Seminar: Grundlagen webbasierter Informationssysteme

Thema: Transaktionen in Web-Service- und Grid-Umgebungen

von Thomas Jörg

Betreuer: Dipl.-Inform. Michael Haustein

23. Juni 2004

Inhaltsverzeichnis

Einleitung....3
1. Transaktionen in Web Services....3
1.1 WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity....3
1.1.1 Web Services Coordination (WS-Coordination)....3
1.1.1.1 Coordination service....4
1.1.1.2 Coordination context....4
1.1.1.3 Activation service....6
1.1.1.4 Registration service....7
1.1.1.5 Importing an activity....9
1.1.2 Web Services Atomic Transaction (WS-AtomicTransaction)....9
1.1.2.1 Completion protocol....10
1.1.2.2 Two-Phase Commit Protocol....11
1.1.2.2.1 Volatile Two-Phase Commit Protocol....12
1.1.2.2.2 Durable Two-Phase Commit Protocol....12
1.1.2.3 Abschließendes Beispiel....12
1.1.3 Web Services Business Activity Framework (WS-BusinessActivity)....13
1.1.3.1 BusinessAgreementWithParticipantCompletion....15
1.1.3.2 BusinessAgreementWithCoordinatorCompletion....16
1.1.3.3 Abschließendes Beispiel....16
1.2 Business Transaction Protocol....17
1.2.1 Business transactions....18
1.2.2 Rollen in einer business transaction....18
1.2.3 Zwei-Phasen-Commit im BTP....20
1.2.3.1 Prepare phase....21
1.2.3.2 Confirm phase....22
1.2.4 Abschließendes Beispiel....22
2. Transaktionen in Grid-Umgebungen....23
Zusammenfassung....25
Literaturverzeichnis....26

Einleitung

Web Services sind Dienste, die im Internet bereitgestellt und vollautomatisch genutzt werden können. Sie bieten eine interessante Möglichkeit, um Geschäftsprozesse, wie beispielsweise das Bestellen von Waren, ohne menschliche Eingriffe abwickeln zu können. Bestehende Workflow- oder ERP-Systeme, die bis jetzt auf einzelne Unternehmungen begrenzt waren, können mittels Web Services über Plattformgrenzen hinweg zusammenarbeiten. Für das Zustandekommen von Verträgen, die automatisch zwischen Informationssystemen verschiedener Organisationen geschlossen werden, muss garantiert werden, dass die Vertragspartner zu einer konsistenten Übereinkunft kommen. Durch Anwendung des Transaktionsparadigmas wird die nötige Zuverlässigkeit bei solchen kritischen Geschäftsprozessen erreicht. Eine Übertragung dieses Konzepts in die Welt der Web Services liegt nahe.

Im ersten Teil dieser Ausarbeitung werden Ansätzen zur Integration von Transaktionen in Web Services erläutert. Das Framework bestehend aus den Spezifikationen *WS-Coordination*, *WS-AtomicTransaction* und *WS-BusinessActivity* von BEA, IBM und Microsoft wird detailliert beschrieben. Anschließend wird auf einen alternativen Ansatz in Form der BTP-Spezifikation von OASIS eingegangen und Unterschiede sowie Gemeinsamkeiten der beiden Lösungen aufgezeigt. Beide Frameworks sind bis jetzt nicht standardisiert.

Der zweite Teil dieser Ausarbeitung befasst sich mit Transaktionen in Grid-Systemen. Auf diesem Gebiet werden erst seit jüngster Zeit Forschungen betrieben. Diese Ausarbeitung gibt einen Überblick über die verschiedenen Ansätze und bisher erzielte Ergebnisse.

1. Transaktionen in Web Services

1.1 WS-Coordination, WS-AtomicTransaction und WS-BusinessActivity

Die Spezifikationen *Web Services Coordination (WS-Coordination)* [1], *Web Services Atomic Transaction (WS-AtomicTransaction)* [2] und *Web Services Business Activity Framework (WS-BusinessActivity)* [3] sind eine Entwicklung von BEA Systems, IBM und Microsoft. Sie ermöglichen die Realisierung verteilten transaktionalen Verhaltens in einer Web-Service-Umgebung. Jede Spezifikation konzentriert sich auf begrenzte Aspekte dieses Problems. Man könnte sie als Bausteine betrachten, die erst zusammengesetzt zu einer Lösung führen. Durch dieses Modulprinzip wird ein Framework geschaffen, das künftig um Funktionalität erweitert werden kann, ohne dass Änderungen bereits bestehender Teile nötig werden.

Die aktuellen Versionen der hier betrachteten Spezifikationen wurden im September 2003 bzw. Januar 2004 veröffentlicht. Auf die ebenfalls von BEA, IBM und Microsoft entwickelte und im August 2002 publizierte Spezifikation *WS-Transaction* wird im Folgenden nicht näher eingegangen. Sie wurde durch *WS-AtomicTransaction* und *WS-BusinessActivity* abgelöst, wobei erstere Part I und letztere Part II der Spezifikation ersetzt.

1.1.1 Web Services Coordination (WS-Coordination)

Durch Nutzung von Web Services wird es Applikationen ermöglicht, verschiedenste Dienste plattformübergreifend via Internet zu nutzen. Von der Applikation initiiert entstehen Einheiten verteilt ablaufender Verarbeitung, wobei die Zahl der Beteiligten groß sein kann und komplexe Beziehungen entstehen können. Eine solche Einheit wird als *activity* bezeichnet. *WS-Coordination* bildet ein erweiterbares Framework als Basis für Protokolle zur Sicherstellung konsistenter verteilter Zustandsübergänge im Rahmen einer *activity*. Die Spezifikation kann somit als Grundlage für die Integration von Transaktionen in eine Web-Service-Umgebung dienen, sie ist jedoch ausreichend allgemein, um auch andere Formen der Koordination zu unterstützen, die eine

wechselseitige Einigung aller Teilnehmer bezüglich des Ausgangs einer *activity* ermöglichen. Wie die Koordination der Teilnehmer letztlich abläuft ist hier jedoch nicht Gegenstand der Betrachtung. Für die Beschreibung konkreter Koordinationsmodelle, die als *coordination types* bezeichnet werden, wird vielmehr eine Basis geschaffen. Die ersten und bis jetzt einzigen beiden Beispiele hierfür sind *WS-AtomicTransaction* und *WS-BusinessActivity*.

Ein zentrales Konzept der hier vorgestellten Spezifikation ist der *coordination context*. Dieser ermöglicht unter anderem die Einbeziehung von Web Services in eine bestehende *activity*. Auf diesen Aspekt wird später detailliert eingegangen.

1.1.1.1 Coordination service

Ähnlich wie im klassischen 2-Phasen-Commit-Protokoll [4] wird auch in *WS-Coordination* eine zentrale Stelle zur Koordination zugeordneter Teilnehmer eingesetzt, die als *coordination service* oder kurz *coordinator* bezeichnet wird. Dieser sorgt für konsistente Zustandsübergänge im Rahmen einer *activity*. Dazu stellt dieser folgende Dienste bereit, die als spezielle Web Services realisiert sind:

- *Activation service:*
  Dieser Dienst erlaubt einer Applikation, die auch als Initiator bezeichnet wird, eine neue *activity* zu beginnen und veranlasst den *coordinator* zur Erzeugung eines zugehörigen *coordination context*.
- *Registration service:*
  Ermöglicht es einem Web Service sich für die Teilnahme an einer laufenden *activity* zu registrieren.
- Eine Reihe von *coordination protocol services:*
  Ein *coordination type* beinhaltet verschiedene *coordination protocols;* eine Menge wohldefinierter Nachrichtenformate und Regeln für den Austausch von konkreten Nachrichten zwischen dem Koordinator und den Teilnehmern einer *activity*. Die Arbeitsweise des *coordination protocol services* wird in der Spezifikation des verwendeten *coordination type* beschrieben.

