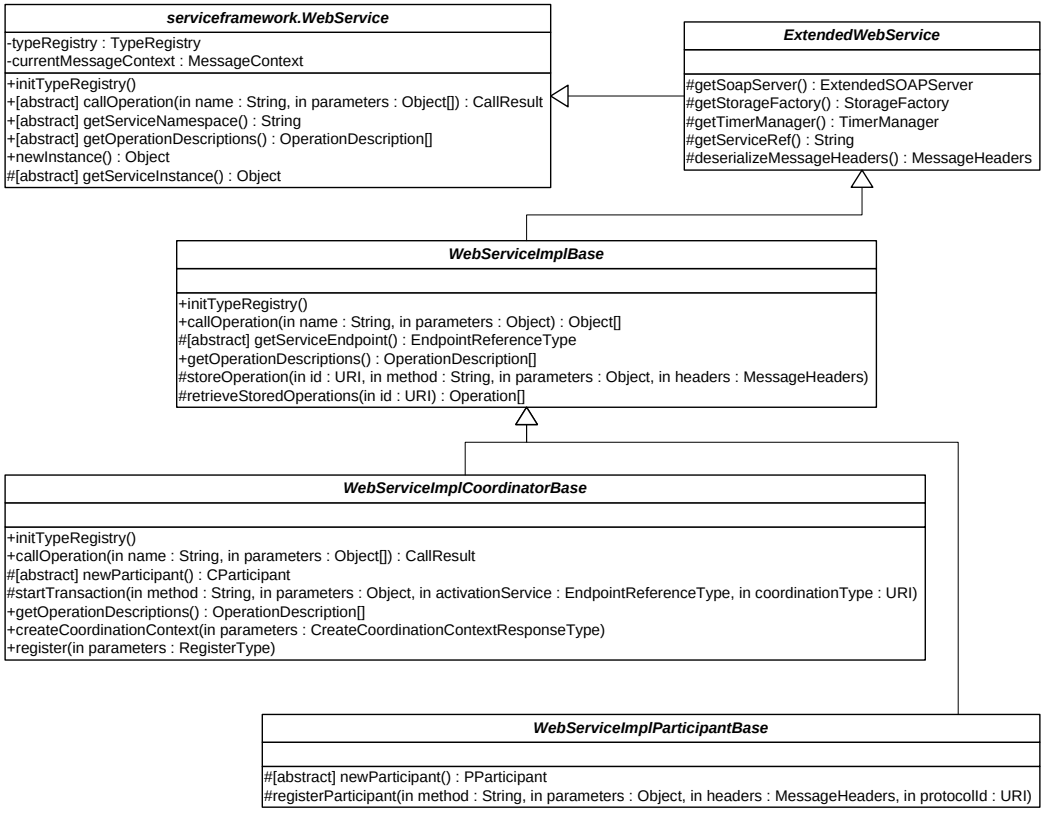

Abbildung 5.14.: Web Service Implementierung – Basisklassen und WS-C

Die Klassen sind in Abb. 5.14 im oberen Bereich dargestellt. Die mit „[abstract]“ gekennzeichneten abstrakten Hook-Methoden der Root-Klasse `serviceframework.WebService` werden noch nicht implementiert und auch keine Erweiterungen der anderen Methoden. Dies geschieht in den Subklassen wie folgt (eine genaue Beschreibung findet sich in (Schrörs, 2006c, Kap. 3)):

initTypeRegistry: Jede Subklasse registriert hier benötigte und auf dieser Ebene bereits bekannte Datentypen zur Deserialisierung von SOAP-Headern (z. B. zur Nutzung von WS-Addressing, s. 4.1.5) und Parametern von Web-Service-Operationen und ruft gemäß Chain-of-Responsibility-Muster die Implementierung der Superklasse auf (falls vorhanden), damit alle benötigten Datentypen registriert werden.[10]

callOperation: Jede Subklasse prüft, ob sie die aufzurufende Operation implementiert. Falls nicht, wird die gleiche Methode der Superklasse aufgerufen. Das Chain-of-Responsibility-Muster kommt hier wie in Abb. 5.15 gezeigt zum Einsatz, um den in J2ME CLDC/MIDP fehlenden Reflection-Mechanismus zu emulieren und einen auf mehrere Stufen der Vererbungshierarchie verteilten Broker für Operationsaufrufe zu implementieren. Zum Broker-Muster s. a. (Buschmann u. a., 2004).

getServiceNamespace: Diese Methode kann nur von konkreten Web-Service-Implementierungen implementiert werden, denn nur für diese ist der XML-Namespace des Web Services bekannt.

getOperationDescriptions: Es wird hier eine Aufrufkette durch die Hierarchie benötigt, um alle implementieren Operationen zusammenzustellen. An den Rückgabewert der Implementierung in der Superklasse werden Informationen über von der Subklasse implementierte Operationen angehängt. Diese gesamte Kollektion wird dann zurückgegeben. Auch hier wird das Muster Chain of Responsibility verwendet.

getServiceInstance: Diese von `newInstance()` aufgerufene Methode dient der Erzeugung einer neuen Instanz der konkreten Implementierungsklasse. Die Methode kann auch nur von eben dieser Klasse implementiert werden, da nur auf dieser Ebene bekannt ist, wie die Erzeugung einer neuen Instanz genau funktioniert (z. B. welcher Konstruktor verwendet wird oder ob ein Singleton-Objekt zurück geliefert wird).

Im Folgenden werden diese Methoden nur dann beschrieben, wenn sie besondere Funktionalität bereitstellen müssen. In den referenzierten Klassendiagrammen tauchen die Methoden immer dort auf, wo sie überschrieben werden müssen.

[10] Zur Nutzung des Registry-Musters (Fowler, 2003) für die Datentypenregistrierung siehe (Schrörs, 2005, 5.2.2) bzw. (Schrörs, 2006b, 3.5, 3.6).

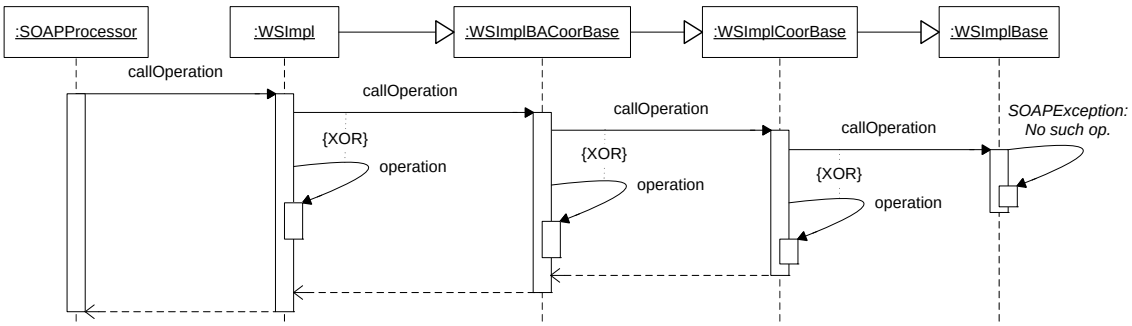

Abbildung 5.15.: Operationsaufruf (Chain of Responsibilty)

5.2.5.2. WS-Coordination

Funktionen für WS-Coordination lassen sich in allgemeine Funktionen sowie Funktionen für koordinierende und teilnehmende Web Services aufteilen. Dies kann durch die drei in Abb. 5.14 im unteren Bereich dargestellten Subklassen von `ExtendedWebService` erreicht werden.

`WebServiceImplBase` implementiert allgemeine Funktionalität:

getServiceEndpoint: Ähnlich wie `getServiceNamespace()` kann diese Hook-Methode nur von der konkreten Implementierung des Web Services implementiert werden, da erst zum Entwicklungszeitpunkt bekannt sein kann, wie der Web Service erreichbar ist. Der Rückgabewert wird ebenfalls für WS-Addressing-Header benötigt (z. B. `ReplyTo`).

getOperationDescriptions: Als Wurzel für die Zusammenstellung von Operationsbeschreibungen sollte hier eine leere Liste zurückgegeben werden.

storeOperation: Koordinator und Teilnehmer müssen während der Aktivierung bzw. der Registrierung Informationen über danach aufzurufende Operationen persistent zwischenspeichern, was bereits auf dieser Ebene erledigt werden kann.

`WebServiceImplCoordinatorBase` implementiert Funktionalität des Koordinators. Insbesondere können Aktivierungs- und Registrierungs-Service auf dieser Ebene mithilfe einer abstrakten Hook-Methode implementiert werden. Die Entscheidung, diese beiden Dienste immer als Teil des Koordinators zu implementieren (s. 4.3.5) wurde vor dem Hintergrund beschränkter Systemressourcen getroffen, um Abhängigkeiten und Kommunikationsaufwand zu reduzieren.

Die auf dieser Ebene benötigten bzw. möglichen Methoden sind:

startTransaction: Diese allgemeine Hilfsmethode kann von den anwendungsspezifischen Implementierungen der Web-Service-Operationen (s. u.) aufgerufen werden, die eine neue Transaktion starten. Der Operationsaufruf wird zwischengespeichert (s. o.) und es wird der Aktivierungs-Service aufgerufen (Abb. 5.16).[11]

createCoordinationContext: Diese Methode implementiert die Web-Service-Operation des Aktivierungs-Services (Abb. 5.17).

register: Diese Methode implementiert den Registrierungs-Service. Dabei wird ein `CParticipant`-API-Objekt erzeugt, welches dem `Coordinator`-Objekt zugeordnet und mit diesem persistent gespeichert wird (Abb. 5.18). Da auf dieser Ebene nicht bekannt ist, welches Terminierungsprotokoll verwendet wird, muss die Erzeugung in einer Subklasse über die folgende Hook-Methode erfolgen.

newParticipant: Diese Hook-Methode dient zur Erzeugung eines neuen API-Objekts für einen Teilnehmer aus Koordinatorsicht.

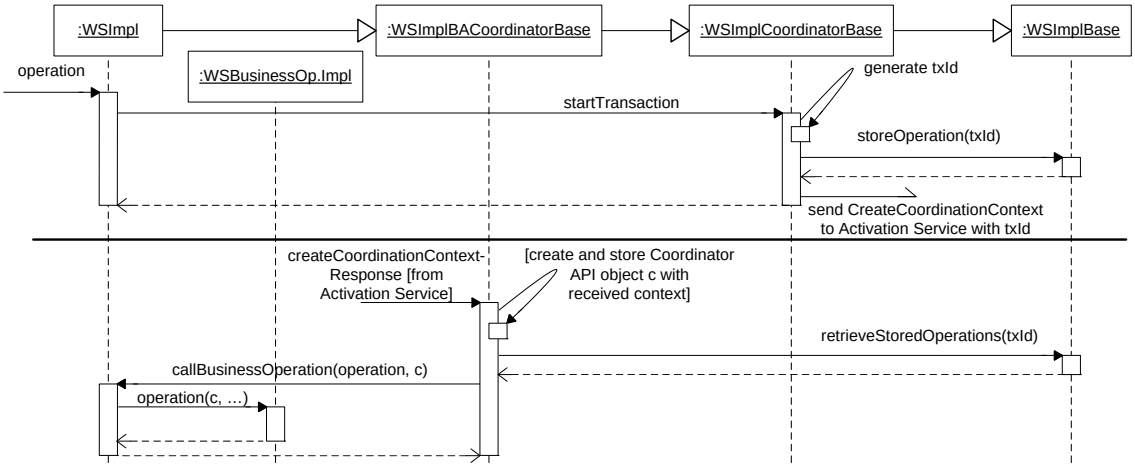

Abbildung 5.16.: Aufruf Aktivierungs-Service

`WebServiceImplParticipantBase` implementiert Funktionalität eines Teilnehmers. Folgende Methoden sollten auf dieser Ebene implementiert werden:

registerParticipant: Diese Methode ruft den Registrierungs-Service auf. Sie wird von der anwendungsspezifischen Framework-Logik der Web-Service-Operationen (s. 5.2.5.4) aufgerufen, wenn der Teilnehmer das erste Mal innerhalb einer Transaktion aufgerufen wurde. Der ursprüngliche Aufruf wird zwischengespeichert, außerdem wird ein

[11] Für alle folgenden Sequenzdiagramme gilt: Die halben Pfeile stellen asynchrone Web-Service-Aufrufe (z. B. wie in 5.2.3 beschrieben) dar, der waagerechte Balken bedeutet eine mögliche neue Instantiierung der Web-Service-Implementierungsklassen bei einem erneuten Aufruf.

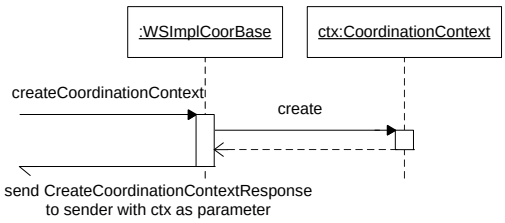

Abbildung 5.17.: Aktivierungs-Service

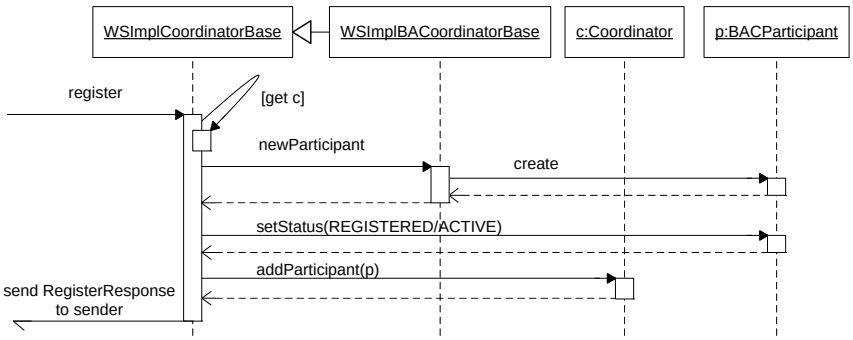

Abbildung 5.18.: Registrierungs-Service

API-Objekt für den Teilnehmer aus Teilnehmersicht erzeugt, welches sich zunächst im Zustand `REGISTERING` befindet. Da auf dieser Ebene nicht bekannt ist, welches Terminierungsprotokoll benutzt wird, wird die Erzeugung über die folgende Hook-Methode vorgenommen, die auf dieser Ebene abstrakt deklariert wird. Abb. 5.19 verdeutlicht den Vorgang.

newParticipant: Diese Hook-Methode dient zur Erzeugung eines neuen API-Objekts für einen Teilnehmer aus Teilnehmersicht.

5.2.5.3. WS-BusinessActivity

Abb. 5.23 auf Seite 105 und Abb. 5.24 auf Seite 106 zeigen die benötigten Implementierungsklassen für einen Koordinator und einen Teilnehmer, die das Terminierungsprotokoll WS-BusinessActivity benutzen.

Die Web-Service-Operationen des Protokolls werden dabei von den auf `Base` endenden allgemeinen Klassen zur Verfügung gestellt, die jeweils im oberen Bereich der Abbildung zu finden sind. Für andere Terminierungsprotokolle als WS-BusinessActivity wären diese Klassen entsprechend zu ersetzen.

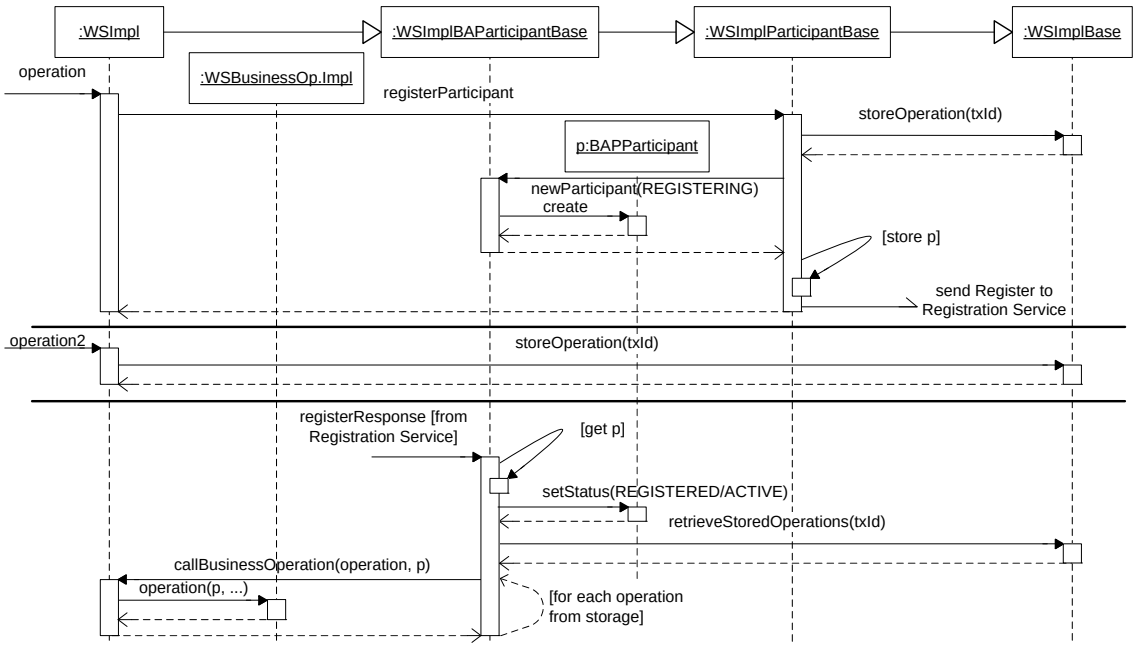

Abbildung 5.19.: Aufruf Registrierungs-Service

Diese erweitern die entsprechenden WS-Coordination-Klassen. Auf dieser Ebene können die `newParticipant()` Hook-Methoden implementiert werden, da bekannt ist, dass die API-Klassen `BACParticipant` (s. 5.2.1.2) bzw. `BAPParticipant` (s. 5.2.1.3) verwendet werden müssen. Das konkrete Entwurfsmuster ist hier Factory Method (Gamma u. a., 1995).