1.1.1.2 Coordination context

Ein *coordination context* wird von einer Applikation mittels der *CreateCoordinationContext*-Operation des *activation service* erzeugt. Der Kontext wird, eingebettet im Header-Element, mit allen SOAP-Nachrichten im Rahmen einer *activity* verschickt.

An dieser Stelle wird kurz auf das Konzept der *endpoint reference* eingegangen, das Teil der *WS-Addressing* Spezifikation ist. Eine *endpoint reference* ermöglicht die Nutzung eines Web Service, indem sie folgende Informationen bereitstellt:

- die URI eines Web Service
- beliebige zusätzliche Daten, unverständlich für jeden, außer für den Erzeuger selbst. Diese Daten müssen in allen Nachrichten an den Web Service enthalten sein und werden in der Regel genutzt, um bestimmte Ressourcen in diesem anzusprechen.

Es folgt ein beispielhafter SOAP-Header, der einen *coordination context* enthält. Die einzelnen Elemente werden anschließend näher erläutert:

```
<soap:Envelope xml-ns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Header>
          ...
          <wscoor:CoordinationContext
               xml-ns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
               xml-ns:wscoor="http://schemas.xmlsoap.org/ws/2003/09/wscoor"
               xml-ns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
               xml-ns:myApp="http://myApplication.de/myApp"

               soap.mustUnderstand="true">

               <wsu:Expires>
                 2004-05-31T12:30:00Z
               </wsu:Expires>
               <wsu:Identifier>
                 http://myCoordinator.de/activity1
               </wsu:Identifier>
               <wscoor:CoordinationType>
                 http://schemas.xmlsoap.org/ws/2003/09/wsat
               </wscoor:CoordinationType>
               <wscoor:RegistrationService>
                    <wsa:Address>
                      http://myCoordinator.de/registrationService
                    </wsa:Address>
                    <wsa:ReferenceProperties>
                         <TransactionId="cfb01dc0-5073-405a-a3aea6038ecc476e">
                    </wsa:ReferenceProperties>
               </wscoor:RegistrationService>
          </wscoor:CoordinationContext>
          <myApp:IsolationLevel>
            RepeatableRead
          </myApp:IsolationLevel>
            ...
     </soap:Header>
     <soap:Body>
       ...
     </soap:Body>
</soap:Envelope>
```

//CoordinationContext/Expires

Dieses Element gibt einen TimeOut-Wert für den *coordination context* an. Im Beispiel wäre dies der 30. Mai 2004, 12:30Uhr (Z-Zeit bzw. UTC). Dieser Wert hat je nach verwendetem *coordination type* unterschiedliche Bedeutung für den Koordinationsprozess.

//CoordinationContext/Identifier

Eine *activity* erhält bei ihrem Beginn einen Identifikator. So kann jede Nachricht eindeutig einer *activity* zugeordnet werden.

//CoordinationContext/CoordinationType

Gibt den *coordination type* an, der in der *activity* Verwendung findet. Dieser stellt ein bestimmtes Koordinationsmodell dar, er regelt das Verhalten aller Beteiligten sowie den Nachrichtenfluss zwischen ihnen über die Spezifikationen aus *WS-Coordination* hinaus. Im Beispiel handelt es sich um eine ACID-Transaktion, die in *WS-AtomicTransaction* spezifiziert ist und durch das Schlüsselwort `http://schemas.xmlsoap.org/ws/2003/09/wsat` identifiziert wird.

//CoordinationContext/RegistrationService

Wenn ein Web Service eine SOAP-Nachricht erhält, die einen *coordination context*

aufweist, muss er sich zunächst beim Koordinator der *activity* als Teilnehmer registrieren. Dazu verwendet er die hier angegebene *endpoint reference*. Im Beispiel würde der Web Service eine *register*-Nachricht an `http://myCoordinator.de/registrationService` senden (siehe hierzu 1.1.1.4). Um die Transaktion zu identifizieren, für die sich der Dienst registrieren will, wird im Beispiel eine `TransactionId` eingesetzt. Diese Information ist nur für den Koordinator selbst von Relevanz, der Web Service kopiert diese blind in die Header der Protokollnachrichten, die an den Koordinator gerichtet sind. Der Einsatz einer `TransactionId` ist nicht in *WS-Coordination* vorgesehen, in der Praxis bietet sich dieses Verfahren jedoch an.
Um zu verhindern, dass sich ein Web Service blind an einer *activity* beteiligt, deren *coordination type* nicht von ihm unterstützt wird, muss das `mustUnderstand`-Attribut gesetzt sein.

//CoordinationContext/{any}
: Es ist möglich, applikationsspezifische Informationen im *coordination context* zu übermitteln. Im Beispiel wird ein zusätzliches `IsolationLevel`-Element verwendet.

//CoordinationContext/@{any}
: Auch der Einsatz zusätzlicher applikationsspezifischer Attribute im *coordination context* ist erlaubt. Im Beispiel wird hiervon kein Gebrauch gemacht.

1.1.1.3 Activation service

Um eine neue *activity* zu beginnen, sendet eine Applikation eine *CreateCoordinationContext*-Nachricht an den *activation service* eines ihr bekannten Koordinators. Diese Applikation wird als Initiator bezeichnet. Der Koordinator erzeugt einen *coordination context* und liefert diesen mittels einer *CreateCoordinationContextResponse*-Nachricht an die Applikation zurück (siehe Abbildung 1).

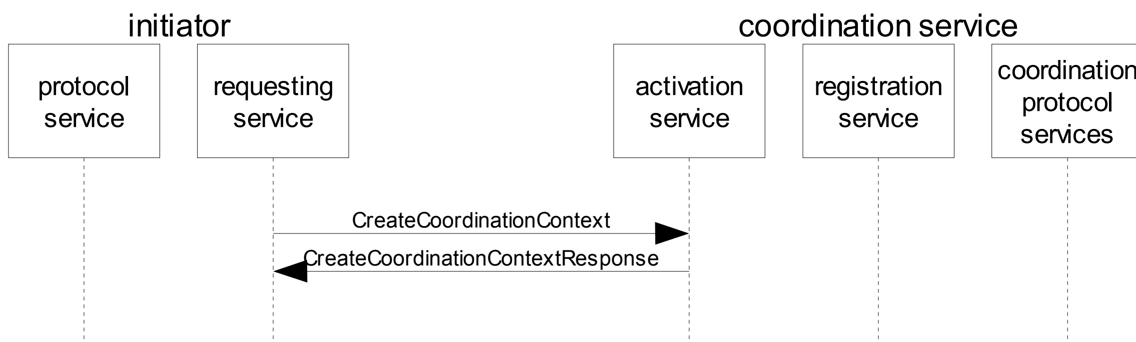

Abbildung 1: Eine Applikation nutzt den activation service

Das folgende XML-Fragment veranschaulicht den Aufbau einer *CreateCoordinationContext*-Nachricht:

```
<CreateCoordinationContext ...>
     <CoordinationType> ... </CoordinationType>
     <wsu:Expires> ... </wsu:Expires>
     <CurrentContext> ... </CurrentContext>
     ...
</CreateCoordinationContext>
```

/CreateCoordinationContext/CoordinationType

Der *coordination type,* der in der *activity* verwendet werden soll, wird durch einen eindeutigen Identifikator beschrieben. Im Fall von *WS-BusinessActivity* wäre diese beispielsweise `http://schemas.xmlsoap.org/ws/2004/01/wsba`.

/CreateCoordinationContext/wsu:Expires (optional)

Dieses Element gibt einen TimeOut-Wert für den *coordination context* an. Die Semantik dieses Parameters ist je nach verwendetem *coordination type* unterschiedlich.