Das Interface `Timeable` wird implementiert, damit der Web Service als Callback für Timeouts (s. 5.2.2) dienen kann. Die entsprechende Methode `timeout()` sollte auf dieser Ebene feststellen, ob der Timeout beim Warten auf eine Protokoll-Nachricht oder eine Anwendungs-Nachricht eingetreten ist. Im Falle eine Anwendungsnachricht muss eine anwendungsspezifische Methode aufgerufen werden, die hier als abstrakte Hook-Klasse `businessTimeout()` dargestellt ist, welche von einem konkreten Web Service implementiert werden muss.

Das Interface `Recoverable` wird implementiert, um den Service als wiederherstellbar zu kennzeichnen. Die Methode `recover()` muss möglichst erreichen, dass alle im persistenten Speicher befindlichen Transaktionen zumindest auf Protokollebene fortgesetzt werden können. Dabei muss der Anwendungslogik auch die Möglichkeit gegeben werden, eine Wiederherstellung durchzuführen, was durch eine Hook-Methode `businessRecover()` angedeutet wird, welche von einem konkreten Web Service implementiert werden muss (s. u., zur Wiederherstellung siehe 4.2.1.4).

Als weitere Hook-Methode für die Anwendungslogik ist `callBusinessOperation()` vorgesehen. Diese funktioniert analog zur Methode `callOperation()` im Framework (Schrörs, 2006a) (s. (Schrörs, 2006c, Kap. 3)) und dient ebenfalls zur Emulation von Java-Reflection, ist aber für den Fall vorgesehen, dass ein zwischengespeicherter Operationsaufruf nach Aktivierung (auf Koordinatorseite) bzw. Registrierung (auf Teilnehmerseite) nun ausgeführt wird. Dabei werden gleich entsprechende API-Objekte zur Benutzung durch die Anwendungslogik übergeben.

Weiterhin wird die Transaktionslogik für alle benötigten Protokoll-Operationen für WS-BusinessActivity implementiert. Diese ruft die in 5.2.1.1 und 5.2.1.3 beschriebenen Listener-Methoden der Anwendungslogik auf.

Die Koordinator-Klasse `WebServiceImplBACoordinatorBase` implementiert außerdem die Callback-Operation für den Aktivierungs-Service `createCoordinationContextResponse()`, welche das API-Objekt vom Typ `Coordinator` erzeugt und die Anwendungslogik der zwischengespeicherte Operation mittels `callBusinessOperation()` aufruft (s. Abb. 5.16 oben).

Außerdem wird mit `invokeCallbackBusinessOperation()` eine allgemeine Hilfsmethode für alle diejenigen konkreten Web-Service-Operationen zur Verfügung gestellt, die Callbacks von Teilnehmern darstellen. Die Methode holt die benötigten API-Objekte aus dem persistenten Speicher und übergibt sie der Anwendungslogik des Callbacks mittels `callBusinessOperation()` (Abb. 5.20).

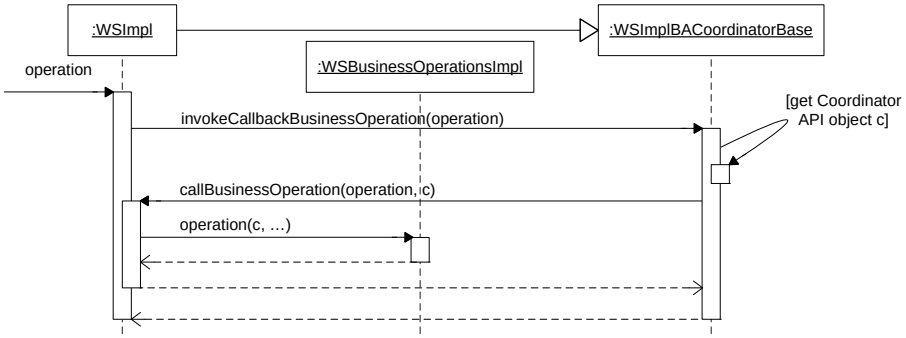

Abbildung 5.20.: Aufruf Callback-Operation

Die Teilnehmer-Klasse `WebServiceImplBAParticipantBase` implementiert analog dazu die Callback-Operation für den Registrierungs-Service `registerResponse()`, welche den Status des zuvor erzeugten `BAPParticipant`-Objekts auf `REGISTERED` setzt und die Anwendungslogik der gepufferten Operationen mittels `callBusinessOperation()` aufruft (s. Abb. 5.19 oben).

Außerdem wird mit `registerParticipantOrInvokeBusinessOperation()` eine allgemeine Hilfsmethode für Web-Service-Operationen zur Verfügung gestellt, die über-

prüft, ob der Teilnehmer für die Transaktion, deren Kontext als Header im Aufruf enthalten war, bereits registriert ist. Ist dies der Fall, so holt die Methode die benötigten API-Objekte aus dem persistenten Speicher und übergibt sie der Anwendungslogik der Operation mittels `callBusinessOperation()`. Dies verläuft analog zum Aufruf einer Callback-Operation beim Koordinator (s. Abb. 5.20 oben).

Ist der Teilnehmer noch nicht registriert und auch nicht im Zustand `REGISTERING`, so wird zur Registrierung die Methode `WebServiceImplParticipantBase.registerParticipant()` aufgerufen.

Ist der Teilnehmer im Zustand `REGISTERING`, so läuft die Registrierung bereits und der Aufruf der Anwendungslogik wird zwischengespeichert, um dann nach dem Eintreffen der `RegisterResponse`-Nachricht ausgeführt zu werden (s. Abb. 5.19 oben).

5.2.5.4. Konkrete Anwendung

Alle bisher beschriebenen Klassen waren im Sinne der Unterteilung 2 oben „allgemein“ und können komplett vom Framework implementiert werden. In diesem Abschnitt wird schließlich anhand des Beispiels des mobilen Freigabemanagers (s. 1.2 und 2.3.2) beschrieben, wie Implementierungsklassen für konkrete Web Services aussehen müssen. Dabei kommt eine zweistufige Hierarchie mit Delegation zum Einsatz, um anwendungsspezifische Framework-Funktionalität und die eigentliche Anwendungslogik zu trennen. Dadurch werden Erweiterbarkeit und Wartbarkeit gefördert.

Im Beispiel implementiert der Koordinator einen Geschäftsprozess, im Laufe dessen Freigaben (Approvals) von einem mobilen Entscheider benötigt werden, auf dessen mobilen Gerät der Freigabe-Service als Teilnehmer ausgeführt wird. Der Vorgang wird in Abb. 5.21 dargestellt wobei links die Koordinatorlogik und rechts die Teilnehmerlogik dargestellt ist. Die zugehörigen Klassen werden weiter unten beschrieben.

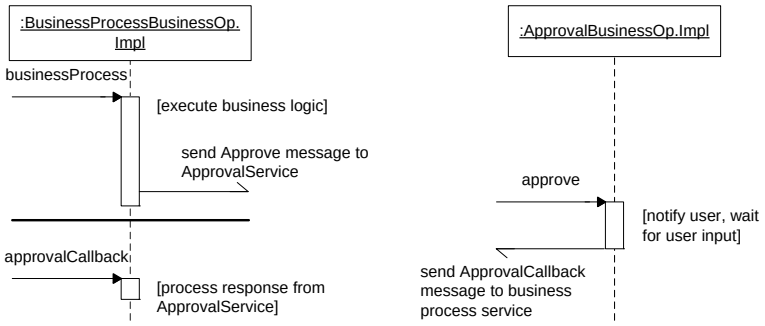

Abbildung 5.21.: Ablauf des Freigabevorgangs (Beispiel-Services)

Die Klasse `BusinessProcessImpl` in Abb. 5.23 (S. 105) zeigt die Implementierungsklasse des Koordinators. Es werden alle noch ausstehenden Methoden des Frameworks implementiert (`getServiceNamespace()`, `...Endpoint()` und `...Instance()`) sowie die benötigten Web-Service-Operationen, welche jedoch auf dieser Ebene zunächst noch Framework-Logik enthalten:

businessProcess: Diese Methode startet eine neue Transaktion mittels Aufruf von `WebServiceImplCoordinatorBase.startTransaction()` (s. 5.2.5.2).

approvalCallback: Diese Methode ruft `WebServiceImplBACoordinatorBase.invokeCallbackBusinessOperation()` auf (s. 5.2.5.3).

Die Klasse `BusinessProcessBusinessLogicImpl` enthält ausschließlich Anwendungslogik. Die Methoden werden jeweils von Framework-Methoden aufgerufen. `businessProcess()` führt u. a. einen Web-Service-Aufruf zum Freigabe-Service mit einer `Approve`-Nachricht durch, `approvalCallback()` verarbeitet die asynchrone Antwort des Freigabe-Service (s. u.). `businessTimeout()` und `businessRecover()` werden vom Framework aufgerufen, wenn ein Timeout auftritt bzw. wenn der SOAP-Server gerade neu gestartet wurde. Hier können spezifische Timeout- und Wiederherstellungsoperationen der Geschäftslogik implementiert werden. Für andere Terminierungsprotokolle als WS-BusinessActivity wären die Listener-Methoden (`received...()`) zu ersetzen.

Die Klasse `ApprovalImpl` in Abb. 5.24 (S. 106) zeigt die Implementierungsklasse des Teilnehmers. Es werden analog zum Koordinator ebenfalls alle noch ausstehenden Methoden des Frameworks implementiert. Die einzige benötigte Web-Service-Operation ist hier

approve: Diese Methode ruft auf dieser Ebene zunächst nur `WebServiceImplBAParticipantBase.registerParticipantOrInvokeBusinessOperation()` auf, welche eine Registrierung durchführt, falls noch nicht geschehen (s. 5.2.5.3).

Die Klasse `ApprovalBusinessLogicImpl` enthält analog zur Koordinatorseite ebenfalls ausschließlich Anwendungslogik. Die Methoden werden jeweils von Framework-Methoden aufgerufen. `approve()` wartet auf die Reaktion des Benutzers auf eine Freigabe-Anforderung und führt anschließend einen Web-Service-Aufruf zum Koordinator mit einer `ApprovalCallback`-Nachricht durch. Die Methoden `businessTimeout()` und `businessRecover()` haben die gleiche Semantik wie oben für den Koordinator-Service beschrieben. Für andere Terminierungsprotokolle als WS-BusinessActivity wären hier ebenfalls die Listener-Methoden (`received...()`) zu ersetzen.

5.2.5.5. Terminierung

Abschließend verdeutlicht Abb. 5.22 die Interaktion der Komponenten bei der Terminierung unter Benutzung des Protokolltyps ParticipantCompletion. Nachdem das Ergebnis der Freigabeanfrage den Geschäftsprozess gesendet wurde, wird `Completed` an den Koordinator (in diesem Fall ebenfalls die Implementierung des Geschäftsprozesses) gesendet. Die Implementierung von `receivedCompleted()` muss überprüfen, ob die Transaktion beendet werden kann. Ist dies der Fall, so sendet der Koordinator die Nachricht `Close` an alle registrierten Teilnehmer. Diese Antworten jeder mit `Closed`, um das Protokoll zu beenden (nicht mehr in der Abbildung gezeigt).

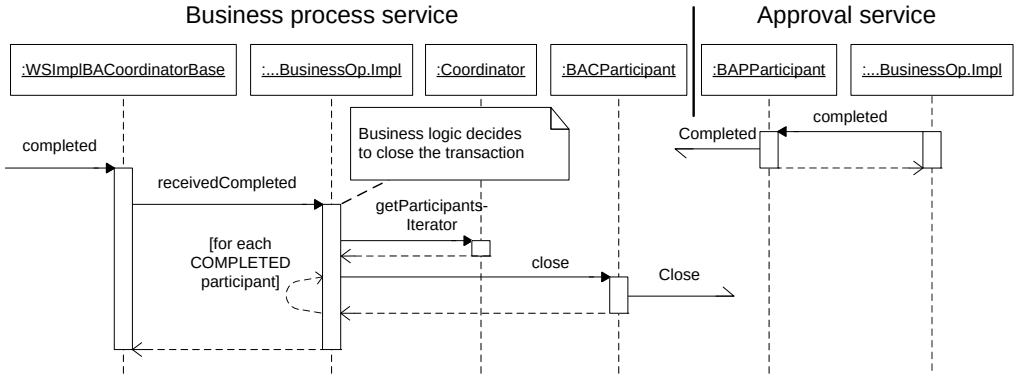

Abbildung 5.22.: Terminierung

Analog lassen sich Interaktionen mit den Nachrichten `Cancel` (Zurückziehen der Freigabeanfrage) und `Compensate` (Informieren des mobilen Benutzers darüber, dass die Freigabeanfrage obsolet ist, obwohl sie bereits beantwortet wurde) vorstellen.

5.2.5.6. Allgemeines zur Web Service Implementierung

Im Rahmen der Forderung nach Skalierbarkeit wurde in 4.2.3 auf den Web Service Scope einer Implementierung hingewiesen. Dieser bezeichnet im Wesentlichen, ob ein Objekt der Implementierungsklasse eines Web Services nur für einen Aufruf (man spricht dann von Request-Scope) oder für mehrere (Session- oder Application-Scope) verwendet wird.

Im ursprünglichen Framework ist nur Request-Scope möglich, da der Aufrufkontext (eine Instanz der Klasse `MessageContext`, zur Verwendung s. a. 5.2.2 und 5.2.4) als Instanzvariable der Implementierungs-Wurzelklasse `serviceframework.WebService` implementiert ist, womit die Klasse nicht Reentrance-fähig ist.

Bei der Implementierung der beschriebenen Erweiterungen sollte dagegen darauf geachtet werden, dass alle benötigten Informationen nur als Aufrufparameter von Methoden modelliert

werden. Das Ziel dieser Vorgehensweise ist, flexibel bezüglich zukünftiger Änderungen in Richtung der möglichen Scopes zu bleiben.

5.2.6. Code-Generator

Bei der Entwicklung des Frameworks sind die Operationen konkreter transaktionaler Web Services noch nicht bekannt. Da diese aber vor der Ausführung der eigentlichen Geschäftslogik noch Vorgänge wie Aktivierung oder Registrierung durchführen müssen, sollte der dafür zuständige Code generiert werden (s. 4.4).

Da der Anwendungsentwickler den generierten Code möglichst nicht sehen soll, empfiehlt sich die Einführung eines zusätzlichen Delegationsschrittes. Die eigentliche Implementierungsklasse des Web-Service wird komplett unter Angabe geeigneter Parameter generiert und enthält die anwendungsspezifische Framework-Logik (`...Impl`). Diese würde zur Laufzeit die gleichnamigen Methoden in der Implementierungsklasse für die Geschäftslogik (`...BusinessLogicImpl`) aufrufen. Für diese Klasse können die Methode `callBusinessOperation()` komplett (als Broker für alle anderen Methoden) und alle anderen Methoden als leere Rümpfe generiert werden. Der Anwendungsentwickler füllt dann „nur noch“ diese leeren Methodenrümpfe.

Im Rahmen dieser Arbeit wurde aus Zeitgründen weder ein detaillierter Entwurf für einen konkreten Code-Generator erarbeitet noch wurde ein solcher realisiert. `WSDL2Java` aus Apache Axis (Axis, 2006) ist jedoch leicht erweiter- bzw. änderbar. Ein möglicher Ansatz wäre daher, diesen Generator um die benötigten Fähigkeiten zu erweitern.

5.3. Exception Handling

An jeder Stelle des Frameworks und der Anwendung können Fehlerbedingungen auftreten. Sofern nicht mit diesen Fehlern gerechnet wird (etwa mit dem Fehlschlagen von Kommunikationsversuchen, s. 4.2.1), führen sie in Java-Laufzeitumgebungen zu Exceptions, die von der Stelle des Auftretens solange die Aufrufkette von Methoden zurückgereicht werden, bis sie irgendwo behandelt werden.

Das ursprüngliche Framework (Schrörs, 2006a) sieht vor, dass Exceptions von der Web-Service-Implementierung abgefangen werden und nur dann als neue Exception vom Typ `SOAPException` an das Framework zurückgegeben werden, wenn der Absender des Web-Service-Aufrufs über den Fehler informiert werden soll. An jeder Stelle der Aufrufkette

kann die Exception abgefangen werden und es kann überprüft werden, ob das Problem reparabel ist. Ist dies nicht der Fall, so können der Exception Fehlerinformationen hinzugefügt werden, bevor sie erneut geworfen (d. h. in der Aufrufkette weiter nach oben gereicht) wird.

Die Klasse `SOAPException` stellt spezielle Operationen zur Verfügung, um aus einem Exception-Objekt eine SOAP-Fault-Nachricht (s. (SOAP, 2003)) zu generieren, die dann als Antwort (z. B. als Body einer HTTP-Response) an den Absender zurückgeschickt werden kann. Die Syntax von SOAP-Faults erlaubt das Kapseln beliebig vieler Fehlerinformationen.

Die Spezifikationen WS-Coordination und WS-BusinessActivity sehen verschiedene Fehlernachrichten vor, die auf diese Art und Weise verschickt werden könnten.

Das Problem der von den Spezifikationen empfohlenen asynchronen Aufrufe ist jedoch, dass der Absender nach einem Aufruf nicht mehr auf eine Antwort wartet. WS-Addressing (WS-Addr, 2004) sieht diesbezüglich vor, dass in den SOAP-Headern `FaultTo` und `ReplyTo` vom Absender festgelegt wird, wohin Fehlernachrichten bzw. Antworten geschickt werden. Zumindest sollte die Absenderadresse als `From`-Header mitgeschickt werden (s. 4.1.5).

Das Framework muss dafür so verändert werden, dass die WS-Addressing-Informationen im Fall von asynchronen Aufrufen zwischengespeichert werden. Tritt dann eine `SOAPException` auf, so wird ein entsprechender neuer asynchroner Web-Service-Aufruf mit dem aus der Exception generierten SOAP-Fault-Element durchgeführt.

5.4. Fazit

In den vorangegangenen Abschnitten wurde eine Möglichkeit zur Umsetzung der in Kapitel 4 erarbeiteten Anforderungen detailliert entworfen. Dabei wurde das bereits existierende Framework aus (Schrörs, 2005) und (Schrörs, 2006a) als Grundlage verwendet. Dieses Framework hat sich als leicht erweiterbar erwiesen. Es sind zur Implementierung des Transaktions-Frameworks nur wenige abwärtskompatible kleine Änderungen bzw. Erweiterungen nötig:

- Über den Aufrufkontext muss auf den SOAP-Server und damit auf Persistenz- und Timer-Objekte zugegriffen werden können.
- Soll eine Instanz einer Web-Service-Implementierungsklasse nicht nur im Request-Scope (s. 4.2.3, Skalierbarkeit) verwendet werden, so dürfte der Aufrufkontext nicht als Instanzvariable der Implementierungs-Wurzelklasse (`serviceframework.WebService`) modelliert werden.
- Das Exception-Handling mit SOAP-Faults muss für asynchrone Aufrufe gemäß WS-Addressing erweitert werden.

Die Erweiterungen des ursprünglichen Frameworks wurden unter Verwendung einiger für die Förderung von Erweiterbarkeit und Wartbarkeit besonders geeigneter Entwurfsmuster modelliert. Dabei muss zwischen Erweiterungen des Frameworks um Framework-Komponenten zwecks Verwendung des Frameworks für andere Koordinationsprotokolle und Erweiterungen des Frameworks um Komponenten zwecks Realisierung einer konkreten Anwendung unterschieden werden.

Die Timer-, Call-, und Persistenzkomponenten sind nur zur Erweiterung um Framework-Komponenten geeignet, während sie von konkreten Anwendungen (Web-Service-Implementierungen) lediglich benutzt werden. Die verwendeten Entwurfsmuster fallen unter die Metakategorien 1:1 Connection (Callback, Strategy, Builder) und Unification (Abstract Factory).

Die Basisklassen für die Implementierung transaktionaler Web-Services sind dagegen erweiterbar sowohl um Framework-Komponenten (Klassen für neue Protokolle) als auch um Anwendungskomponenten in Form von konkreten Implementierungen. Da sich hierbei alles innerhalb einer Vererbungshierarchie abspielt, kommen vor allem Entwurfsmuster der Kategorien Unification (in einer Superklasse wird eine abstrakte Methode aufgerufen, die in einer Subklasse protokoll- oder anwendungsspezifisch implementiert ist) und 1:1 Recursive Connection (Chain of Responsibility) vor.

Durch Verwendung der Muster erfolgt eine saubere Trennung von Verantwortlichkeiten der einzelnen Protokollschichten SOAP, WS-Coordination, WS-BusinessActivity und Anwendung. Es wird viel Logik in prinzipiell voneinander unabhängigen Framework-Schichten implementiert während weitere, anwendungsspezifische, Framework-Logik generiert wird. Der Entwickler der Geschäftslogik hat auf diese Weise nur dann Verantwortlichkeiten hinsichtlich der korrekten Implementierung des Terminierungsprotokolls, wenn das verwendete Protokoll dies fordert.

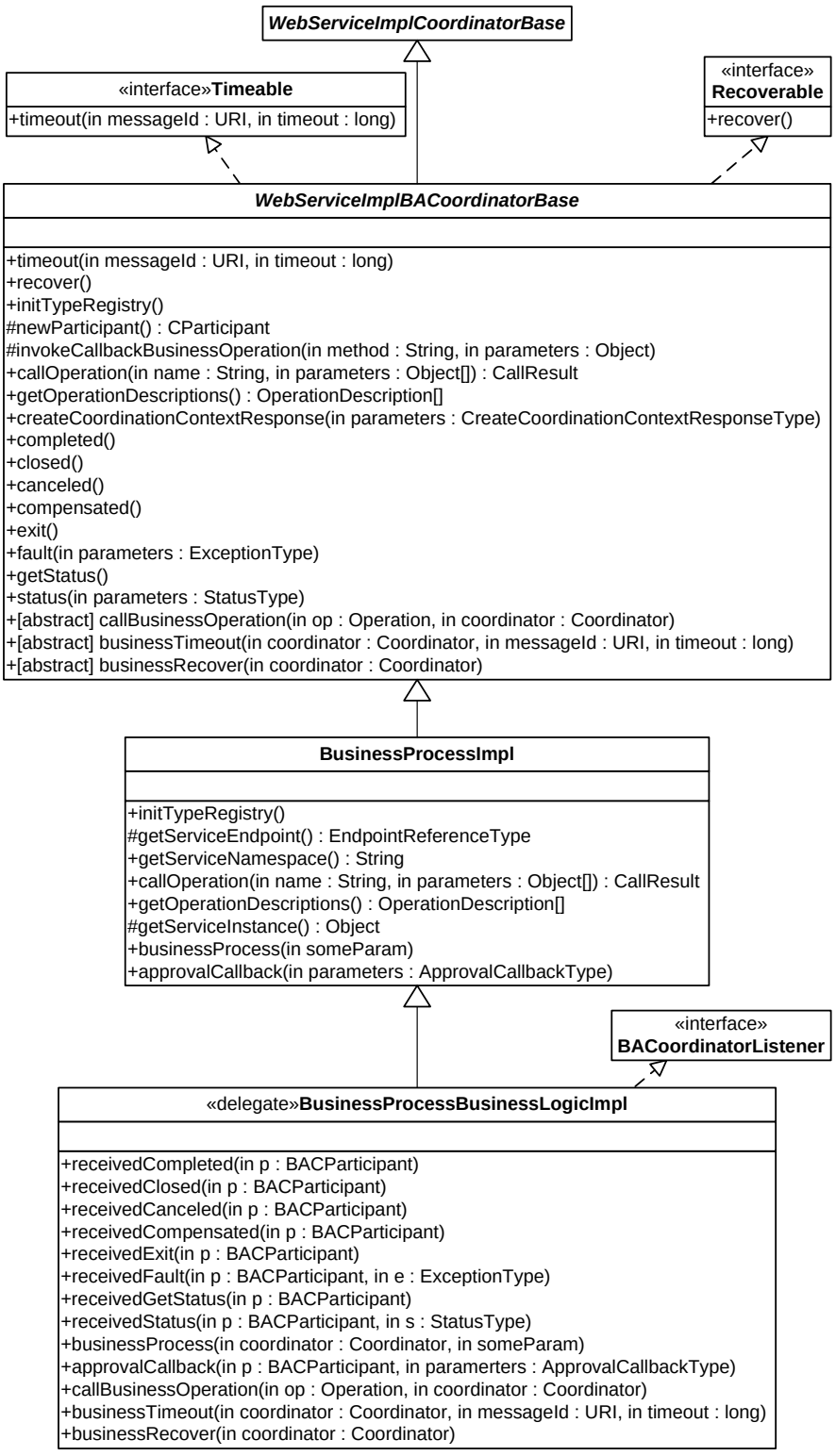

Abbildung 5.23.: Web Service Implementierung – WS-BA Koordinator

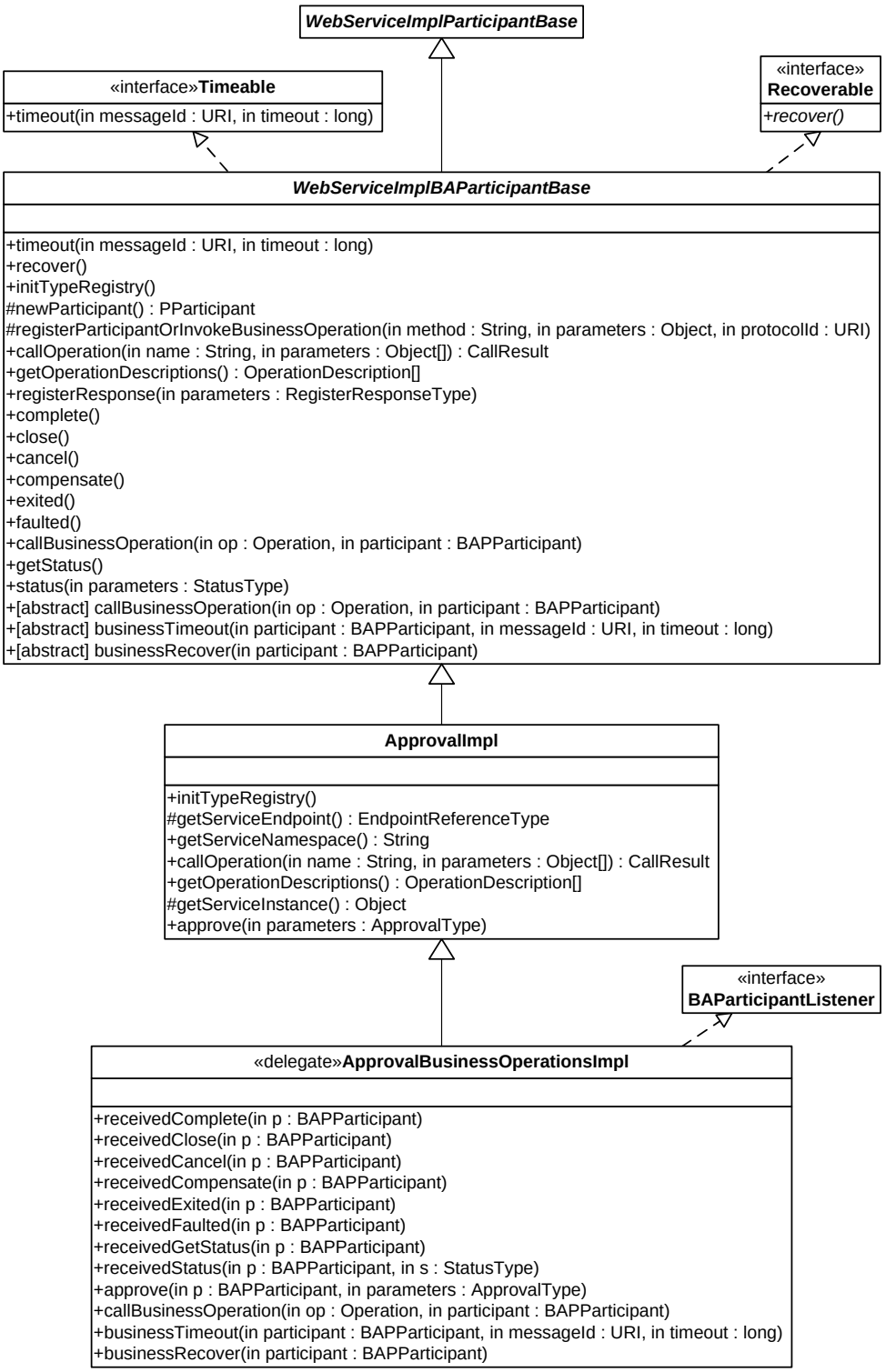

Abbildung 5.24.: Web Service Implementierung – WS-BA Teilnehmer

6. Implementierung

In diesem Kapitel wird beschrieben, welche der im vorigen Kapitel entworfenen Komponenten umgesetzt wurden und bis zu welchem Fertigstellungsgrad dies geschehen ist. Ziel konnte dabei im Rahmen der zur Verfügung stehenden Zeit nicht sein, eine vollständige, voll funktionsfähige Version des Transaktions-Frameworks zu erstellen. Vielmehr stand im Mittelpunkt, die prinzipielle Realisierbarkeit der Entwürfe zu überprüfen und Beispiel-Code für zukünftige Entwicklungen zu liefern.

Es wird zunächst kurz auf die Entwicklungs- und Testumgebung sowie auf das Beispielszenario eingegangen (6.1), bevor die Realisierung des Frameworks beschrieben wird (6.2). Abschließend folgt eine kurze Betrachtung technischer Probleme (6.3).

Auf der beiliegenden CD sind alle für die Einrichtung der Entwicklungs- und Testumgebung benötigten Softwarepakete, der Sourcecode in Form von Eclipse-Projekten sowie Installationsanleitungen zu finden. Siehe dazu auch Anhang B.

6.1. Entwicklungs- und Testumgebung

Da das Beispielszenario aus einem stationären Koordinator und einem mobilen, ressourcenbeschränkten Teilnehmer besteht, musste neben dem mobilen Web Service auch ein stationärer Teil entwickelt werden. Dazu wurde ein vom Autor im Rahmen eines Studienprojektes entwickeltes Framework (Gerlach, 2006) verwendet und der in dem dortigen Beispiel-Code enthaltene Koordinator-Service leicht abgeändert, so dass dieser neben anderen Beispiel-Teilnehmern auch den mobilen Freigabe-Service aufruft und auf eine Rückmeldung wartet.

Die Entwicklungsumgebung bestand aus je einer Installation von Eclipse 3.1 (Eclipse, 2006) für stationären und mobilen Teil (Abb. 6.1).

Für den stationären Teil wurde eine leicht veränderte und im Rahmen des o. g. Projekts um WS-Coordination- und WS-BusinessActivity-Fähigkeiten erweiterte Version von Apache Axis 1.2.1 (Axis, 2006) verwendet, die im JBoss 4.0.2 Application Server (JBoss, 2006) installiert wurde. Der Server war durch die JBoss IDE für Eclipse (JBoss-Eclipse, 2006) einfach aus der Eclipse-Entwicklungsumgebung heraus konfigurierbar und steuerbar.

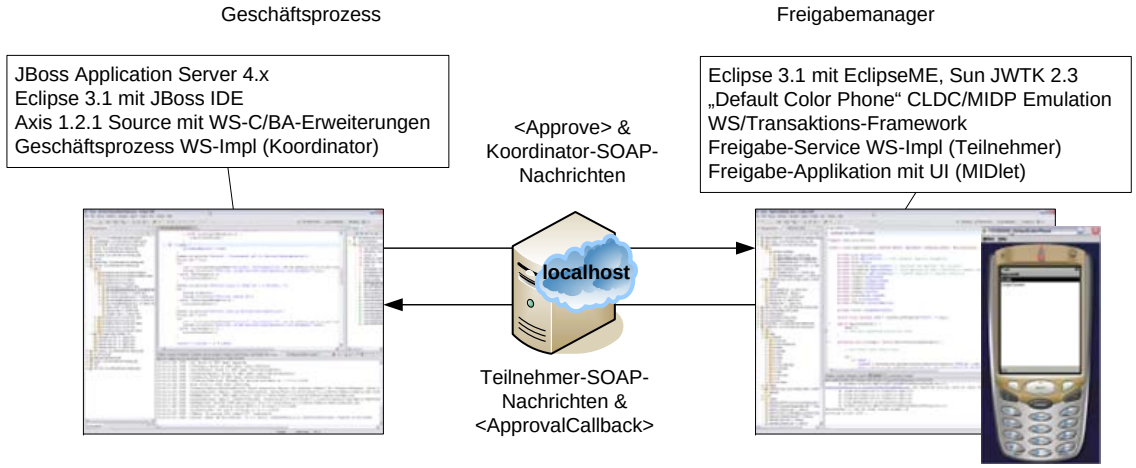

Abbildung 6.1.: Entwicklungsumgebung (eigene Darstellung)

Zur Entwicklung des mobile Teils wurde das Sun Java Wireless Toolkit for CLDC (Sun-JWTK, 2006) in der Version 2.3 verwendet. Das Toolkit stellt innerhalb der Eclipse-Entwicklungsumgebung die benötigten Bibliotheken und Werkzeuge zur Verfügung, um J2ME-Anwendungen für verschiedene Versionen und Profile der J2ME Connected Limited Device Configuration (CLDC) zu entwickeln. Es enthält z. B. auch einen Emulator für das Mobile Information Device Profile (MIDP) in Version 2.0, der die Laufzeitbedingungen auf einem Mobiltelefon nachstellt. Das Web-Service-Framework aus (Schrörs, 2006a) wurde mit der Vorgängerversion dieses Toolkits entwickelt.