/CreateCoordinationContext/CurrentContext (optional)

Wenn dieses Element nicht vorkommt, wird ein neuer *coordination context* erzeugt. Wird hingegen der *coordination context* einer laufenden *activity* übergeben, beteiligt sich der Koordinator an dieser. Dieser Vorgang wird als Importieren einer *activity* bezeichnet und in Abschnitt 1.1.1.5 detailliert beschrieben.

Die *WS-Coordination*-Spezifikation schlägt den Einsatz des `CurrentContext`-Elements auch bei Recovery vor, bleibt jedoch eine genauere Erläuterung dieses Aspektes schuldig.

Die Verwendung beliebiger zusätzlicher Elemente und Attribute zum Versenden weiterer Informationen ist möglich.

Nachdem der Koordinator die *CreateCoordinationContext*-Nachricht erhalten und den gewünschten Kontext erzeugt hat, sendet er diesen mittels einer *CreateCoordinationContextResponse*-Nachricht an die aufrufende Applikation zurück. Der Aufbau einer solchen Nachricht ist denkbar einfach und wird von folgendem XML-Fragment skizziert:

```
<CreateCoordinationContextResponse ...>
     <CoordinationContext> ... </CoordinationContext>
     ...
</CreatCoordinationCotextResponse>
```

Das einzig geforderte Element enthält den erzeugten *Coordination Context*. Dessen Aufbau wurde bereits in Abschnitt 1.1.1.2 erläutert. Einziger Unterschied ist, dass der Kontext hier im `body` einer SOAP-Nachricht versendet wird, nicht im `header`.

Die Verwendung beliebiger zusätzlicher Elemente und Attribute zum Versenden weiterer Informationen ist ebenfalls möglich.

1.1.1.4 Registration service

Nachdem eine Applikation eine *activity* begonnen und den zugehörigen *Coordination Context* erhalten hat, muss sie sich selbst zur Teilnahme registrieren. Dazu entnimmt sie dem Kontext die *endpoint reference* auf den *registration service* des Koordinators und sendet eine *register*-Nachricht. Folgendes XML-Fragment verdeutlicht deren Aufbau:

```
<Register ...>
     <ProtocolIdentifier> ... </ProtocolIdentifier >
     <ParticipantProtocolService> ... </ParticipantProtocolService>
     ...
</Register>
```

/Register/ProtocolIdentifier

Die Applikation entscheidet sich für ein bestimmtes *coordination protocol,* das vom *coordination type* der *activity* unterstützt wird und übergibt dessen Identifikator. Im Fall des *completion protocol* wäre das beispielsweise folgende URI:

`http://schemas.xmlsoap.org/ws/2003/09/wsat#Completion`.

/Register/ParticipantProtocolService

Die Applikation übergibt außerdem eine *endpoint reference,* an die der Koordinator künftige Protokollnachrichten senden soll.

Die Verwendung beliebiger zusätzlicher Elemente und Attribute zum Versenden weiterer Informationen ist möglich.

Es ist möglich, sich zur Teilnahme an mehr als einem Protokoll zu registrieren, dazu werden einfach mehrere *register*-Operationen ausgeführt.

Die Antwort auf eine *register*-Nachricht hat folgenden Aufbau:

```
<RegisterResponse ...>
     <CoordinationProtocolService> ... </CoordinationProtocolService>
     ...
</RegisterResponse>
```

Der Koordinator gibt in einer *registerResponse*-Nachricht eine *endpoint reference* zurück, die von der Applikation für künftige Protokollnachrichten verwendet werden soll. Auch hier ist die Verwendung beliebiger zusätzlicher Elemente und Attribute erlaubt.

Abbildung 2 zeigt den Nachrichtenfluss im Verlauf einer Registrierung.

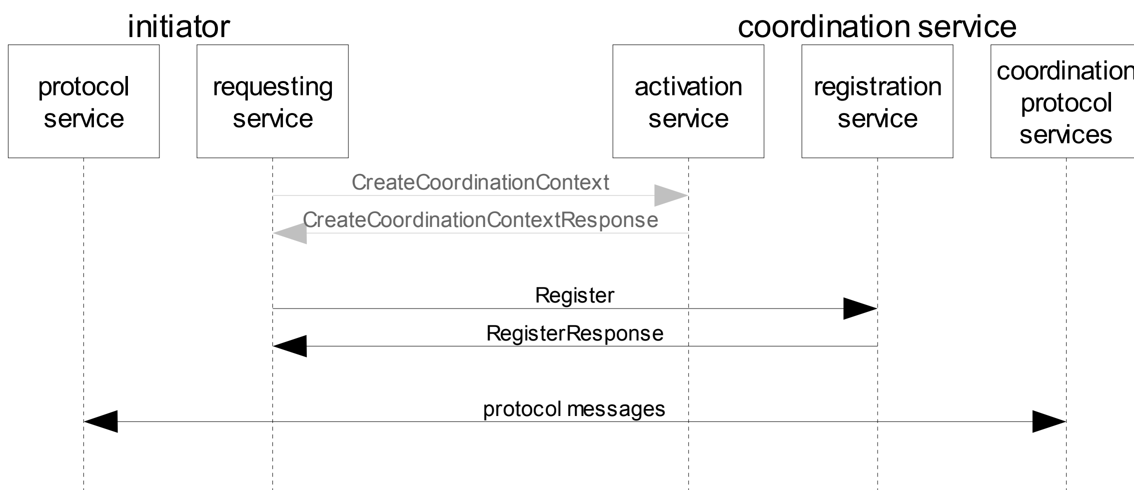

Abbildung 2: registration service

Nach den eben erläuterten Schritten haben beide Seiten, Applikation und Koordinator, eine *endpoint reference* auf den Protokolldienst des jeweils anderen. Nun können protokollspezifische Nachrichten versendet werden.

1.1.1.5 Importing an activity

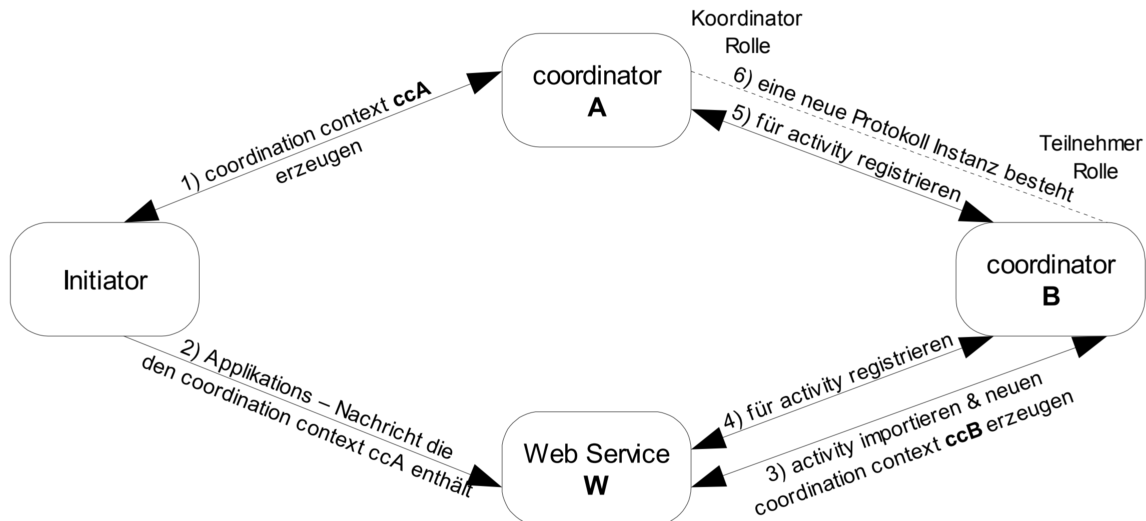

Abbildung 3: Ein Web Service installiert seinen eigenen Koordinator

Der *activation service* kann genutzt werden, um einen Koordinator in eine bestehenden *activity* einzubinden. Dieser Vorgang wird importieren oder *interposing* genannt. Erhält ein Web Service eine Applikationsnachricht, die einen *coordination context* beinhaltet, kann er sich beim Koordinator registrieren, der diesen erzeugt hat. Es gibt jedoch auch Gründe, einen eigenen Koordinator heranzuziehen, da dieser beispielsweise eine höhere Leistungsfähigkeit aufweist oder als vertrauenswürdiger gilt. Abbildung 3 zeigt diesen Vorgang im Detail:

1. Der Initiator veranlasst Koordinator A zur Erzeugung eines neuen *coordination context* ccA.
2. Der Initiator nutzt den Web Service W und fügt ccA in den Header seiner Nachricht ein.
3. W hat nun die Wahl: Er kann sich beim Koordinator A registrieren, oder einen anderen Koordinator beauftragen, die *activity* zu importieren. Im Beispiel wählt er die zweite Variante. Dazu nutzt er den *activation service* von Koordinator B, wobei er in der *CreateCoordinationContext*-Nachricht ccA als optionales `currentContext`-Element übergibt. B erzeugt daraufhin den *coordination context* ccB, der sich lediglich durch die *endpoint reference* auf seinen eigenen *registration service* von ccA unterscheidet.
4. W verwendet die *endpoint reference* aus ccB zur Registrierung bei B.
5. B registriert sich bei A, gewissermaßen als Stellvertreter für W. Dabei benutzt er das gleiche Interface wie gewöhnliche Web Services.
6. Es entsteht eine neue Protokoll Instanz, in der A die Rolle des Koordinators und B die Rolle eines Teilnehmers zukommt.

Koordinator B wird nun als *interposed* oder *subordinate coordinator* bezeichnet. Durch den beschriebenen Mechanismus ist die Bildung hierarchischer Strukturen beliebiger Tiefe möglich. Ein Koordinator kann hierbei auch mehrere *subordinate coordinators* unter sich haben.

1.1.2 Web Services Atomic Transaction (WS-AtomicTransaction)

Die WS-*AtomicTransaction*-Spezifikation führt einen *coordination type* ein, der das klassische Transaktionsparadigma in die Welt der Web Services überträgt. Den Beteiligten wird die wechselseitige Einigung auf den Ausgang einer *activity* ermöglicht. Falls die Transaktion fehlschlägt, werden alle Operationen im Rahmen der *activity* zurückgesetzt, kommt sie zu einem

erfolgreichen Ende, werden alle Änderungen dauerhaft beibehalten. Im Verlauf einer *activity* haben die Operationen keine nach außen sichtbare Auswirkungen, erst nach erfolgreichem Abschluss wird das Ergebnis für andere lesbar. Eine *atomic transaction* weist die ACID-Eigenschaften auf, die aus Datenbanksystemen bekannt sind. Die Übertragung dieser bekannten Prinzipien auf eine Web-Service-Umgebung geht nicht völlig problemlos vonstatten. Folgende Punkte müssen beim Einsatz von *atomic transactions* bedacht werden:

- Eine Transaktion sollte von kurzer Dauer sein, da Sperren auf Datenbeständen bis zu ihrem Ende gehalten werden müssen, um Isolation und Atomarität zu gewährleisten. Insbesondere sollte im Verlauf einer Transaktion nicht auf Benutzereingaben gewartet werden.
- Ein Web Service kann potentiell von jeder beliebigen Applikation via Internet verwendet werden. Eine *atomic transaction* bietet keinen Schutz vor *Denial-of-Service*-Angriffen, vor allem das Setzen von Sperren einer Datenbank sollte nur von vertrauenswürdiger Seite erfolgen dürfen.
- WS-AtomicTransaction basiert auf dem klassischen 2-Phase-Commit-Protokoll (2PC). Hier wird vorrausgesetzt, dass sich alle Beteiligten kooperativ verhalten. Daher sollte dieser *coordination types* nur im vertrauenswürdigen Umfeld eingesetzt werden.

Die Übertragung von ACID-Eigenschaften auf Web Services hat den Vorteil, dass existierenden Transaktionssystemen trotz verschiedener, potentiell inkompatibler eigener Protokolle die Zusammenarbeit über Plattformgrenzen hinweg ermöglicht wird.

WS-AtomicTransaction bietet drei Protokolle zum Abschluss von Transaktionen, die in den folgenden Abschnitten erläutert werden.

1.1.2.1 Completion protocol

Der Initiator einer *activity* registriert sich für das *completion protocol.* Es bietet ein Interface, mit dem der Koordinator angewiesen werden kann, die Transaktion zu beenden. Das folgende Diagramm in Abbildung 4 veranschaulicht das Protokoll.

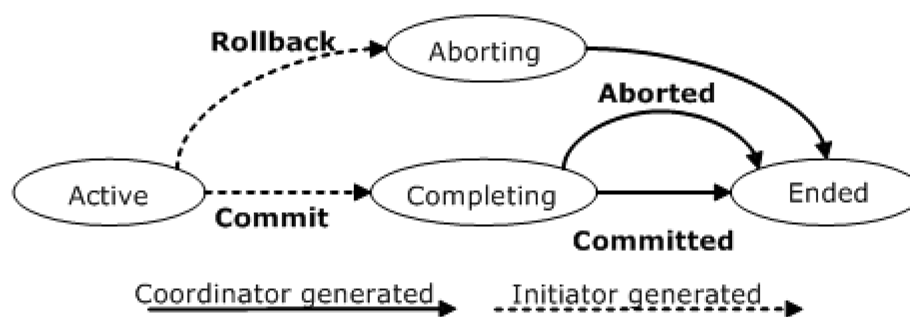

Abbildung 4: abstrakte Darstellung des completion protocol

Der Initiator hat die Wahl, die Transaktion entweder zurückzusetzen oder erfolgreich zu beenden, falls dies möglich ist. Dazu sendet er eine **Rollback-** bzw. eine **Commit-**Nachricht an den Koordinator. Dieser versucht, die Transaktion in der gewünschten Weise zu beenden und teilt anschließend das Ergebnis mit.

Die *WS-AtomicTransaction*-Spezifikation macht keine Aussage darüber, welche Teilnehmer sich für das *completion protocol* registrieren können und ob der Initiator Einfluss darauf hat. Die Semantik des *completion protocol* ist im Bezug auf untergeordnete Koordinatoren ebenfalls unklar.

1.1.2.2 Two-Phase Commit Protocol

Das *Two-Phase-Commit*-Protokoll (2PC) führt eine Einigung der teilnehmenden, verteilten Dienste auf den Ausgang einer Transaktion herbei. Diese Entscheidung geschieht in zwei Phasen:

1. *Prepare phase*:
   Alle Beteiligten werden vom Koordinator angewiesen, sich auf das Ende der Transaktion vorzubereiten. Falls diese in der Lage sind, die Transaktion erfolgreich zu beenden, antworten sie mit **Prepared**, anderenfalls mit **Aborted**.
   Web Services, die nur als Leser an der Transaktion beteiligt waren, können mit **ReadOnly** antworten, um nicht an der Commit-Phase teilnehmen zu müssen.
2. *Commit phase*:
   Nur falls sich alle Beteiligten der Transaktion für **Prepared** bzw. **ReadOnly** entschieden haben, kann die Transaktion erfolgreich beendet werden, ansonsten muss sie zurückgesetzt werden. Der Koordinator wartet, bis alle Stimmen abgegeben wurden und informiert die Beteiligten über das Resultat.