Die Teststellung bestand aus einem beliebigen J2EE Application Server zur Ausführung des stationären Teils, sowie einem WLAN-fähigen PDA für den mobilen Teil (Abb. 6.2).

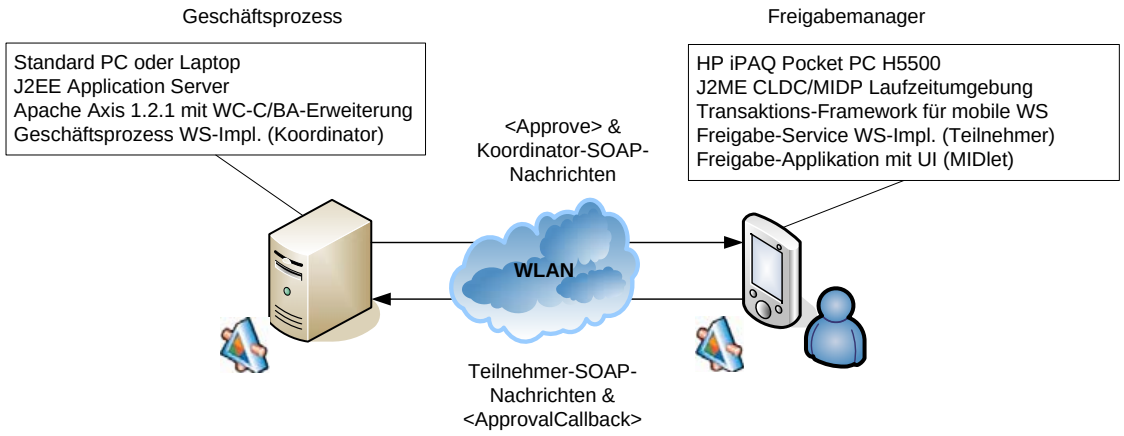

Abbildung 6.2.: Testumgebung (eigene Darstellung)

Im Application Server musste eine Webanwendung bestehend aus dem erweiterten Apache Axis und den an das Freigabemanager-Szenario angepassten Web Services aus dem o. g. Studienprojekt installiert werden. Der Einfachheit halber wurde die gleiche Eclipse-JBoss-Umgebung verwendet wie für die Entwicklung. Der angepasste Koordinator-Service muss durch eine separate Anwendung aufgerufen werden. Dieser startet bei jedem Aufruf eine langlebige Transaktion, sendet eine Freigabeanfrage an den mobilen Teil und ruft einige weitere (Dummy-)Teilnehmer auf. Wurde von allen Teilnehmern ein Ergebnis empfangen, wird die Transaktion beendet.

Der mobile Teil der Beispielanwendung wurde in Form von MIDlets (s. (Li und Knudsen, 2005, Kap. 1-3) und (Schrörs, 2005, 6.3)) auf dem PDA installiert. Zur Verwendung kam ein von der Hochschule für Angewandte Wissenschaften Hamburg zur Verfügung gestellter HP iPAQ Pocket PC H5500 mit dem Betriebssystem Windows Pocket PC 2003 und verschiedenen Java Virtual Machines wie in (Schrörs, 2005, 6.2) beschrieben.

Die mobile Anwendung besteht neben dem mit dem Transaktions-Framework laufenden Web Service, welcher genauso initialisiert und gestartet wird wie Web Services ohne Transaktionen (s. (Schrörs, 2005, 6.1) bzw. (Schrörs, 2006c, Kap. 3)), aus einem einfachen MIDlet, welches eine Liste der aktuell anliegenden Freigabeanfragen darstellt (Abb. 6.3 links). Diese sind einzeln anwählbar. In einem separaten Screen kann die angewählte Anfrage genehmigt, abgelehnt, delegiert oder verschoben werden und es kann ein kurzer Begründungstext eingeben werden (Abb. 6.3 rechts).

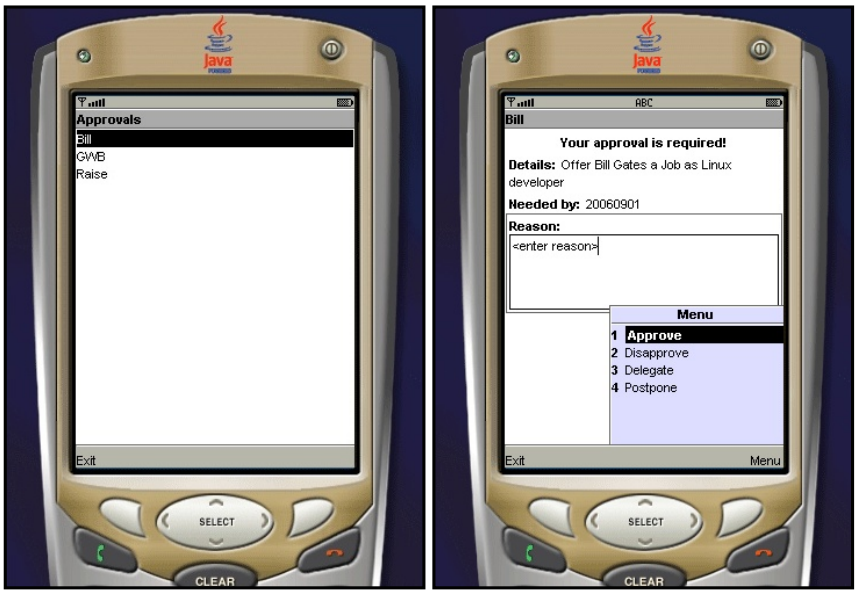

Abbildung 6.3.: UI der Beispielanwendung im Sun Java WTK Emulator (eigene Screenshots)

Das MIDlet und der Web Service kommunizieren über einen gemeinsam genutzten (shared) MIDP RecordStore (s. (Li und Knudsen, 2005, Kap. 8)) miteinander. Empfängt der Web Service eine Anfrage, so speichert er diese im RecordStore und das MIDlet aktualisiert die Liste, so dass der Benutzer die Anfrage bearbeiten kann. Wird eine Anfrage im Speicher verändert, so sendet der Web Service die Entscheidung und den Begründungstext zurück an den Koordinator.

Der Ansatz, den Koordinator stationär zu implementieren, folgt einer im abschließenden Kapitel beschriebenen Konfigurationsempfehlung, die auf den Ergebnissen dieser Arbeit beruht (s. 7.1.4).

6.2. Realisierte Framework-Komponenten

Die Komponenten des Transaktions-Frameworks wurden gemäß des Entwurfs in Kapitel 5 in Java implementiert. Im Folgenden werden die Fertigstellungsgrade aller Komponenten aufgelistet und auf offene Punkte bzw. noch zu vervollständigende Funktionalität hingewiesen.

WS-C & WS-BA (I) – API Implementierung Das API für WS-Coordination und WS-BusinessActivity ist vollständig gemäß Analyse und Entwurf implementiert. Hinsichtlich der verwendeten Datentypen ist zu bemerken, dass das Framework keine Serialisierung und Deserialisierung von beliebigen XML-Kindelementen und -Attributen erlaubt (XML Schema Typ „any“). Weiterhin wurden einige Datentypen (z. B. `EndPointReferenceType` und `CoordinationContext`) nur unvollständig, für die Zwecke dieser Arbeit jedoch ausreichend implementiert.

Timer-Handling Die Timer-Komponente ist soweit implementiert, dass Timer verwendet werden können, solange der SOAP-Server nicht neu gestartet wird. Da noch keine produktiv verwendbare Persistenzschicht implementiert wurde (s. u.), werden Timer nicht persistiert und demzufolge auch nicht nach einem Neustart wieder hergestellt.

WS-Addressing und Calls Die Erweiterungen der `Call`-Klasse des ursprünglichen Frameworks wurden wie entworfen vollständig implementiert. Die WS-Addressing- und WS-Coordination-Header werden automatisch für die Serialisierung registriert.

Es wurde bisher nur eine synchrone Aufrufstrategie zu Testzwecken implementiert. Die statische Methode `CallStrategy.getDefaultCallStrategy` gibt immer eine Instanz dieser Strategie zurück, die genau einen Sendeversuch und keinen Timeout vorsieht, da

das Timeout-Handling der Transaktionslogik nicht implementiert wurde (s. u.). Diese Factory-Methode wäre ein Ansatz für eine variable, konfigurierbare Erzeugung von Aufrufstrategien für verschiedene Online/Offline-Situationen.

Persistenz Als einzige Ausprägung des Persistenz-APIs wurde eine einfache Factory-Klasse, `SimpleStorageFactory`, nebst zugehörigem Konverter implementiert, welcher die zu persistierenden Objekte einfach in einer Standard-`Hashtable` speichert. Dieser Speicher besteht nicht über die Lebensdauer der Java Virtual Machine hinaus und ist damit für Wiederherstellungszwecke ungeeignet, reichte aber zum Testen der grundlegenden Funktionen aus. Eine „echte“ Implementierung wäre für den Umfang dieser Arbeit zu komplex gewesen.

WS-C & WS-BA (II) – Web Service Implementierung Die Web-Service-Basisklassen wurden soweit implementiert, dass die Protokolllogik bis auf Ausnahmefälle wie Timeouts, fehlgeschlagene Sendeversuche und andere Fehlerbedingungen dem Entwurf und damit den Spezifikationen entspricht. Das Timeout-Handling für WS-Coordination- und WS-BusinessActivity-Nachrichten wurde nicht implementiert, da dies für diese Arbeit zu komplex gewesen wäre. Die Default-Aufrufstrategie sieht dies auch nicht vor (s. o.).

Die Wiederherstellung von Transaktionsdaten nach dem Neustart des SOAP-Servers wurde ebenfalls nicht implementiert, da dies mit der vorliegenden Persistenzimplementierung nicht funktionieren würde und auch zu komplex gewesen wäre. Infolgedessen wird das Abschalten eines mobilen Gerätes während einer langlebigen Transaktion nicht unterstützt. Dies wäre für einen produktiven Einsatz jedoch unerlässlich und sollte daher bei einer Weiterentwicklung oberste Priorität haben.

Code-Generator Ein Code-Generator wurde aus Zeitgründen nicht implementiert.

Exception Handling Exceptions werden zwar abgefangen und als SOAPException weiter an den Kern des Frameworks gereicht. Die Exceptions werden aber nicht genau ausgewertet und die SOAPExceptions entsprechen nicht den in den Spezifikationen vorgesehenen Fehlernachrichten. Die SOAPExceptions werden weiterhin nicht asynchron als SOAP-Faults an den Absender der ursprünglichen Nachricht verschickt.

6.3. Technische Probleme

Nennenswerte technische Probleme gab es hauptsächlich im Zusammenhang mit den verschiedenen Java Virtual Machines (JVMs), welche für den HP iPAQ verfügbar sind. Das bereits in (Schrörs, 2005, 6.2) beschriebenen Verhalten der JVMs Jeode von Insignia (mit dem iPAQ mitgeliefert), J9 von IBM (J9, 2006) und CrEme von NSICOM (CrEme, 2006) konnte bestätigt werden.

Zusätzlich war für das Beispielszenario noch gefordert, dass zwei Java-Prozesse gleichzeitig ausgeführt werden, nämlich die UI-Anwendung und der Web Service zusammen mit dem Framework, wobei die Konsolenausgaben von Web Service und Framework nach Möglichkeit sichtbar sein sollten. Dies erwies sich sowohl für die J9 JVM als auch für die CrEme JVM überraschend als nicht möglich, da ein Shortcut auf das JVM-Executable lediglich dem bereits laufenden Prozess den Fokus gibt anstatt einen neuen zu starten. Mit der Jeode JVM funktionierte dies: Es wurden zwei JVMs gestartet, jede in einem eigenen Fenster.

Weiterhin basiert die Beispielanwendung auf der Kommunikation über MIDP RecordStores. Da die drei genannten JVMs jedoch die ehemals als „Personal Java“ bekannte CDC (JSR-218, 2005) mit „Personal Profile“ PP (JSR-216, 2005) implementieren, muss die RecordStore-Komponente emuliert werden, was dazu führt, dass die Kommunikation über einen gemeinsam genutzten RecordStore nur innerhalb des selben JVM-Prozesses möglich ist. Die Beispielanwendung wurde daher so gebündelt, dass das MIDlet vor der Initialisierung des UIs einen separaten Thread startet, welcher den Web Service initialisiert und den SOAP-Server startet.

6.4. Fazit

Das Transaktions-Framework wurde wie entworfen soweit implementiert, dass die Transaktionslogik gemäß den Spezifikationen mit kleinen Einschränkungen funktioniert, sofern die Anwendung nicht während einer Transaktion unerwartet beendet wird, Kommunikationsprobleme zum Verlust von Nachrichten führen oder Nachrichten aufgrund von Kommunikationsproblemen nicht gesendet werden können.

Da diese Ausnahmebedingungen im Falle langlebiger Transaktionen mit mobilen Teilnehmern aber unvermeidlich sind, wurden mit Timer-Handling und Persistenzschicht bereits Komponenten entworfen und in Grundzügen implementiert, die diese Ausnahmebedingungen bei Auftreten so behandeln, dass Transaktionen gemäß der Spezifikationen entweder weiterlaufen können oder aber so kontrolliert und definiert wie möglich beendet werden. In den übrigen Komponenten sind entsprechende Erweiterungen in Form von Timeout- und Wiederherstellungsmethoden vorgesehen.

Bei einer Weiterentwicklung des Frameworks sollten die Vervollständigung der Implementierung von Timer-Handling und Persistenzschicht sowie die Implementierung des Timeout-Handlings und der Wiederherstellung innerhalb der Transaktionslogik die höchste Priorität haben.

Das Freigabemanager-Beispiel wurde erfolgreich mit einem stationären Koordinator auf einem Standard-PC und einem mobilen Teilnehmer auf einem PDA getestet, wobei PC und PDA über ein WLAN miteinander verbunden waren.

7. Fazit

Zum Abschluss dieser Arbeit wird zunächst die Erreichung der Ziele aus 1.2 anhand der wichtigsten Ergebnisse der einzelnen Kapitel dokumentiert (7.1), bevor noch ein Ausblick auf mögliche Weiterentwicklungen und Anwendungen des Transaktions-Frameworks erfolgt (7.2).

7.1. Ergebnisse

In 1.2 wurden die folgenden Ziele festgelegt:

- Die Erarbeitung von Möglichkeiten des Transaktionsmanagements für mobile Web Services unter Einbeziehung existierender Spezifikationen, wobei langlebige Transaktionen im Vordergrund stehen sollten,
- die Analyse einer Auswahl dieser Spezifikationen (WS-Coordination und WS-BusinessActivity) mit dem Ziel der Erweiterung des Frameworks für mobile Web Services (Schrörs, 2006a) um die Fähigkeit, langlebige Transaktionen für Web Services auf kleinen mobilen Geräten zu unterstützen,
- der Entwurf dieser Erweiterung (Transaktions-Framework) anhand der Analyse, sowie
- die prototypische Umsetzung des Entwurfs und des Freigabemanager-Beispiels zwecks Überprüfung der Realisierbarkeit.

In den folgenden Abschnitten werden die Ergebnisse zusammengefasst.

7.1.1. Transaktionsmanagement für mobile Web Services

Die Verwendung von dienstorientierten Architekturen und Web Services ist eine weit verbreitete Möglichkeit zur Integration heterogener Systeme bzw. Anwendungen. Eine wichtige

Rolle spielt dabei das Workflow-Management zum Zweck der Geschäftsprozessautomatisierung. Da das Dienstkonzept dem Konzept der Komponenten ähnelt, sollte auch das Transaktionsmanagement für Web Services konzeptionell ähnlich zum Transaktionsmanagement nach X/Open DTP für verteilte Komponenten wie CORBA-Objekte oder EJBs sein.

Für Web-Service-Transaktionen existieren mit dem Business Transaction Protocol (BTP), Web Service Coordination (WS-Coordination) und dem Web Services Composite Application Framework (WS-CAF) drei Sätze von Spezifikationen für Koordination, atomare Transaktionen und langlebige Transaktionen. Vor dem Hintergrund der Verbreitung mobiler Technologien sollte in dieser Arbeit die Möglichkeit untersucht werden, auch Web Services auf mobilen Geräten in Transaktionen einzubeziehen.

Die drei Ansätze sind sich sehr ähnlich, WS-Coordination weist jedoch die geringste Komplexität auf und derjenige Teil der Spezifikation, der sich mit atomaren Transaktionen (WS-AtomicTransaction) befasst, wird bereits in kommerziellen und Open-Source-Produkten unterstützt, weswegen dieser Ansatz für diese Arbeit näher untersucht wurde.

Da Workflows lange andauernde Teilaktivitäten und Benutzerinteraktionen beinhalten können, erfordern sie zur Konsistenzsicherung Konzepte für langlebige Transaktionen mit Kompensation an Stelle atomarer, isolierter Transaktionen. WS-Coordination und WS-BusinessActivity beschreiben Architektur und Protokolle für die Koordination langlebiger Transaktionen mit Web Services. Eine zentrale Instanz, der Koordinator, übernimmt dabei die Aufgabe der Koordination beliebig vieler Teilnehmer. Die Protokollkommunikation findet dabei immer nur zwischen dem Koordinator und einem Teilnehmer statt, nicht zwischen Teilnehmern. WS-Coordination spezifiziert die Aktivierung (das Starten der Transaktion) sowie die Registrierung der Teilnehmer, WS-BusinessActivity spezifiziert Terminierungsprotokolle inklusive Abbruch-, Kompensations- und Fehlermechanismen.