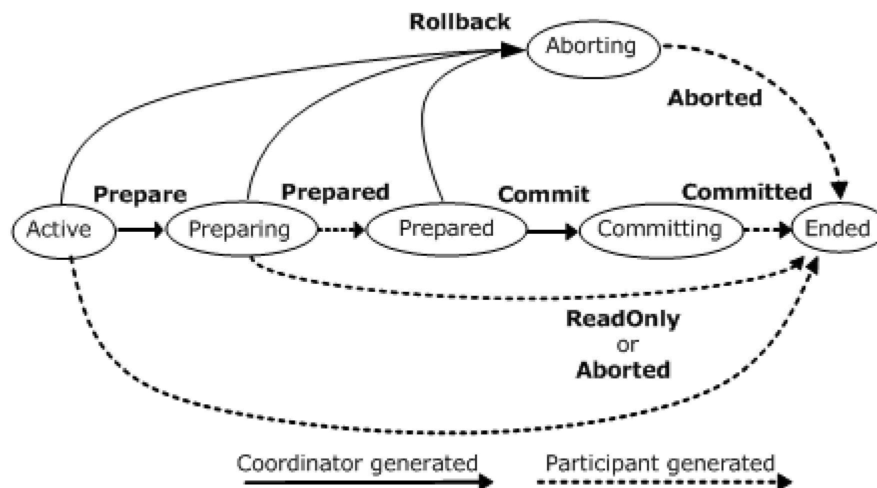

Abbildung 5: abstrakte Darstellung des 2PC protocol

Abbildung 5 zeigt die Zustände, die im Verlauf eines 2PC-Protokolls erreicht werden können und die Nachrichten, die die jeweiligen Zustandsübergänge auslösen.

Das `Expires`-Element des *coordination context* wird im 2PC-Protokoll als TimeOut für die Prepare-Phase verstanden. Falls diese bis zum Ablauf des TimeOut nicht beendet ist, entscheidet sich der Koordinator einseitig, die Transaktion zurückzusetzen. Nach Abschluss der Prepare-Phase hat der *expires*-Wert keine Bedeutung mehr.

Zu kritisieren ist, dass bei Verwendung eines 2PC-Protokolls alle Teilnehmer durch den Ausfall des Koordinators während der Prepare-Phase blockiert werden. Das gleiche Problem tritt bei einer Störung der Kommunikation zwischen einem Teilnehmer und dem Koordinator auf. Keiner der Teilnehmer verfügt über zentrales Wissen, um zu entscheiden, ob die Transaktion erfolgreich beendet werden kann oder zurückgesetzt werden muss. Diesen bleibt keine andere Möglichkeit, als auf die Recovery des Koordinators bzw. die Wiederherstellung der Verbindung zu warten, um die Transaktion zu beenden. Das ist insbesondere deshalb kritisch, da während des gesamten Zeitraums alle Sperren gehalten werden müssen. Das Drei-Phasen-Commit-Protokoll wäre eine geeignete Lösung für dieses Problem, es wurde in der Spezifikation jedoch nicht berücksichtigt.

WS-AtomicTransaction definiert zwei Varianten des Two-Phase-Commit-Protokolls: das *durable*

2PC coordination protocol und das *volatile 2PC coordination protocol.* Es besteht die Möglichkeit, an mehreren Protokollen gleichzeitig teilzunehmen.

1.1.2.2.1 Volatile Two-Phase Commit Protocol

Das *volatile 2PC protocol* ist für Web Services vorgesehen, die flüchtige Ressourcen, wie beispielsweise Caches, verwalten. Nach Erhalt einer Commit-Nachricht im Rahmen des *completion protocol*, beginnt der Koordinator die Prepare-Phase für alle registrierten Teilnehmer des *volatile 2PC protocol.* Bis zur deren Beendigung ist eine Registrierung für das *durable 2PC protocol* weiterhin möglich. Danach startet der Koordinator die Prepare-Phase für die Teilnehmer des *durable 2PC protocol.* Den Teilnehmer am *volatile 2PC protocol* wird <u>nicht</u> garantiert, dass sie informiert werden, ob die Transaktion mit Commit oder Abort beendet wurde.

Im klassischen 2PC-Protokoll ist kein Verfahren vorgesehen, dass mit der *volatile* Variante vergleichbar wäre. Diese Neuerung trägt einer Entwicklung des Umfelds Rechnung, in der Transaktionen ablaufen [5]. Zum Beispiel verwendet eine typische Web-Applikation front-end Server, die für die Präsentation zuständig sind, middle-tier Application Server für die Datenverarbeitung und back-end Datenbanken für die Datenhaltung. Die middle-tier Server schreiben Daten in der Regel nicht ständig in die DB zurück, sondern speichern diese zunächst zwischen. Um Transaktionen in diesem Umfeld zu ermöglichen, wurde das *volatile 2PC protocol* eingeführt. Dem middle-tier Server wird die Möglichkeit gegeben, Änderungen, die er im Cache zwischengespeichert hat, rechtzeitig in die Datenbank zu schreiben, bevor die (*durable*) Prepare Phase beginnt.

1.1.2.2.2 Durable Two-Phase Commit Protocol

Teilnehmer, die nicht flüchtige Speicher wie beispielweise Datenbanken verwalten, sollten sich für das *durable 2PC protocol* registrieren. Nachdem die *volatile*-Prepare-Phase erfolgreich beendet wurde, beginnt der Koordinator die Prepare-Phase für alle Teilnehmer des *durable 2PC protocol.* Eine Transaktion wird erfolgreich beendet, falls der Koordinator ausschließlich Commit bzw. ReadOnly-Nachrichten erhält. In diesem Fall informiert er alle Teilnehmer <u>beider</u> 2PC-Protokolle, dass die Transaktion erfolgreich beendet werden kann.

1.1.2.3 Abschließendes Beispiel

Das Beispiel in Abbildung 6 zeigt wie WS-AtomicTransaction dazu genutzt werden kann, um Transaktionen über Grenzen von Datenbanksystemen hinweg durchzuführen, vorausgesetzt diese stellen bestimmte Operationen als Web Service zur Verfügung. Das 2PC-Protokoll sichert, dass die verteilten Operationen eine atomare Einheit bilden.

Im Beispiel nutzt der *initiator* Dienste, die von zwei verschiedenen Datenbanksystemen zur Verfügung gestellt werden und als Einheit erfolgreich ausgeführt werden bzw. scheitern sollen. Dazu beteiligen sich beide Web Services an einer *atomic transaction*. Eines der Datenbanksysteme verändert seinen Zustand im Rahmen der Transaktion nicht, daher meldet der zugehörige Web Service in der Prepare-Phase **ReadOnly** an der Koordinator. Der zweite Web Service sendet **Prepared**, somit kann die Transaktion erfolgreich beendet werden.