WS-Coordination und WS-BusinessActivity sehen aufgrund der langen Vorgänge außerdem asynchrone Nachrichtenaustauschmuster vor. Asynchrone Kommunikation ist aber auch geeignet für die Kommunikation mit Web Services auf mobilen Geräten, da hier zusätzlich zu langen Bearbeitungszeiten (etwa im Falle von Benutzerinteraktion) Besonderheiten in der Kommunikation wie Offline-Zeiten verschiedener Länge berücksichtigt werden müssen.

7.1.2. Analyse von WS-Coordination und WS-BusinessActivity

Die Analyse der beiden Spezifikationen hinsichtlich der Umsetzbarkeit für mobile Systeme mit begrenzten Ressourcen und Offline-Situationen ergibt, dass sie für die Koordination mobiler Web Services geeignet sind. Es müssen jedoch hinsichtlich der Robustheit einer Implementierung einige Punkte besonders beachtet werden:

Asynchrone Kommunikation

Aufgrund der asynchronen Kommunikation müssen Mechanismen zur Korrelierung von Nachrichten verwendet werden. Die Spezifikationen sehen hierfür die Verwendung von WS-Addressing vor. Auch die Anwendungslogik transaktionaler Operationen wird asynchron aufgerufen, was zu komplexen Abläufen führen kann, im Rahmen derer Zustandsinformationen von Aufruf zu Aufruf zwischengespeichert werden müssen.

Timer und Timeouts

Das Terminierungsprotokoll WS-BusinessActivity hat kritische Phasen wenn auf asynchrone Quittungen gewartet wird. Bei Einsatz von mobilen Geräten unter Benutzung verschiedenartiger Netzwerktechnologien besteht jedoch das Risiko, dass Nachrichten verloren gehen. Dies muss durch den Einsatz von Timern kompensiert werden, so dass nicht zu lange gewartet wird und sich das Protokoll immer in einem definierten Zustand befindet. Dies gilt auch für die Anwendungslogik. Transaktionslogik und Anwendungslogik müssen also mit Timeouts umgehen können.

Wiederherstellung

Da mobile Geräte wie Mobiltelefone und Smartphones nur begrenzte Akku-Laufzeiten haben, muss weiterhin damit gerechnet werden, dass die Geräte ausgeschaltet werden während noch eine langlebige Transaktion läuft. Ist das Gerät zum Zeitpunkt des Ausschaltens nicht mit der Ausführung von Transaktions- oder Anwendungslogik beschäftigt, so muss es möglich sein, dass die transaktionale Anwendung nach dem Wiedereinschalten einfach weiter an der Transaktion teilnimmt, falls die Transaktion dann nicht zu weit fortgeschritten ist.

Persistenz

Wegen der geforderten Wiederherstellungsfähigkeit wird eine effiziente Persistenzschicht benötigt, die für die nichtflüchtige Speicherung von Daten sorgt. Dazu gehören Protokollzustände der aktuell laufenden Transaktionen, Anwendungsdaten und Informationen über alle aktuell laufenden Timer.

7.1.3. Entwurf des Transaktions-Frameworks

Der Entwurf des Transaktions-Frameworks als Erweiterung des Frameworks für mobile Web Services aus (Schrörs, 2006a) hat gezeigt, dass eine Umsetzung der Spezifikationen und der speziellen Robustheits-Anforderungen für mobile, auf CLDC 1.1 und MIDP 2.0 basierende J2ME-Laufzeitumgebungen realisierbar ist.

Die dafür notwendige Persistenzschicht muss dabei für eventuell stark begrenzte Ressourcen wie eingeschränkte Java-Fähigkeiten (keine Objektserialisierung in MIDP 2.0), begrenzten Speicherplatz und je nach Gerätetyp variierende Zugriffszeiten ausgelegt werden. An Stellen, an denen herkömmliche Entwurfsmuster Callback-Objekte vorsehen, die als Parameter an bestimmte Operationen (z. B. Timer starten) übergeben und persistent abgelegt werden müssen, muss auf diese Einschränkungen Rücksicht genommen werden.

Durch konsequenten Einsatz von verbreiteten für Frameworks geeigneten Entwurfsmustern fördert der Entwurf die leichte Wartbarkeit und Erweiterbarkeit des Transaktions-Frameworks. Die Verantwortlichkeiten der einzelnen Protokolle SOAP, WS-Coordination, WS-BusinessActivity sowie des von der Anwendung (z. B. Workflow-Management) benutzten Protokolls werden sauber getrennt auf die einzelnen Ebenen einer Vererbungshierarchie von Web-Service-Implementierungsklassen verteilt.

Der speziellen Problematik häufiger Offline-Situationen bei mobiler Kommunikation wird dadurch Rechnung getragen, dass ein adaptierbarer Ansatz für den Aufruf von Web Services konzipiert wurde, welcher dafür sorgt, dass transparent für die Logik, die den Aufruf tätigt, unterschiedliche, konfigurierbare Aufrufstrategien zum Einsatz kommen können. Aufrufstrategien können z. B. darin bestehen, den Aufruf asynchron auszuführen, den Aufruf im Fehlerfall mehrfach zu wiederholen, oder den Aufruf an ein Gateway zu delegieren, welches dann für eine bestimmte Zeit weiter versucht, den Aufruf an den Empfänger (z. B. ein gerade ausgeschaltetes Mobiltelefon) zu senden. Welche Strategien verwendet werden, ließe sich z. B. je nach Anwendungszweck oder Online/Offline-Situation konfigurieren.

7.1.4. Umsetzung und Konfigurationsempfehlungen

Das Transaktions-Framework wurde gemäß den Ergebnissen von Analyse und Entwurf erfolgreich zu großen Teilen umgesetzt. Das Freigabemanager-Beispiel konnte mit dem Framework sowohl im MIDP-Emulator als auch auf einem mobilen Gerät mit CDC und MIDP-Emulation zum Laufen gebracht werden.

Aus den verschiedenen Betrachtungen in Analyse und Entwurf sowie aus der darauf basierenden (teilweisen) Realisierung des Transaktions-Frameworks und des Freigabemanager-Beispiels resultieren die folgenden Konfigurationsempfehlungen.

Verteilte Anwendungen, die Transaktionen mit mobilen Web Services nutzen, sollten so entworfen werden, dass ein stationärer, ausfallsicherer, immer erreichbarer (d. h. besonders robuster) Koordinator verwendet wird, da dieser der zentrale Kommunikationspartner für alle Teilnehmer des WS-BusinessActivity-Protokolls ist. Fällt der Koordinator auch nur für kurze Zeit aus oder ist nicht erreichbar, ist die Konsistenz aller Teilnehmer gefährdet. Die Wiederherstellungslogik wäre daher besonders komplex. Dieses Problem ist auch im herkömmlichen Transaktionsmanagement bekannt.

Weiterhin müssen im Falle komplexer Workflows möglicherweise so viele Teilnehmer koordiniert werden, dass Systemressourcen wie Speicher und Rechenzeit in einem Umfang benötigt werden, der auf kleinen mobilen Geräten nicht zur Verfügung steht. Die Teilnehmer können dagegen immer mobil sein, da hier die Transaktionslogik nicht so viele Ressourcen benötigt.

Eine weitere Empfehlung besteht darin, dass für Teilnehmer, deren Offline-Wahrscheinlichkeit hoch ist, Gateways bereitgestellt werden sollten, die ausfallsicher und immer online sind. Über diese könnten solche Teilnehmer dann vom Koordinator und anderen Teilnehmern angesprochen werden. Nachrichten würden von einem Gateway solange gepuffert, bis sie ausgeliefert werden können oder bis ein Zeitlimit überschritten wird.

7.2. Ausblick

In diesem Abschnitt wird abschließend noch auf einige Weiterentwicklungsmöglichkeiten und Anwendungen für das Transaktions-Framework in unmittelbarer Zukunft hingewiesen. Dies ist auch als Anregung für Projekte und Abschlussarbeiten zu verstehen, die in demselben Themenfeld angesiedelt sind.

7.2.1. Fehlende wichtige Funktionalität

Eine hohe Priorität sollte zunächst die Vervollständigung des Transaktions-Frameworks in den Bereichen Persistenz, Timeout-Handling und Wiederherstellung haben. Die in 6.2 diesbezüglich genannte fehlende Funktionalität sollte implementiert und getestet werden. Dabei ist zu beachten, dass jeder dieser Bereiche sehr komplex ist.

Weiterhin wäre zu überlegen, wie die für Transaktionen sehr nützliche At-Least-Once-Auslieferungsstrategie für Web-Service-Aufrufe realisiert werden könnte. Mit Hilfe des Konzepts der Aufrufstrategien, dem Einsatz von Gateways und Mechanismen wie WS-Reliability (WS-Rel, 2004) und WS-ReliableMessaging (WS-RM, 2005) ließen sich hier vermutlich Verbesserungen erzielen.

Beim Testen und anschließenden Optimieren all dieser Features sollten verschieden lange Offline-Phasen der Teilnehmer ebenso simuliert werden wie Zeiten, in denen die mobilen Geräte abgeschaltet sind.

Ein Code-Generator als Implementierungshilfe für einfache (d. h. nicht transaktionale) und transaktionale Web Services würde schließlich die Benutzbarkeit sowohl des Basis-Frameworks als auch des Transaktions-Frameworks erhöhen.

7.2.2. Zielplattformen

Da sowohl das Basis-Framework als auch das Transaktions-Framework noch nicht auf reinen MIDP-Geräten ausgeführt und auf Funktionsfähigkeit und Leistungsfähigkeit (s. u.) getestet wurde, wäre ein entsprechender Test ein nahe liegendes Projekt.

Die Größe des Transaktions-Frameworks liegt z. B. mit bisher ca. 300 KByte über der Mindestempfehlung von 192 KByte für die Java-Laufzeitumgebung für CLDC 1.1. Hier wäre einerseits zu prüfen, ob dies ein Problem für bestimmte Geräte darstellt, andererseits könnte auch geprüft werden, inwieweit man diese Größe durch Platz sparenderen Code verringern könnte.

Ein weiterer interessanter Punkt wäre die Untersuchung der Leistungsfähigkeit des persistenten Speichers auf unterschiedlichen mobilen Geräten.

Der Speicherbedarf zur Laufzeit wurde ebenfalls noch nicht analysiert. Optimierungsansätze bestünden hier beim Web-Service-Scope (reentrante Implementierungen) sowie in der Minimierung der Anzahl von Objektinstantiierungen und des Bedarfs an persistentem Speicher.

Geräte wie Mobiltelefone und Smartphones, die zur Zeit der Erstellung dieser Arbeit noch mit CLDC/MIDP-Laufzeitumgebungen ausgeliefert wurden, werden vermutlich schon bald – so wie PDAs – CDC-fähig sein, während noch kleinere Geräte, z. B. aus dem Embedded-Bereich, vielleicht bald CLDC/MIDP-fähig sein werden.

Wenn sich entsprechende Anwendungen für diese Plattformen finden, könnte die Einsatzfähigkeit des Frameworks darauf überprüft werden. Dabei ist zu beachten, dass das Konzept eines mobilen Koordinators, welches für Workflow-Anwendungen nicht sinnvoll ist, in anderen Szenarien durchaus sinnvoll sein könnte, etwa im Falle spontaner, autonomer Transaktionen mit einigen wenigen Teilnehmern zum Zwecke der Konsistenzerhaltung von Daten in Sensornetzwerken o. Ä..

7.2.3. Portierungen und Erweiterungen

Auf anderen mobilen Plattformen, wie z. B. dem Microsoft .NET Compact Framework (s. (MS-NET-CF, 2006)), existiert noch keine verbreitete, hinlänglich bekannte Möglichkeit, Web Services zu betreiben. Es böte sich daher der Versuch einer Portierung des Basis-Frameworks und des Transaktions-Frameworks an.

OASIS arbeitet zum Zeitpunkt der Abgabe dieser Arbeit an neuen Versionen der WS-Coordination-Spezifikationen (s. Anhang A.3). Es wäre zu überprüfen, inwieweit sich diese von den bisherigen Versionen unterscheiden, und wie eine parallele Unterstützung aktueller und zukünftiger Versionen der Protokolle oder auch eine Migration des Transaktions-Frameworks und potentiell bereits dafür entwickelter Anwendungen auf zukünftige Versionen der Spezifikationen durchgeführt werden könnte. Das in 3.4.4 erwähnte JAXTX könnte hier dabei behilflich sein, verschiedene Protokollversionen und sogar verschiedene Ansätze (z. B. auch WS-CAF) zu kapseln.

Basis-Framework und Transaktions-Framework sollten weiterhin um Mechanismen wie Service-Repository und Service-Discovery (z. B. UDDI, s. (UDDI, 2006)) erweitert werden, damit die miteinander kommunizierenden Web Services sich gegenseitig addressieren können, ohne dass die genauen Endpunkt-Adressen zum Entwicklungs- oder Installationszeitpunkt bekannt sein müssen.

Abschließend soll auch die Sicherheit (im Sinne von Security) noch Erwähnung finden. Im Basis-Framework noch nicht unterstützt, empfehlen WS-Coordination und WS-BusinessActivity die Verwendung von WS-Security (WS-Sec, 2006) und verwandten Spezifikationen, siehe dazu (WSS, 2006). Security ist unabhängig von Transaktionen und sollte daher auf Ebene des Basis-Frameworks implementiert werden.

A. Ergänzungen

A.1. Zu Grundlagen: WS-CAF Details

Zum Vergleich mit WS-Coordination (s. 3.4.2) werden hier die drei Schichten des Web Service Composite Application Frameworks (WS-CAF, s. 3.4.3) kurz beschrieben.

A.1.1. 1. Ebene: Kontextverwaltung für Aktivitäten, WS-CTX

WS-CTX (Context, (WS-CTX, 2003)) ist eine generische Spezifikation über die Verwaltung und Propagierung von Kontexten für gemeinsame Aktivitäten mehrerer Web Services. Im Gegensatz zu WS-Coordination kann der Kontext die beteiligten Web Services als Endpunkt-Referenzen (URIs) enthalten und es sind geschachtelte Kontexte syntaktisch möglich, ohne dass eine Semantik hierfür spezifiziert wird.

WS-CTX sieht einen speziellen zentralen Dienst zur Verwaltung von Kontexten vor, der wie der Activation Service von WS-Coordination zum Erzeugen eines Kontextes (was dem Starten einer Aktivität entspricht) benutzt werden kann, darüber hinaus aber die Kontexte auch für die Dauer der Aktivitäten in einem Repository vorhält.

Der Kontext kann „by value“ als vollständiges Objekt (im Falle von SOAP also als XML-Struktur) als auch „by reference“ übergeben werden. Eine Kontextreferenz verweist dabei auf den Verwaltungsdienst und enthält außerdem die Kontext-ID.

Weiterhin spezifiziert WS-CTX ein Interface für „Activity Lifecycle Services“ (ALS), welche über den Lebenszyklus einer Aktivität per Aufruf (z. B. `begin` oder `complete`) mit entsprechenden asynchronen Antwortnachrichten (`begun`, `completed`) informiert werden. ALS müssen beim Kontextverwaltungsdienst an- und abgemeldet werden.

A.1.2. 2. Ebene: Koordination, WS-CF

WS-CF (Coordination Framework, (WS-CF, 2003)) baut auf WS-CTX auf, indem eine Anwendung von WS-CTX spezifiziert wird. Es werden wie bei WS-Coordination die Rollen Koordinator und Teilnehmer definiert. Darüber hinaus werden explizit Koordinatoren spezifiziert, die selbst Teilnehmer einer anderen Koordination sind (Schachtelung oder auch Interposition), was mit WS-Coordination zwar auch möglich wäre, aber dort nicht explizit erwähnt wird.

Koordination wird in WS-CF allgemein als Hilfsmittel zur Lösung verteilter Aufgaben wie z. B. Transaktionsmanagement, Sicherheitsbelange, Caching oder Replikation angesehen, bei denen es darum geht, dass eine zentrale Instanz (Koordinator) Information und Kontrolle über die beteiligten Komponenten (in diesem Fall Web Services) hat.

Der Koordinations-Dienst ist dabei ein ALS in WS-CTX. Der Kontext wird um Felder für eine Referenz auf den Koordinator, den Koordinationstyp und mögliche Koordinations-Protokolle erweitert. Es wird dabei keine Semantik für Koordinations-Protokolle festgelegt, sondern lediglich für Koordinationsadminstration, d. h. Registrierung und Abmeldung von Teilnehmern, Statusabfragen und allgemeine Fehlerbehandlung.

A.1.3. 3. Ebene: Transaktionen, WS-TXM

WS-TXM (Transaction Management, (WS-TXM, 2003)) baut auf WS-CF auf und spezifiziert Web-Service-Transaktionen als eine Art der Koordination. Aufgrund unterschiedlicher Komplexität von Anwendungen, unterschiedlicher, möglicherweise langer, Ausführungszeit aufgrund von komplexen Berechnungen oder nötiger Benutzerinteraktion sowie unterschiedlicher Kopplung (im Falle von Web Services idealerweise sehr lose Kopplung) werden unterschiedliche Mechanismen für Wiederherstellung und Kompensation benötigt. Es werden Transaktionen daher in drei Ausprägungen spezifiziert: Atomare Transaktionen, langlebige Aktionen, sowie Geschäftsprozesse.