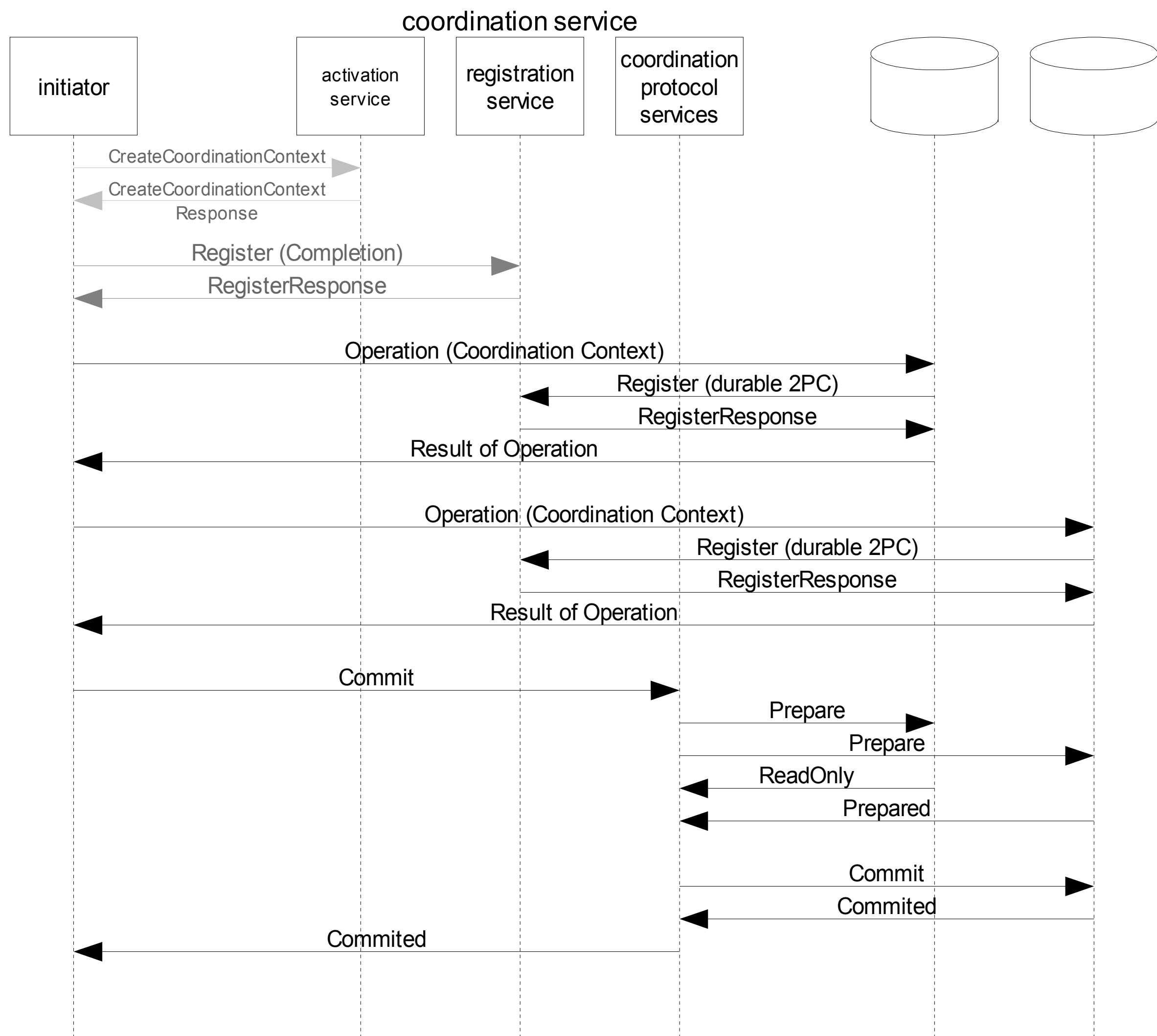

Abbildung 6: Beispielhafter Ablauf einer einer 2PC-Protokollinstanz

1.1.3 Web Services Business Activity Framework (WS-BusinessActivity)

WS-BusinessActivity spezifiziert den *coordination type business activity*, basierend auf dem erweiterbaren *WS-Coordination* Framework. Eine *business activity* ermöglicht den Teilnehmer die wechselseitige Einigung auf verteilt auszuführende Operationen. Im Gegensatz zur *atomic transaction,* ist dieser *coordination type* auch für den Einsatz bei langlebigen *activities* mit Teilnehmern aus verschiedenen *trust domains* gedacht. Kurze *atomic transactions* können Teil einer *business activity* sein.

Eine *business activity* ist zusammengesetzt aus einzelnen *business tasks*, die auch als *scopes* bezeichnet werden. Ein *scope* kann wiederum mehrere untergeordnete *child scopes* enthalten. Diese Art der Verschachtelung wird als *nested scope* bezeichnet und ermöglicht den Aufbau von Hierarchien beliebiger Tiefe. Die Applikation an der Spitze dieser Hierarchie hat die Möglichkeit auszuwählen, welche *child scopes* Einfluss auf den Ausgang der *activity* haben sollen. Im Gegensatz zu einer *atomic transaction* kann eine *business activity* auch beim Scheitern einzelner Teilaufgaben erfolgreich beendet werden. Soll zum Beispiel eine Warenlieferung bei einem möglichst günstigen Zulieferer bestellt werden, könnten im Rahmen einer business activity mehrere Angebote eingeholt werden. Falls einer der Zulieferer kein Angebot unterbreiten kann, ist eine erfolgreiche Abwicklung der Bestellung trotzdem möglich. Diese Entscheidung muss von Anwendungslogik getroffen

werden und geht über *WS-BusinessActivity* Spezifikation hinaus.

Ein *child scope* kann der Applikation sein Scheitern durch Werfen einer Exception signalisieren. Dies geschieht asynchron, nicht erst auf eine Anfrage hin, die mit der Prepare-Phase des 2PC-Protokolls vergleichbar wäre. Die Applikation reagiert entsprechend auf die Exception und kann die Verarbeitung gegebenenfalls trotz des Fehlers fortsetzen. Falls keiner der Zulieferer des eben erwähnten Beispiels liefern kann, hat die Applikation die Möglichkeit, neue Händler mittels UDDI zu suchen und dort die gewünschten Waren zu bestellen. *Exception handler* werden durch Anwendungslogik realisiert und sind nicht Gegenstand von *WS-BusinessAcitiy*.

Die Isolationseigenschaft, die aus dem klassischen Transaktionsparadigma bekannt ist, wird von *WS-BusinessActivity* aufgeweicht. Grund dafür ist, dass eine *business activity* sehr langlebig sein kann – Benutzereingaben, Fertigung oder Lieferung von Waren können Voraussetzung für deren Beendigung sein. Daher ist das Halten von Sperren auf Ressourcen, wie es in der Regel bei ACID-Transaktionen der Fall ist, ausgesprochen problematisch. Es wird der Kompromiss nötig, Zwischenergebnisse im Verlauf einer *business activity* nach außen sichtbar zu machen bevor diese beendet wird. Durch die vorzeitige Sperrfreigabe wird die Verfügbarkeit von Ressourcen gesteigert. Ein Zurücksetzen im Sinne von Rollback, wird dadurch jedoch unmöglich. Als Alternative ist in *WS-BusinessActivity* der Einsatz von Kompensation vorgesehen. Dazu wird einem *scope* ein *compensation task* zugeordnet, der in der Lage ist, die ausgeführten Operationen rückgängig zu machen. Rückgängig darf nicht im Sinne von ungeschehen verstanden werden, wie es bei einem Rollback zutreffend wäre. Das Resultat eines *scope* kann trotz Kompensation weiterhin Auswirkungen auf die reale Welt haben. Zum Beispiel wäre ein *compensation task* für eine Bestellung eine korrespondierende Stornierung. Durch Bearbeitungsgebühren oder Wechselkursschwankungen kann der Kontostand trotz Kompensation nicht in den Zustand versetzt werden, den er vor der Bestellung hatte.

Der weitgehende Verzicht auf Sperren hat gewisse Nachteile. Es kann nicht sichergestellt werden, dass eine *activity* Ressourcen exklusiv nutzt. Gelesene Daten können jederzeit von anderen Teilnehmern überschrieben werden – es wird gewissermaßen auf Konsistenzebene 0 gearbeitet. Kritische Operationen sollten daher als *atomic transaction* implementiert werden, die Teil einer *business activity* sein können.

Die Resultate eines *task* sind im Zeitraum zwischen dessen Ende und der korrespondierenden Kompensation sichtbar und können die Arbeit anderer *activities* beeinflussen. In WS-BusinessActivity ist jedoch keine Möglichkeit vorgesehen, diese abhängigen *activities* ebenfalls zu kompensieren. Dies würde gegebenenfalls zu komplexen Kompensationskaskaden führen und trotzdem nicht garantieren, dass das System in den Zustand vor Ausführung der ersten *activity* gebracht werden kann.

Kompensation ist ein spezieller *task* und kann als solcher fehlschlagen. Manuelle Recovery wird nötig, falls dies (regelmäßig) geschieht.

Den Teilnehmern einer *business activity* ist es möglich, diese jederzeit zu verlassen, auch vor ihrer Beendigung. Die Teilnehmerliste ist dynamisch.