A.1.3.1. Atomare Transaktionen

Atomare Transaktionen („ACID Transactions“) gehorchen dem ACID-Prinzip und werden häufig mittels des 2-Phase-Commit-Protokolls gesteuert (terminiert), welches in WS-TXM ähnlich wie in WS-AtomicTransaction (s. o.) spezifiziert wird. WS-TXM sieht darüber hinaus noch die Möglichkeit vor, „Synchronization“-Listener in Form weiterer Web Services beim Koordinator zu registrieren, welche über Nachrichten wie `beforeCompletion` und `afterCompletion` über den Status der Transaktion informiert werden. Dies ist in auf X/Open DTP basierenden Transaktionsmanagementumgebungen (s. 3.2) ebenfall üblich.

A.1.3.2. Langlebige Aktionen

Langlebige Aktionen („Long Running Actions“, LRAs) entsprechen den von WS-BusinessActivity spezifizierten langlebigen Transaktionen. Beteiligte Web Services müssen die Fähigkeit zur Kompensation bereits abgeschlossener Vorgänge haben, um an einer LRA teilzunehmen. Wie dies im einzelnen geschieht, ist wie bei WS-BA dem Implementierer des Dienstes überlassen.

WS-TXM unterscheidet für LRAs applikationsunabhängige und applikationsabhängige Kompensationsfähigkeit. Applikationsunabhängige Komepensationsfähigkeit kann von einem Teilnehmer über den Kontext verlangt werden. Der Teilnehmer muss dann einen Kompensationsdienst mit zugehörigen Parametern beim Koordinator registrieren, der im Falle eines nicht-erfolgreichen Durchlaufs aufgerufen wird wie in Abb. A.1 gezeigt. Die Aufrufreihenfolge für die Kompensationsdienste ist umgekehrt zur Registrierungsreihenfolge. Dieses Prinzip ist auch aus älteren Ansätzen wie Sagas (s. 3.3) bekannt. Wird die Registrierung eines Kompensationsdienstes nicht verlangt und führt ein Teilnehmer diese auch nicht durch, so kann die Kompensation einer übergeordneten LRA dazu führen, dass gewisse Vorgänge oder ganze Unter-LRAs nicht kompensiert werden. Diese benötigen dann applikationsspezifische Kompensation, die in der Anwendungslogik zu implementieren ist.

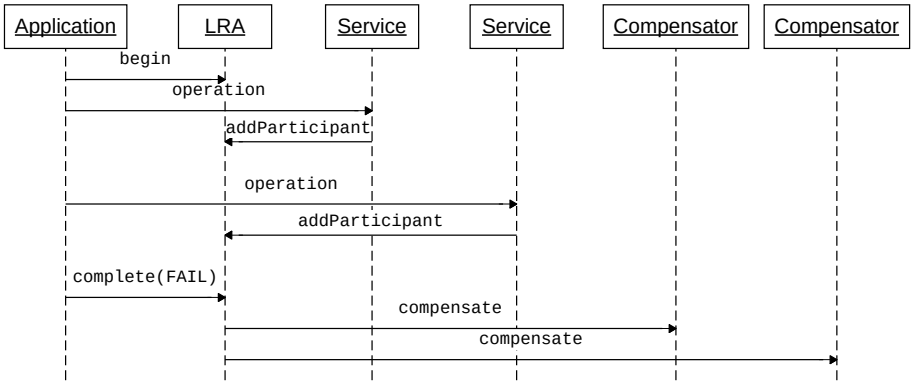

Abbildung A.1.: WS-TXM LRA mit Kompensation (WS-TXM, 2003, S. 30)

Es ist anzumerken, dass das Registrieren des Kompensationsdienstes eine Herausforderung an die Sicherheitsarchitektur des betroffenen Systems darstellt. Es muss sichergestellt werden, dass die Kompensation auch wirklich den eigentlichen Vorgang rückgängig macht und nicht andere Aktionen durchführt. Außerdem muss die Authentizität des registrierenden Teilnehmers überprüft werden.

A.1.3.3. Geschäftsprozesse

Geschäftsprozesse („Business Processes“, BPs) gehen in Komplexität und Semantik über LRAs (und auch über die langlebigen Transaktionen aus WS-BusinessActivity) hinaus. WS-TXM spezifiziert hier das Management von komplexen Workflows zwischen unterschiedlichen „Business Domains“ inklusive Mechanismen wie „Checkpointing“ und „Replay“. Dies geht über reines Transaktionsmanagement hinaus und eine Einbeziehung dieser Konzepte würde den Rahmen dieser Arbeit sprengen. Die Details sind in (WS-TXM, 2003) zu finden.

A.2. Zu Analyse: Semantik WS-C & WS-BA

Die exakte Analyse der Anwendungsfälle für das Framework in 4.1.3 führt zu der in den folgenden Abschnitten beschriebene Semantik der Protokollnachrichten (bzw. Operationen) von WS-Coordination und WS-BusinessActivity.

Transaktionale Operation (Koordinator)

Wird eine Operation des Koordinators aufgerufen, die zu ihrer Ausfürung eine Transaktion benötigt, so speichert die Framework-Logik den Aufruf zwischen und ruft zunächst die Operation `CreateCoordinationContext` des Aktivierungs-Services auf. Die Anwendungslogik für die Operation wird zunächst nicht aufgerufen. Die Aufgabe des Koordinators legt dabei den Koordinationstyp fest. Dieser entscheidet, ob alle Teilnehmer erfolgreich abschließen müssen, oder ob es auch eine Teilmenge genügt, um die Transaktion erfolgreich zu beenden. Der Koordinationstyp wird dem Aktivierungs-Service als Parameter übergeben.

CreateCoordinationContext

Die Operation `CreateCoordinationContext` des Aktivierungs-Services wird vom Koordinator aufgerufen. Der Aktivierungs-Service erzeugt den Transaktionskontext, in dem die Adresse des Registrierungs-Services enthalten ist und schickt den Kontext mittels der Operation `CreateCoordinationContextResponse` zurück an den Koordinator. Hier ist keine Anwendungslogik beteiligt.

CreateCoordinationContextResponse

Die Operation `CreateCoordinationContextResponse` des Koordinators wird vom Aktivierungs-Service aufgerufen.Die Framework-Logik des Koordinators speichert den Transaktionskontext und führt nun die Anwendungslogik der ursprünglich aufgerufenen Operation aus. Ruft diese Operation nun weitere Services auf, die an der Transaktion teilnehmen sollen, so wird der Transaktionskontext vom Framework an den Aufruf angehängt.

Anwendungsfälle ACC und MCC: Die Anwendungslogik des Koordinators muss für Teilnehmer, die für den Protokolltyp CoordinatorCompletion registriert sind, entscheiden, wann diese Teilnehmer mittels der Operation `Complete` dazu aufgefordert werden sollen, die Bearbeitung im Rahmen der aktuellen Transaktion zu beenden.

Erste Operation eines Teilnehmers

Wird eine Operation eines Teilnehmers innerhalb einer Transaktion (d. h. mit Transaktionskontext) vom Koordinator oder einem bereits registrierten anderen Teilnehmer aufgerufen, und wurde vorher noch keine weitere Operation des Teilnehmers innerhalb dieser Transaktion aufgerufen, so sorgt die Framework-Logik der Operation dafür, dass der Aufruf zwischengespeichert wird und `Register` des Registrierungs-Services aufgerufen wird. Die Aufgabe des Teilnehmers legt dabei zusammen mit dem Koordinator bzw. dessen Aufgabe den Protokolltyp fest. Dieser entscheidet, ob der Teilnehmer eine Aufforderung zum Beenden der Transaktion braucht oder nicht. Der Protokolltyp wird dem Registrierungs-Service als Parameter übergeben. Der Teilnehmer ist nun im Zustand `REGISTERING`. In diesem Zustand bewirkt Framework-Logik für alle weiteren aufgerufenen Operationen ebenfalls eine Zwischenspeicherung, solange bis die Registrierung bestätigt wurde. Die Anwendungslogik der aufgerufenen Operationen wird zunächst nicht ausgeführt.

Register

Die Operation `Register` des Registrierungs-Services wird vom Teilnehmer aufgerufen. Der Registrierungs-Service registriert den Teilnehmer für die im Transaktionskontext des Aufrufs referenzierte Transaktion mit dem als Parameter angegebenen Protokolltyp. Der Koordinator kann beim Registrierungs-Service jederzeit auf die Daten (Adresse, Protokolltyp, Zustand) aller registrierten Teilnehmer zugreifen.

RegisterResponse

Die Operation `RegisterResponse` des Teilnehmers wird vom Registrierungs-Service aufgerufen. Die Framework-Logik des Teilnehmers ändert den Zustand auf `REGISTERED` bzw. `ACTIVE` (WS-BusinessActivity) und ruft nun die Anwendungslogik aller Operationen auf, die im Zustand `REGISTERING` aufgerufen wurden.

Weitere Operationen des Teilnehmers

Werden im Laufe der Transaktion nach der Registrierung weitere Operationen des Teilnehmers aufgerufen, so wird im Zustand `REGISTERED` bzw. `ACTIVE` (WS-BusinessActivity) von der Framework-Logik zu jeder aufgerufenen Operation sofort die Anwendungslogik der Operation aufgerufen.

Anwendungsfälle APP und MPP: Die Anwendungslogik aller im Laufe der Transaktion aufgerufenen Operationen (einschließlich der im Zustand `REGISTERING` aufgerufenen Operationen) muss im Falle des Protokolltyps ParticipantCompletion den Koordinator genau einmal mittels Aufruf der Operation `Completed` über die erfolgreich beendete Ausführung aller im Rahmen der aktuellen Transaktion aufgerufenen Operationen informieren. Das Framework sollte hierfür eine Funktion bereitstellen, die nach dem Aufruf den Zustand aus Sicht des Teilnehmers auf `COMPLETED` setzt.

Callback-Operation (Koordinator)

Nachdem der Koordinator oder ein anderer Teilnehmer eine Operation des Teilnehmers aufgerufen hat und der Teilnehmer sich registriert hat, ruft die Anwendungslogik des Teilnehmers nach erfolgter Verarbeitung eine Callback-Methode des Koordinators auf. Die Framework-Logik des Koordinators ruft sofort die Anwendungslogik auf, die das vom Teilnehmer zurück gelieferte Ergebnis verarbeitet.

Anwendungsfälle ACC und MCC: Die Anwendungslogik des Koordinators muss für einen Teilnehmer, der für den Protokolltyp CoordinatorCompletion registriert ist, entscheiden, wann dieser Teilnehmer mittels der Operation `Complete` dazu aufgefordert werden soll, die Bearbeitung im Rahmen der aktuellen Transaktion zu beenden. Das Framework sollte hierfür eine Funktion bereitstellen, die nach dem Aufruf den Zustand aus Sicht des Koordinators auf `COMPLETING` setzt.

Teilnehmer.Complete

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `COMPLETING` und ruft dann eine spezielle Listener-Methode der Anwendungslogik auf, die vom Entwickler der Anwendungslogik so implementiert werden muss, dass jegliche Verarbeitung, die innerhalb der Transaktion noch läuft, erfolgreich beendet wird. Erst dann darf aus der Listener-Methode zurückgesprungen werden, woraufhin die Framework-Logik den Zustand auf `COMPLETED` setzt und die Operation `Completed` des Koordinators aufruft.

Anwendungsfälle ACP und MCP: Dieser Aufruf ist nur für Teilnehmer erlaubt, die für das CoordinatorCompletion-Protokoll registriert sind. Die Anwendungslogik braucht im Gegensatz zu den Anwendungsfällen APP und MPP nicht selber die `Completed`-Nachricht schicken.

Koordinator.Completed

Die Framework-Logik setzt den Zustand aus Koordinatorsicht auf `COMPLETED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf. Die Verantwortlichkeiten unterscheiden sich je nach Koordinationstyp.

Anwendungsfälle APC und ACC: Ist der Koordinationstyp AtomicOutcome, so kann die Framework-Logik prüfen, ob alle registrierten Teilnehmer im Zustand `COMPLETED` sind. Ist dies der Fall, so kann an alle die Nachricht `Close` geschickt werden, um die Transaktion erfolgreich abzuschließen, der Zustand ändert sich für jeden Teilnehmer dadurch (aus Koordinatorsicht) auf `CLOSING`. Die Anwendungslogik (Listener-Methode) hat keine weiteren Verantwortlichkeiten.

Anwendungsfälle MPC und MCC: Ist der Koordinationstyp dagegen MixedOutcome, so kann die Framework-Logik nicht entscheiden, wann bzw. ob die Transaktion erfolgreich abgeschlossen werden soll, da für einen erfolgreichen Gesamtabschluss nicht alle Teilnehmer erfolgreich abschließen müssen. Die Listener-Methode der Anwendungslogik muss dies prüfen und ggf. allen Teilnehmern entsprechende Anweisungen schicken (`Close`, `Cancel` oder `Compensate`). Das Framework sollte hierfür Funktionen bereitstellen, die bei Aufruf den Zustand des jeweiligen Teilnehmers aus Koordinatorsicht auf `CLOSING`, `CANCELING_ACTIVE`, `CANCELING_COMPLETING` oder `COMPENSATING` setzen.

Teilnehmer.Close

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `CLOSING` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die vom Entwickler der Anwendungslogik so implementiert werden muss, dass alle notwendigen Schritte zum erfolgreichen Abschließen der Transaktion durchgeführt werden, bevor aus der Methode zurückgesprungen wird. Nach dem Rücksprung setzt die Framework-Logik den Zustand auf `ENDED` und ruft die Operation `Closed` des Koordinators auf.

Von der Anwendungslogik dürfen dabei keine weiteren (d. h. noch nicht registrierten) Teilnehmer aufgerufen werden, da davon ausgegangen werden muss, dass alle Teilnehmer die Transaktion gerade abschließen müssen. Es bestünde sonst die Gefahr, dass ein neuer Teilnehmer nicht mehr von der Koordinator-Logik erfasst werden würde, die diesen Abschluss gerade durchführt.

Koordinator.Closed

Die Framework-Logik setzt den Zustand aus Koordinatorsicht auf `ENDED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch speziell für den aufrufenden Teilnehmer Aufräumarbeiten ausführen kann.

Teilnehmer.Cancel

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `CANCELING_ACTIVE` oder `CANCELING_COMPLETING` (je nach aktuellem Zustand) und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die vom Entwickler der Anwendungslogik so implementiert werden muss, dass alle notwendigen Schritte zum Abbrechen der Transaktion durchgeführt werden, bevor aus der Methode zurückgesprungen wird. Nach dem Rücksprung setzt die Framework-Logik den Zustand auf `ENDED` und ruft die Operation `Canceled` des Koordinators auf.

Von der Anwendungslogik dürfen dabei keine weiteren (d. h. noch nicht registrierten) Teilnehmer aufgerufen werden, da davon ausgegangen werden muss, dass alle Teilnehmer die Transaktion gerade abschließen müssen. Es bestünde sonst die Gefahr, dass ein neuer Teilnehmer nicht mehr von der Koordinator-Logik erfasst werden würde, die diesen Abschluss gerade durchführt.

Koordinator.Canceled

Die Framework-Logik setzt den Zustand aus Koordinatorsicht auf `ENDED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch speziell für den aufrufenden Teilnehmer Aufräumarbeiten ausführen kann.

Teilnehmer.Compensate

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `COMPENSATING` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die vom Entwickler der Anwendungslogik so implementiert werden muss, dass alle notwendigen Schritte zum Kompensieren der gesamten im Laufe der Transaktion durchgeführten Bearbeitung durchgeführt werden, bevor aus der Methode zurückgesprungen wird. Nach dem Rücksprung setzt die Framework-Logik den Zustand auf `ENDED` und ruft die Operation `Compensated` des Koordinators auf. Zu Kompensation siehe auch 4.1.7.

Von der Anwendungslogik dürfen dabei keine weiteren (d. h. noch nicht registrierten) Teilnehmer aufgerufen werden, da davon ausgegangen werden muss, dass alle Teilnehmer die Transaktion gerade abschließen müssen. Es bestünde sonst die Gefahr, dass ein neuer Teilnehmer nicht mehr von der Koordinator-Logik erfasst werden würde, die diesen Abschluss gerade durchführt.

Koordinator.Compensated

Die Framework-Logik setzt den Zustand aus Koordinatorsicht auf `ENDED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch speziell für den aufrufenden Teilnehmer Aufräumarbeiten ausführen kann.

Koordinator.Exit

Der kontrollierte Ausstieg eines Teilnehmers aus der Transaktion wurde von der Anwendungslogik des Teilnehmers über eine Funktion des Frameworks initiiert, die den Zustand aus Teilnehmersicht auf `EXITING` setzt und die Operation `Exit` des Koordinators aufruft.

Die Framework-Logik des Koordinators setzt nach dem Empfang der `Exit`-Nachricht auch den Zustand aus Koordinatorsicht auf `EXITING` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch speziell für den aufrufenden Teilnehmer Aufräumarbeiten ausführen kann. Nach Rücksprung dieser Methode wird der Zustand auf `ENDED` gesetzt und die Operation `Exited` des Teilnehmers aufgerufen.

Teilnehmer.Exited

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `ENDED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch Aufräumarbeiten ausführen kann.

Koordinator.Fault

Die Fehlermeldung eines Teilnehmers wurde von der Anwendungslogik des Teilnehmers über eine Funktion des Frameworks initiiert, die den Zustand aus Teilnehmersicht auf `FAULTING_ACTIVE` oder `FAULTING_COMPENSATING` (je nach aktuellem Zustand) setzt und die Operation `Fault` des Koordinators mit einer von der Anwendungslogik übergebenen Fehlermeldung aufruft.

Die Framework-Logik des Koordinators setzt nach dem Empfang der `Fault`-Nachricht auch den Zustand aus Koordinatorsicht auf `FAULTING_ACTIVE` oder `FAULTING_COMPENSATING` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch speziell für den aufrufenden Teilnehmer Aufräumarbeiten ausführen kann. Nach Rücksprung dieser Methode wird der Zustand auf `ENDED` gesetzt und die Operation `Faulted` des Teilnehmers aufgerufen.

Anwendungsfälle APC und ACC: Ist der Koordinationstyp AtomicOutcome, so kann die Framework-Logik die Transaktion abbrechen, da nicht mehr alle Teilnehmer erfolgreich abschließen können. Allen Teilnehmern, die sich im Zustand `ACTIVE` befinden, wird daher `Cancel` geschickt, allen die sich im Zustand `CLOSED` befinden, wird dagegen `Compensate` geschickt, damit diese die Kompensation ihrer bereits abgeschlossenen Verarbeitung durchführen können. Die Zustände der Teilnehmer aus Koordinatorsicht werden entsprechend angepasst.

Anwendungsfälle MPC und MCC: Ist der Koordinationstyp dagegen MixedOutcome, so kann die Framework-Logik nicht entscheiden, ob die Transaktion nach der Fehlermeldung noch erfolgreich abgeschlossen werden kann, da für einen erfolgreichen Gesamtabschluss nicht alle Teilnehmer erfolgreich abschließen müssen. Die Listener-Methode der Anwendungslogik muss dies prüfen und ggf. allen Teilnehmern entsprechende Anweisungen schicken (`Close`, `Cancel` oder `Compensate`). Das Framework sollte hierfür Funktionen bereitstellen, die bei Aufruf den Zustand des jeweiligen Teilnehmers aus Koordinatorsicht auf `CLOSING`, `CANCELING_ACTIVE`, `CANCELING_COMPLETING` oder `COMPENSATING` setzen.

Teilnehmer.Faulted

Die Framework-Logik setzt den Zustand aus Teilnehmersicht auf `ENDED` und ruft eine spezielle Listener-Methode der Anwendungslogik auf, die keine besonderen Verantwortlichkeiten mehr hat, jedoch Aufräumarbeiten ausführen kann.

GetStatus

Dieser Aufruf kann vom Koordinator an den Teilnehmer geschickt werden oder umgekehrt. Er kann von Anwendungs- und Framework-Logik auf beiden Seiten genutzt werden, um gezielt den Zustand des Protokolls aus Sicht des Empfängers anzufragen (z. B. zu Zwecken der Wiederherstellung, siehe 4.2.1.4). Die Framework-Logik kann eine Listener-Methode der Anwendungslogik aufrufen, die jedoch keine besonderen Verantwortlichkeiten hätte. Nach Rücksprung dieser Methode wird vom Framework die Operation `Status` des Absenders mit dem aktuellen Zustand als Parameter aufgerufen.

Status

Die Framework-Logik ruft direkt ein spezielle Listener-Methode der Anwendungslogik auf, die damit über den Zustand des Protokolls aus Sicht des Absenders informiert wird.

A.3. Neue Versionen der Spezifikationen

WS-Coordination wird seit November 2005 von OASIS weiterentwickelt (WS-TX, 2006). Im Zeitraum der Erstellung dieser Arbeit erschienen dort Committee Drafts für neue Versionen der Spezifikationen WS-Coordination (WS-C, 2006), WS-AtomicTransaction (WS-AT,

2006) und WS-BusinessActivity (WS-BA, 2006) bei OASIS. Die Versionsnummer lautet jeweils 1.1. Die neuen Versionen beinhalten gegenüber den in dieser Arbeit benutzten 1.0-Versionen neben einigen semantischen Änderungen neue XML-Namespaces für die SOAP-Nachrichten.

Die im Rahmen dieser Arbeit erstellten Implementierungen des Frameworks sowie der Koordinator- und Teilnehmer-Web-Services des Beispielszenarios basieren noch auf WS-Coordination 1.0 und WS-BusinessActivity 1.0, da zum Einen das für den Koordinator verwendete Axis-basierte Framework (Gerlach, 2005) diese bereits implementierte und zum Anderen bis zur Abgabe der Arbeit noch keine endgültigen Dokumente für die neuen Versionen verfügbar waren.

Aufgrund der neuen Namespaces sind die SOAP-Nachrichten der 1.0- und 1.1-Versionen der Spezifikationen nicht kompatibel. Um die 1.1-Versionen zu unterstützen, müssten für das Transaktions-Framework daher neue Datentypen und Operationen vorgesehen werden, auch wenn diese sich in Struktur und Funktionalität nicht von denen der 1.0-Versionen unterschieden. Es bleibt abzuwarten, wie weit sich die neuen Versionen schließlich von den alten unterscheiden werden, um den für eine Erweiterung bzw. eine Migration des Frameworks benötigten Aufwand zu beurteilen.

Auch WS-CAF wird bei OASIS gepflegt (WS-CAF, 2006). Es gibt hier noch keine neuen Versionen gegenüber den in 3.4.3 und Anhang A.1 vorgestellten Spezifikationen. Auf der Webseite können unter Documents alle bisherigen Versionen der Spezifikationen gefunden werden.

Mit der Version 2.0 von WSDL (WSDL-2, 2006) sind aufgrund der neuen Features wie Interface-Vererbung weitere Veränderungen der Transaktions-Spezifikationen zu erwarten, da diese die Web-Service-Beschreibung direkt betreffen.

Für das Mobile Information Device Profile (MIDP) wurde mit der Version 2.1 ein Maintenance Release veröffentlicht (JSR-118b, 2006).

B. Inhalt der beiliegenden CD

Die beiliegende CD hat folgenden Inhalt:

- `/!contents.txt`: Inhalt der CD
- `/masterthesis.pdf`: Dieses Dokument
- `/Development`: Sourcecode
  - `workspace_j2me.zip`: Eclipse-Projekte für den mobilen Teil der Implementierung inkl. des Transaktions-Frameworks im Source-Folder `coordination` und des erweiterten Web-Servce-Frameworks aus (Schrörs, 2006a)
  - `workspace_axis.zip`: Eclipse-Projekte für den stationären Teil der Implementierung, Sourcecode zu (Gerlach, 2006) mit kleinen Änderungen
  - `testclients_schoers.zip`: J2SE-Testclients für (Schrörs, 2006a) für die verschiedenen Kommunikationsstile für Web-Services
- `/Documentation`: Installationsanleitungen und JavaDoc
  - `install_mobile.txt`: Installationsanweisungen für Entwicklungs- und Testsystem des mobilen Teils der Implementierung
  - `install_stationary.txt`: Installationsanweisungen für Entwicklungs- und Testsystem des stationären Teils der Implementierung
  - `javadoc_all.zip`: JavaDoc für das mobile Web-Service-Framework, das Transaktions-Framework und die mobile Beispiel-Anwendung (GUI-MIDlet und Web Service)
  - `javadoc_server.zip`: JavaDoc nur für das mobile Web-Service-Framework und das Transaktions-Framework
- `/Required_Software`: Installationspakete für die benötigte, frei verfügbare Software, beschrieben in den Installationsanleitungen
- `/Resources`: Frei verfügbare oder für wissenschaftliche Zwecke veröffentlichbare Quellen aus dem Literaturverzeichnis, jeweils in einem eigenen Unterverzeichnis, das so benannt ist wie das Kürzel der zugehörigen Quelle

Glossar

2-Phase-Commit-Protokoll Übliches Terminierungsprotokoll für atomare Transaktionen mit zwei Phasen, bei dem in der ersten Phase alle Teilnehmer vom Koordinator bzw. Transaktionsmanager dazu aufgefordert werden, das Ergebnis ihrer Verarbeitung (Erfolg oder Misserfolg) zu melden, bevor sie aufgrund des Gesamtergebnisses in der zweiten Phase zum Festschreiben (Commit) oder Verwerfen (Rollback) aller Änderungen aufgefordert werden. S. z. B. (Gray und Reuter, 1993), (Tanenbaum und van Steen, 2002, Kap. 7)

2PC(-Protokoll) s. 2-Phase-Commit-Protokoll.

ACID Engl. Abkürzung für die Eigenschaften einer Menge von Operationen im Rahmen einer atomaren Transaktion: Atomicity, Consistency, Isolation, Durablilty (s. z. B. (Gray und Reuter, 1993)).

Aktivierung Start einer Transaktion/Koordination, schließt das Erzeugen des Transaktions-/Koordinationskontexts ein.

Dienst Entdeckbare, beschreibbare, entkoppelte Komponente mit wohldefiniertem Interface zur Erfüllung einer genau definierten Aufgabe der Geschäftslogik.

Dienstorientierte Architektur IT-Systemarchitektur, in der Kernfunktionalitäten als Dienste implementiert werden und Workflows mittels Dienstaufrufen realisiert werden. Zentrale Bestandteile sind die Dienste, Service-Repository (zum Auffinden von Diensten), Service-Bus (zur einheitlichen Kommunikation) und die Benutzerschnittstellen.

EAI Enterprise Application Integration, Integration heterogener IT-Systeme innerhalb einer Anwendung.

Kompensation Das Rückgängigmachen einer bereits abgeschlossenen Operation, z. B. Storno einer Buchung.

Koordination Allgemeiner, zusammenfassender Ausdruck für den Vorgang der Registrierung von Transaktionsteilnehmern bei einer zentralen Instanz (Koordinator) und das steuernde Ausführen von Protokollen für alle Teilnehmer durch diese zentrale Instanz. Langlebige Transaktionen sind Koordinationen.

Koordinationskontext In langlebigen Transaktionen und allgemein in Koordinationen eine dem Transaktionskontext entsprechende Struktur.

Koordinator In langlebigen Transaktionen der gebräuchliche Ausdruck für den Transaktionsmanager.

Metamuster Beschreibung einer allgemeinen Implementierungstechnik für eine Klasse von für die Framework-Entwicklung geeigneten Entwurfsmustern nach (Jacobsen u. a., 1997).

Prozessautomatisierung Ansatz, Geschäftsprozesse durch den Einsatz von IT-Systemen sowie von Reengineering und Optimierung so weit wie möglich zu automatisieren und damit effizienter zu gestalten.

Registrierung Anmeldung eines (neuen) Teilnehmers für die Transaktion/Koordination beim Koordinator, nachdem der Teilnehmer vom Koordinator oder einem bereits registrierten Teilnehmer aufgerufen wurde. Nach der Antwort des Koordinators ist der Teilnehmer ebenfalls registriert.

Ressourcenmanager Eine Serverkomponente, z. B. eine Datenbank, die an atomaren Transaktionen teilnimmt.

Service Siehe Dienst.

SOA Service Oriented Architecture, siehe Dienstorientierte Architektur.

Teilnehmer In langlebigen Transaktionen der gebräuchliche Ausdruck für einen Ressourcenmanager.

Terminierung Beenden einer Transaktion/Koordination durch das Ausführen eines Terminierungsprotokolls, wobei das Protokoll immer zwischen Koordinator und Teilnehmer, nicht jedoch zwischen zwei Teilnehmern der Transaktion/Koordination abläuft. Eine Transaktion/Koordination ist beendet, wenn das Terminierungsprotokoll zwischen Koordinator und jedem Teilnehmer bis zum Endzustand abgelaufen ist.

Transaktion, atomare Abfolge von Operationen, die nur komplett oder gar nicht durchgeführt wird.

Transaktion, langlebige Abfolge von Operationen, die nicht unbedingt komplett durchgeführt wird, und die nicht die ACID-Eigenschaften hat, wobei durch Mechanismen wie Koordination und Kompensation aber trotzdem eine weitgehende Konsistenzsicherung erreicht werden soll.

Transaktionskontext Datenstruktur, die mit einem transaktionalen Aufruf an den Empfänger übergeben wird. Enthält eine eindeutige Identifikation für die Transaktion, innerhalb derer der Aufruf stattfindet.

Transaktionsmanager Zentrale, koordinierende Instanz in einer Transaktion.

Web Service Implementierung eines Dienstes, welcher mittels Web Service Description Language nach (WSDL-1.1, 2001) bzw. (WSDL-2, 2006) beschrieben, mittels Universal Description, Discovery and Integration (UDDI, 2006) aufgefunden und mittels Simple Object Access Protocol (SOAP, 2003) aufgerufen wird.

Workflow IT-seitiger Vorgang zur Unterstützung von Geschäftsprozessen.

X/Open DTP Referenzmodell des X/Open-Konsortiums zu verteiltem Transaktionsmanagement (Distributed Transaction Processing) für atomare Transaktionen, s. z. B. (Gray und Reuter, 1993).

Literaturverzeichnis

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

[ASN1 2006] *ASN.1 Information Site*. 2006. – URL `http://asn1.elibel.tm.fr/en/`. – (27.07.2006)

[Axis 2006] *Apache Axis*. 2006. – URL `http://ws.apache.org/axis/`. – Open Source Web Services Container (16.07.2006)

[Axis2 2006] *Apache Axis 2*. 2006. – URL `http://ws.apache.org/axis2/`. – Open Source Web Services Container (16.07.2006)

[Braig und Gemkow 2002] BRAIG, Andreas ; GEMKOW, Steffen: The BonSai Principle: Persistenz in der Java 2 Micro Edition. In: *Java Spektrum* (2002), Nr. 9/10, S. 28–32

[Brandner u. a. 2004] BRANDNER, Michael ; CRAES, Michael ; OELLERMANN, Frank ; ZIMMERMANN, Olaf: Web services-oriented architecture in production in the finance industry. In: *Informatik-Spektrum* 27 (2004), April, Nr. 2, S. 136–145

[BTP 2004] FURNISS, Peter u. a.: *Business Transaction Protocol Version 1.1.0*. November 2004. – URL `http://docs.oasis-open.org/business-transaction/business_transaction-btp-1.1-spec-cd-01.pdf`. – (20.07.2006)

[Buschmann u. a. 2004] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 2004

[Carr 2006] CARR, Harold: *Sun's Project Tango*. Juni 2006. – URL `http://java.sun.com/developer/technicalArticles/glassfish/ProjectTango/`. – (19.07.2006)

[Chappell 2004] CHAPPELL, David A.: *Enterprise Service Bus*. O'Reilly, 2004

[CrEme 2006] NSICOM: *CrEme - The Java Enabler for Windows CE*. 2006. – URL `http://www.nsicom.com/Default.aspx?tabid=138`. – (16.07.2006)

[Dokovski u. a. 2004] DOKOVSKI, Nikolai ; WIDYA, Ing ; HALTEREN, Aart van: *Paradigm Service Oriented Computing*. November 2004. – URL https://doc.telin.nl/dscgi/ds.py/Get/File-49216/D2.7b_-_Paradigm_-_Service_Oriented_Computing.pdf. – (27.07.2006)

[Dolan 2004] DOLAN, Liam: *SOAP in a Mobile Environment*, University of Dublin, Masterarbeit, September 2004. – URL https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-11.pdf. – (27.07.2006)

[Eclipse 2006] *Eclipse Development Platform*. 2006. – URL http://www.eclipse.org/. – (27.07.2006)

[EJB-2.1 2003] DEMICHIEL, Linda G.: *Enterprise JavaBeans™ Specification, Version 2.1*. November 2003. – URL http://java.sun.com/products/ejb/. – (27.07.2006)

[Endrei u. a. 2004] ENDREI, Mark ; ANG, Jenny ; ARSANJANI, Ali u. a.: *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, April 2004 (SG246303). – URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf. – (27.07.2006)

[Erl 2005] ERL, Thomas: *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson Education (Prentice Hall PTR), 2005

[Fielding 2000] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000

[Fowler 2003] FOWLER, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

[Garcia-Molina und Salem 1987] GARCIA-MOLINA, Hector ; SALEM, Kenneth: Sagas. In: *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. San Francisco, California, USA, Mai 1987, S. 249–259

[Gerlach 2005] GERLACH, Martin: Langlebige Transaktionen in dienstorientierten Umgebungen / Hochschule für Angewandte Wissenschaften Hamburg. URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master05-06/gerlach/abstract.pdf, Dezember 2005. – Forschungsbericht. Ausarbeitung zum Seminar (SR) des Master-Studiengangs Informatik an der HAW Hamburg (27.07.2006)

[Gerlach 2006] GERLACH, Martin: WS-Coordination und WS-BusinessActivity in Action mit Apache Axis / Hochschule für Angewandte Wissenschaften Hamburg. URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master05-06-proj/gerlach/paper.pdf, Januar 2006. – Forschungsbericht. Ausarbeitung zum Projekt (PJ) des Master-Studiengangs Informatik an der HAW Hamburg (22.02.2006)