Im Gegensatz zu *WS-AtomicTransaction* schreibt *WS-BusinessActivity* kein konkretes Koordinatorverhalten vor. Je nach Aufgabe der *activity*, kann die benötigte Koordination recht verschieden ausfallen. Anwendungslogik wird in die Koordination einbezogen, um auf die Resultate von *child scopes* oder Exceptions angemessen zu reagieren. Der Anwendungszustand hat somit Auswirkungen auf die Koordination der *tasks*. Diese enge Kopplung zwischen Applikation und Koordinator legt eine Integration beider in einem einzelnen Dienst nah. Eine getrennte Realisierung würde ein zusätzliches applikationsspezifisches Protokoll zur Kommunikation zwischen Anwendung und Koordinator nötig machen. *WS-BusinessActivity* schlägt die Ausnutzung der Erweiterbarkeit des *business activity coordination type* vor, um auf dieser Basis zusätzliche

Protokolle zu entwickeln, die das Koordinatorverhalten konkret vorschreiben.

Das `Expires`-Element des *coordination context* stellt ein TimeOut für die *business activity* dar. Nach Ablauf des TimeOut ist es jedem Teilnehmer gestattet, sich eigenmächtig aus der *activity* zurückzuziehen. Der *expires*-Wert sollte geeignet gewählt werden, um nur beim Auftreten eines schweren Fehlers überschritten zu werden.

1.1.3.1 BusinessAgreementWithParticipantCompletion

Ein Teilnehmer registriert sich für dieses Protokoll, falls er in der Lage ist, eigenständig zu entscheiden, wann seine Aufgabe im Rahmen einer *activity* beendet ist. Eine Protokollinstanz besteht in der Regel nur zwischen dem Koordinator und einem einzelnen Web Service. Abbildung 7 zeigt die Zustände, die von der Koordinationsbeziehung zwischen den Partnern angenommen werden können.

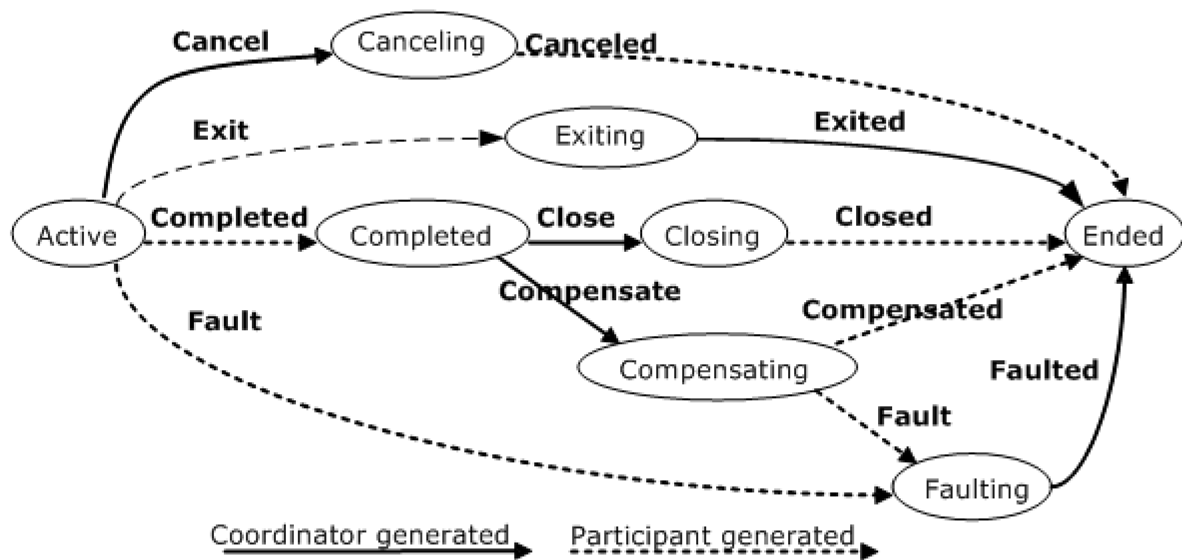

Abbildung 7: Zustandsdiagramm BusinessAgreementWithParticipantCompletion

Wenn der Teilnehmer seine Arbeit erfolgreich beendet hat, teilt er dies dem Koordinator mittels einer **Completed** Nachricht mit. Durch senden von **Exit** kann eine Protokollinstanz vorzeitig abgebaut werden. **Fault** signalisiert einen Fehler während der Verarbeitung des *scope* bzw. während dessen Kompensation. Die Nachrichten **Exited** und **Faulted** werden vom Koordinator verschickt, um zu bestätigen, dass vom Teilnehmer keine weiteren Aktionen erwartet werden, Fehlerbehandlung ist Sache des Koordinators.

Nach Erhalt von **Completed** weiß der Koordinator, dass der *scope* seine Arbeit erfolgreich ausgeführt hat. Zu diesem Zeitpunkt hat er die Wahl, die Protokollinstanz mittels **Close** erfolgreich zu beenden oder durch **Compensate** zurückzusetzen. Diese Entscheidung wird in der Regel durch Anwendungslogik getroffen und kann vom Resultat anderer *tasks* abhängig sein. Der Teilnehmer antwortet mit **Closed** bzw. **Compensated**.

Die Arbeit eines *scope* kann mittels **Cancel** schon vorzeitig abgebrochen werden. Der Teilnehmer antwortet mit **Canceled**.

Im **Active-**Zustand kann ein Zustandsübergang sowohl durch den Teilnehmer als auch durch den Koordinator ausgelöst werden. Da Nachrichten eine gewisse Laufzeit haben, kann es zu einer Race-Situation kommen. Zum Beispiel könnte der Koordinator **Cancel** senden und in den Zustand **Canceling** wechseln, während der Teilnehmer **Completed** sendet und seinerseits in Zustand

Completed übergeht. In diesem Fall ist es in der Verantwortung des Koordinators, die Protokollinstanz in einen konsistenten Zustand zurückzuversetzen. Dazu geht er in seinen vorherigen Zustand zurück und setzt mit der Verarbeitung der Nachricht des Teilnehmers fort, wie im Protokoll vorgesehen. Der Teilnehmer ignoriert Nachrichten, die er in seinem aktuellen Zustand nicht erwartet.