[Gray 1981] GRAY, Jim: The Transaction Concept - Virtues and Limitations. In: *Proceedings of Seventh International Conference of Very Large Databases*, Tandem Computers Inc., September 1981

[Gray und Reuter 1993] GRAY, Jim ; REUTER, Andreas: *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993

[Grimm u. a. 2004] GRIMM, Robert ; DAVIS, Janet ; LEMAR, Eric ; MACBETH, Adam ; SWANSON, Steven ; ANDERSON, Thomas ; BERSHAD, Brian ; BORRIELLO, Gaetano ; GRIBBLE, Steven ; WETHERALL, David: System Support for Pervasive Applications. In: *ACM Transactions on Computer Systems* 22 (2004), Nr. 4, S. 421–486. – ISSN 0734-2071

[Hohendahl 2005] HOHENDAHL, Stephanie: *Web Service Orchestration - Modellierung komplexer Geschäftsprozesse mit BPEL*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, August 2005

[IZT 2001] OERTEL, Britta ; HEINZE, Michael ; BEYER, Lothar u. a.: Entwicklung und zukünftige Bedeutung mobiler Multimediadienste / Institut für Zukunftsstudien und Technologiebewertung (IZT). Berlin, Dezember 2001. – Forschungsbericht. – URL http://www.izt.de/publikationen/werkstattberichte/wb49_-_mobile_multimediadienste.html. (23.07.2006)

[J2ME 2006] *Java 2 Micro Edition*. 2006. – URL http://java.sun.com/j2me/. – (02.03.2006)

[J9 2006] IBM: *WebSphere Everyplace Micro Environment*. 2006. – URL http://www-306.ibm.com/software/wireless/weme/. – (16.07.2006)

[Jacobsen u. a. 1997] JACOBSEN, Eyðun E. ; KRISTENSEN, Bent B. ; NOWACK, Palle: Characterising patterns in framework development. In: *Technology of Object-Oriented Languages and Systems. TOOLS 25, Proceedings*, November 1997, S. 121–142

[Java API 2006] *Java Technology API Specifications*. 2006. – URL http://java.sun.com/reference/api/. – (27.07.2006)

[JBoss 2006] *JBoss Application Server*. 2006. – URL http://www.jboss.com/products/jbossas. – (27.07.2006)

[JBoss-Eclipse 2006] *JBoss Eclipse IDE*. 2006. – URL `http://www.jboss.com/products/jbosside`. – (27.07.2006)

[JSR-101 2003] CHINNICI, Roberto: *JSR 101: Java APIs for XML based RPC (JAX-RPC)*. Oktober 2003. – URL `http://www.jcp.org/en/jsr/detail?id=101`. – (27.07.2006)

[JSR-118a 2002] PEURSEM, Jim van ; WARDEN, James: *JSR 118: Mobile Information Device Profile 2.0*. November 2002. – URL `http://www.jcp.org/en/jsr/detail?id=118`. – (27.07.2006)

[JSR-118b 2006] PEURSEM, Jim van ; WARDEN, James: *JSR 118: Mobile Information Device Profile 2.1*. Mai 2006. – URL `http://www.jcp.org/en/jsr/detail?id=118`. – (05.08.2006)

[JSR-12 2003] RUSSELL, Craig: *JSR 12: Java Data Objects (JDO)*. Mai 2003. – URL `http://www.jcp.org/en/jsr/detail?id=12`. – (05.08.2006)

[JSR-139 2003] TAIVALSAARI, Antero: *JSR 139: Connected, Limited Device Configuration 1.1*. März 2003. – URL `http://www.jcp.org/en/jsr/detail?id=139`. – (27.07.2006)

[JSR-156 2005] LITTLE, Mark: *JSR 156: Java API for XML Transactions*. August 2005. – URL `http://www.jcp.org/en/jsr/detail?id=156`. – (27.07.2006)

[JSR-216 2005] COURTNEY, Jon ; CALDER, Bartley: *JSR 216: Personal Profile 1.1*. August 2005. – URL `http://www.jcp.org/en/jsr/detail?id=216`. – (16.07.2006)

[JSR-218 2005] COURTNEY, Jon: *JSR 218: Connected Device Configuration 1.1*. August 2005. – URL `http://www.jcp.org/en/jsr/detail?id=218`. – (16.07.2006)

[JSR-95 2006] ROBINSON, Ian: *JSR 95: J2EE™ Activity Service for Extended Transactions*. Februar 2006. – URL `http://www.jcp.org/en/jsr/detail?id=95`. – (27.07.2006)

[JTA 2002] CHEUNG, Susan ; MATENA, Vlada: *Java Transaction API (JTA) Version 1.0.1B*. November 2002. – URL `http://java.sun.com/products/jta/`. – (27.07.2006)

[Kandula 2006] *Apache Kandula*. 2006. – URL `http://ws.apache.org/kandula/`. – Versuch einer Open Source Implementierung von WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity auf Basis von Axis und Axis2 (16.07.2006)

[Khalaf und Leymann 2003] KHALAF, Rania ; LEYMANN, Frank: On Web Services Aggregation. In: *Proceedings of the VLDB Technologies for e-Services Workshop*. Berlin : Springer LNCS, September 2003

[Krafzig u. a. 2005] KRAFZIG, Dirk ; BANKE, Karl ; SLAMA, Dirk: *Enterprise SOA: Service-Oriented Architecture Best Practices*. Pearson Education (Prentice Hall PTR), 2005

[kSOAP2 2006] *kSOAP 2 Homepage*. 2006. – URL `http://ksoap2.sourceforge.net/`. – (27.07.2006)

[kXML2 2006] *kXML 2 Homepage*. 2006. – URL `http://kxml.sourceforge.net/kxml2/`. – (27.07.2006)

[Leymann 2003] LEYMANN, Frank: Web Services: Distributed Applications Without Limits. In: *Proceedings, Database Systems for Business, Technology, and Web (BTW)*. Leipzig : Springer LNCS, Februar 2003

[Leymann und Roller 1997] LEYMANN, Frank ; ROLLER, Dieter: Workflow Based Applications. In: *IBM Systems Journal* 36 (1997), Nr. 1

[Leymann u. a. 2002] LEYMANN, Frank ; ROLLER, Dieter ; SCHMIDT, Marc-Thomas: Web Services and Business Process Management. In: *IBM Systems Journal* 42 (2002), Nr. 2

[Li und Knudsen 2005] LI, Sing ; KNUDSEN, Jonathan: *Beginning J2ME*. Apress, 2005

[Limthanmaphon und Zhang 2004] LIMTHANMAPHON, Benchaphon ; ZHANG, Yanchun: Web Service Composition Transaction Management. In: *Proceedings of the fifteenth Australasian database conference (ADC2004)* Bd. 27, Januar 2004

[Little 2003a] LITTLE, Mark: Transactions and Web Services. In: *Communications of the ACM* 46 (2003), Nr. 10, S. 49–54

[Little 2003b] 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*. Pearson Education (Prentice Hall PTR), 2004

[MIDP-Bench 2006] *MIDP telephones benchmark*. 2006. – URL `http://www.club-java.com/TastePhone/J2ME/MIDP_Benchmark.jsp`. – (23.07.2006)

[MS-NET-CF 2006] *Microsoft .NET Framework Developer Center - .NET Compact Framework*. 2006. – URL `http://msdn.microsoft.com/netframework/programming/netcf/default.aspx`. – (24.07.2006)

[MS-WCF 2006] MICROSOFT: *Microsoft Windows Communication Foundation*. 2006. – URL `http://msdn.microsoft.com/winfx/technologies/communication/default.aspx`. – Bestandteil des .NET Framework 3.0 (19.07.2006)

[Papazoglou 2003] PAPAZOGLOU, Michael P.: Web Services and Business Transactions. In: *World Wide Web: Internet and Web Information Systems* 6 (2003), Nr. 1, S. 49–91

[Parsons 2005] PARSONS, David: Java Architectures for Mobilised Enterprise Systems. In: *Proceedings of the 38th Hawaii International Conference on System Sciences*, IEEE, 2005, S. 1–9

[SAAJ 2005] JAYANTI, V B K. ; HADLEY, Marc: *SOAP with Attachments API for Java (SAAJ) 1.3*. 2005. – URL `http://java.sun.com/webservices/saaj/index.jsp`. – (27.07.2006)

[Sandoz u. a. 2003] SANDOZ, Paul ; PERICAS-GEERTSEN, Santiago ; KAWAGUCHI, Kohuske ; HADLEY, Marc ; PELEGRI-LLOPART, Eduardo: *Fast Web Services*. August 2003. – URL `http://java.sun.com/developer/technicalArticles/WebServices/fastWS/`. – (27.07.2006)

[Sandoz u. a. 2004] SANDOZ, Paul ; TRIGLIA, Alessando ; PERICAS-GEERTSEN, Santiago: *Fast Infoset*. Juni 2004. – URL `http://java.sun.com/developer/technicalArticles/xml/fastinfoset/`. – (27.07.2006)

[Schmidt und Buschmann 2003] SCHMIDT, Douglas C. ; BUSCHMANN, Frank: Patterns, Frameworks, and Middleware: Their Synergetic Relationships. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington DC, USA : IEEE Computer Society, 2003, S. 694–703

[Schmidt u. a. 2000] SCHMIDT, Douglas C. ; STAL, Michael ; ROHNERT, Hans ; BUSCHMANN, Frank: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000

[Schrörs 2005] SCHRÖRS, Christoph: *Entwicklung einer Web Service Middleware für mobile Endgeräte*, Hochschule für Angewandte Wissenschaften Hamburg, Diplomarbeit, Dezember 2005

[Schrörs 2006a] SCHRÖRS, Christoph: *Erweiterung der Web Service Middleware für mobile Endgeräte*. Mai 2006. – Verfügbar als Teil des auf der beiliegenden CD ausgelieferten Source-Codes des Transaktions-Frameworks.

[Schrörs 2006b] SCHRÖRS, Christoph: Handbuch für Framework-Entwickler / Hochschule für Angewandte Wissenschaften Hamburg. Juli 2006. – Forschungsbericht. Handbuch zur (Weiter-)Entwicklung des Frameworks aus (Schrörs, 2006a). Verfügbar auf der beiliegenden CD.

[Schrörs 2006c] SCHRÖRS, Christoph: Handbuch für Web Service und Client Entwickler / Hochschule für Angewandte Wissenschaften Hamburg. Juli 2006. – Forschungsbericht. Handbuch zur Implementierung von Web Services für das Framework aus (Schrörs, 2006a). Verfügbar auf der beiliegenden CD.

[Siegel 2000] SIEGEL, Jon: *CORBA 3 Fundamentals and Programming*. OMG Press, 2000

[SOAP 2003] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noad u. a.: *SOAP Version 1.2*. 2003. – URL `http://www.w3.org/TR/soap/`. – (22.07.2006)

[Sun-JWTK 2006] SUN MICROSYSTEMS: *Sun Java Wireless Toolkit for CLDC*. 2006. – URL `http://java.sun.com/products/sjwtoolkit/`. – (16.07.2006)

[Szyperski 1998] SZYPERSKI, Clemens: *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1998

[Tai u. a. 2004] TAI, Stefan ; KHALAF, Rania ; MIKALSEN, Thomas: Composition of Coordinated Web Services. In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Oktober 2004, S. 294–310

[Tanenbaum und van Steen 2002] TANENBAUM, Andrew S. ; STEEN, Martin van: *Distributed Systems - Principles and Paradigms*. Pearson Education (Prentice Hall), 2002

[Thomé 2006] THOMÉ, Mark: Java 2 Micro Edition und .NET Compact Framework - Laufzeitumgebungen für mobile Anwendungen / Hochschule für Angewandte Wissenschaften Hamburg. Januar 2006. – Forschungsbericht. Präsentation zum Seminar Anwendungen 2 des Master-Studiengangs Informatik an der HAW Hamburg (02.08.2006)

[UDDI 2006] *OASIS UDDI*. 2006. – URL `http://www.uddi.org/`. – (24.07.2006)

[UML 2006] OBJECT MANAGEMENT GROUP: *UML® Resource Page*. 2006. – URL `http://www.uml.org/`. – (02.08.2006)

[WAS60-AT 2006] IBM: *Web Services Atomic Transaction support in WebSphere Application Server*. 2006. – URL `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/topic/com.ibm.websphere.express.doc/info/exp/ae/cjta_wstran.html`. – (27.07.2006)

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

[WS-Addr 2004] BOX, Don ; CURBERA, Francisco u. a.: *Web Services Addressing (WS-Addressing)*. August 2004. – URL `http://www.w3.org/Submission/ws-addressing/`. – (20.07.2006)

[WS-Arch 2004] BOOTH, David ; HAAS, Hugo ; MCCABE, Francis u. a.: *Web Services Architecture*. Februar 2004. – URL http://www.w3.org/TR/ws-arch/. – 20.07.2006

[WS-AT 2005] FEINGOLD, Max u. a.: *Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.0*. August 2005. – URL ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf. – (20.07.2006)

[WS-AT 2006] LITTLE, Mark ; WILKINSON, Andrew u. a.: *Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1*. März 2006. – URL http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-cd-01.pdf. – (20.07.2006)

[WS-BA 2005] FEINGOLD, Max u. a.: *Web Services Business Activity Framework (WS-BusinessActivity) Version 1.0*. August 2005. – URL ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf. – (20.07.2006)

[WS-BA 2006] FREUND, Tom ; GREEN, Alastair ; HARBY, John ; LITTLE, Mark u. a.: *Web Services Business Activity (WS-BusinessActivity) Version 1.1*. März 2006. – URL http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-cd-01.pdf. – (20.07.2006)

[WS-BPEL 2003] THATTE, Satish u. a.: *Business Process Execution Language for Web Services Version 1.1*. Mai 2003. – URL ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf. – (20.07.2006)

[WS-C 2005] FEINGOLD, Max u. a.: *Web Services Coordination (WS-Coordination) Version 1.0*. August 2005. – URL ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf. – (20.07.2006)

[WS-C 2006] FEINGOLD, Max u. a.: *Web Services Coordination (WS-Coordination) Version 1.1*. März 2006. – URL http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-cd-01.pdf. – (20.07.2006)

[WS-CAF 2006] *OASIS Web Services Composite Application Framework (WS-CAF) TC*. 2006. – URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf. – (20.07.2006)

[WS-CF 2003] LITTLE, Mark ; NEWCOMER, Eric u. a.: *Web Services Coordination Framework (WS-CF) Version 1.0*. Juli 2003. – URL http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CF.pdf. – (20.07.2006)

[WS-CTX 2003] LITTLE, Mark ; NEWCOMER, Eric u. a.: *Web Services Context (WS-Context) Version 1.0*. 2003. – URL http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf. – (20.07.2006)

[WS-P 2006] SCHLIMMER, Jeffrey u. a.: *Web Services Policy Framework (WS-Policy)*. März 2006. – URL http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf. – (22.07.2006)

[WS-Rel 2004] IWASA, Kazunori u. a.: *WS-Reliability 1.1 (OASIS Standard)*. November 2004. – URL http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf. – (20.07.2006)

[WS-RM 2005] FERRIS, Christopher ; LANGWORTHY, David u. a.: *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*. Februar 2005. – URL ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf. – (20.07.2006)

[WS-Sec 2006] NADALIN, Anthony ; KALER, Chris ; MONZILLO, Ronald ; HALLAM-BAKER, Phillip: *Web Service Security: SOAP Message Security 1.1*. Februar 2006. – URL http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf. – (24.07.2006)

[WS-TX 2006] *OASIS Web Services Transaction (WS-TX) TC*. 2006. – URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx. – (20.07.2006)

[WS-TXM 2003] LITTLE, Mark ; NEWCOMER, Eric u. a.: *Web Services Transaction Management (WS-TXM) Version 1.0*. Juli 2003. – URL http://www.arjuna.com/library/specs/ws_caf_1-0/WS-TXM.pdf. – (20.07.2006)

[WSDL-1.1 2001] CHRISTENSEN, Erik ; CURBERA, Francisco ; MEREDITH, Greg ; WEERAWARANA, Sanjiva: *Web Services Description Language (WSDL) Version 1.1*. März 2001. – URL http://www.w3.org/TR/wsdl. – (22.07.2006)

[WSDL-2 2006] CHINNICI, Roberto ; MOREAU, Jean-Jacques ; RYMAN, Arthor ; WEERAWARANA, Sanjiva: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. März 2006. – URL http://www.w3.org/TR/wsdl20/. – (22.07.2006)

[WSS 2006] *OASIS Web Service Security (WSS) TC*. 2006. – URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss. – (24.07.2006)

[Zimmermann u. a. 2005] ZIMMERMANN, Olaf ; DOUBROVSKI, Vadim ; GRUNDLER, Jonas ; HOGG, Kerard: Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rational, Concepts, Lessons Learned. In: *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM Press, 2005

[Zimmermann u. a. 2004] ZIMMERMANN, Olaf ; MILINSKI, Sven ; CRAES, Michael ; OELLERMANN, Frank: Second generation web services-oriented architecture in production in the finance industry. In: *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM Press, 2004, S. 283–289

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

Versicherung über die Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungs- und Studienordnung des Masterstudiengangs Informatik an der Hochschule für Angewandte Wissenschaften Hamburg vom 22. November 2001, geändert am 07. Dezember 2004, nach §22(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10. August 2006 Martin Gerlach