1.1.3.2 BusinessAgreementWithCoordinatorCompletion

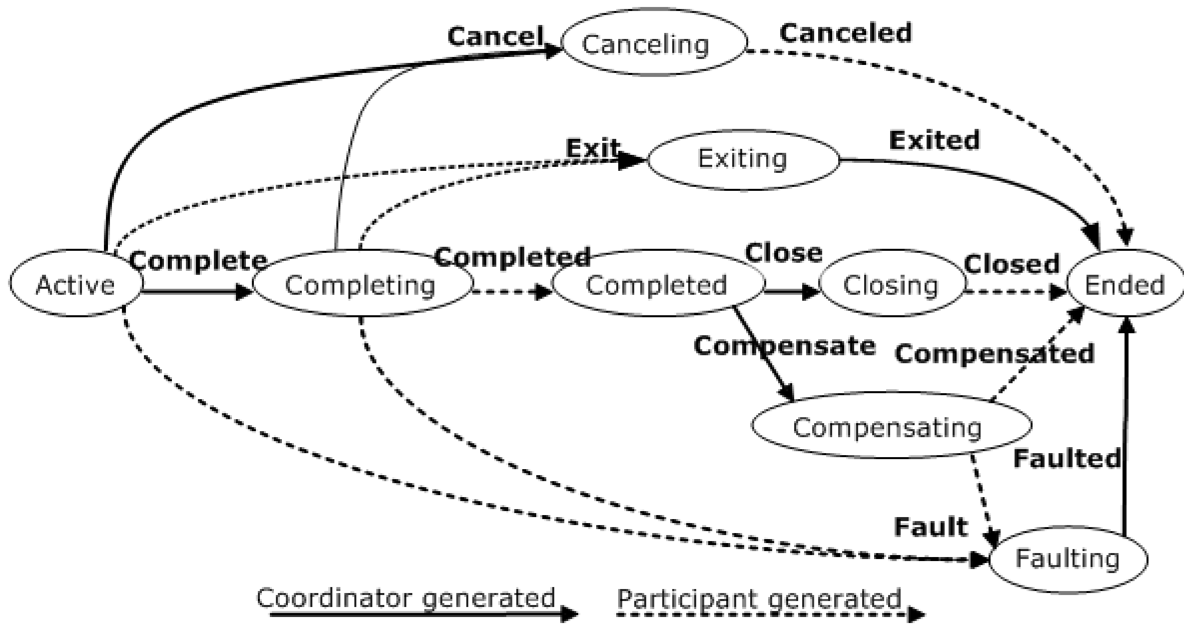

Abbildung 8: Zustandsdiagramm BusinessAgreementWithCoordinatorCompletion

BusinessAgreementWithCoordinatorCompletion ist *BusinessAgreementWithParticipantCompletion* sehr ähnlich. Der einzige Unterschied besteht darin, dass dem Teilnehmer mittels einer **Complete-**Nachricht signalisiert wird, dass dieser keine weiteren Anfragen im Verlauf einer *business activity* zu erwarten hat. Die Entscheidung, dass die Dienste eines Teilnehmers nicht länger benötigt werden, muss von Anwendungslogik getroffen und gegebenenfalls dem Koordinator mitgeteilt werden, falls er getrennt von der Applikation implementiert ist.

1.1.3.3 Abschließendes Beispiel

Das folgende Beispiel in Abbildung 9 zeigt den Ablauf einer Flugbuchung, die als *business activity* realisiert ist. Dazu erzeugt eine Applikation mit integriertem Koordinator einen entsprechenden *coordination context* und nutzt gleichzeitig mehrere Web Services, um einen möglichst günstigen Flug zu buchen. Nachdem die Web Services die Nachrichten erhalten haben, verwenden sie die Informationen aus dem *coordination context*, um sich für die *business activity* zu registrieren. Alle drei entscheiden sich für das *BusinessAgreementWithParticipantCompletion*-Protokoll.

Airline A kann kein Angebot vorlegen und signalisiert dies durch Senden von **Fault**. Die Applikation bestätigt den Erhalt durch **Faulted** und beendet die Protokollinstanz.

Die Airlines B und C wickeln die Buchung ab und teilen die entstandenen Kosten mittels einer **Completed-**Nachricht mit. Da B das günstigere Angebot gemacht hat, entscheidet sich die Applikation, die Buchung bei C wieder zu stornieren und sendet **Compensate**. Um die *activity* zu beenden, sendet die Applikation **Close** an Airline B.

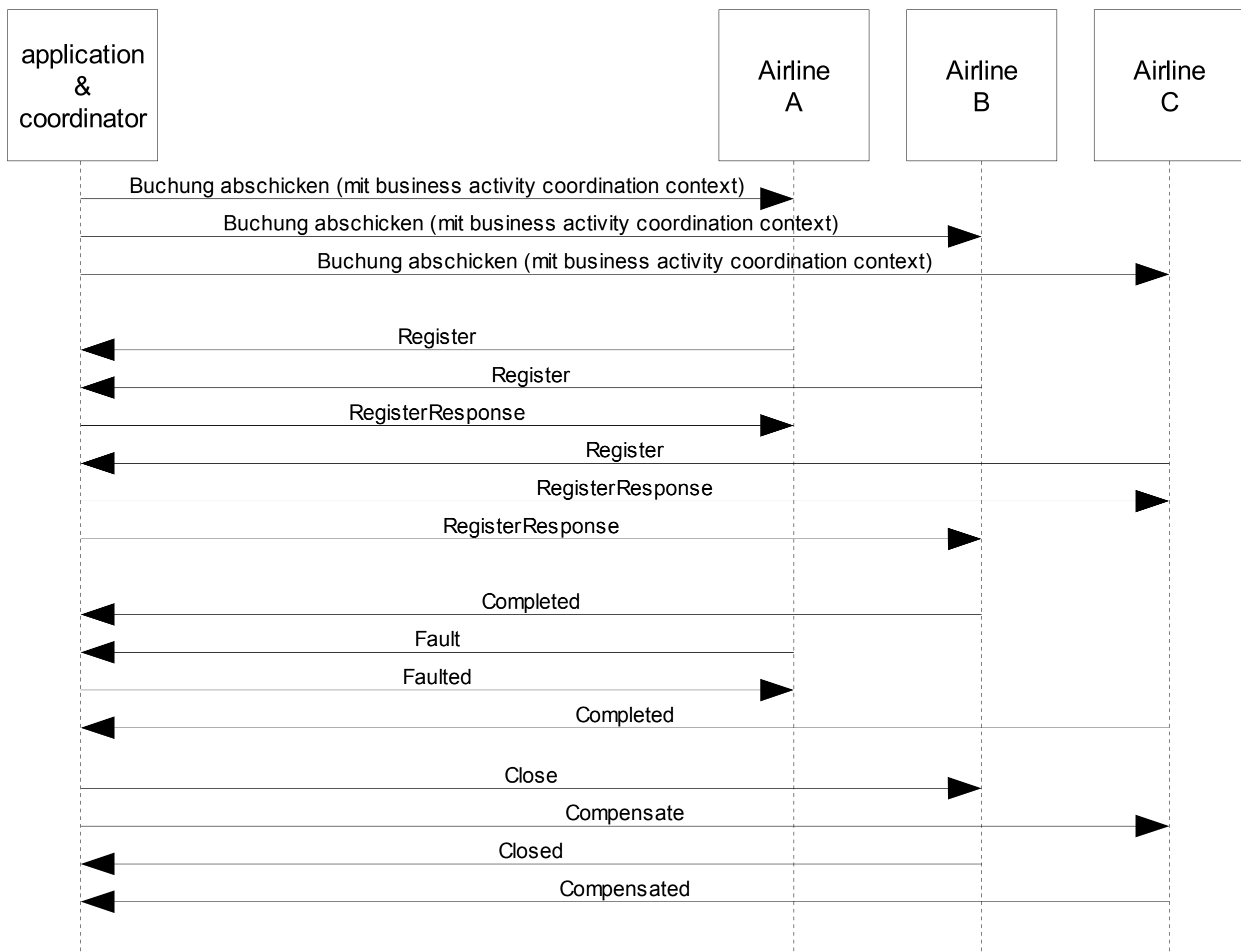

Abbildung 9: Beispielhafter Ablauf einer business activity

1.2 Business Transaction Protocol

Die aktuelle Version 1.0 des *Business Transaction Protocol (BTP)* [6] wurde im Juni 2002 vom OASIS Committee veröffentlicht. Die Spezifikation wurde parallel mit *WS-Coordination* und *WS-Transaction*, unter Teilnahme von Vertretern der Firmen Oracle, Sun Microsystems, Iona, Hewlett-Packard, BEA und anderen entwickelt.

BTP konzentriert sich auf die konzeptionelle Ebene und vermeidet bewusst Abhängigkeiten von anderen Standards. Implementierungsentscheidungen sollen keinen vermeidbaren Beschränkungen unterworfen werden. BTP wurde zur Anwendung in einer Web Service Umgebung entwickelt, lässt sich jedoch auch mit anderen Protokollen als SOAP kombinieren. Anders als *WS-Coordination* baut es nicht unmittelbar auf bestehenden Web-Service-Standards auf.

BTP basiert auf dem 2PC-Protokoll und erweitert es, um die Abwicklung von geschäftlichen Transaktionen zwischen verschiedenen Gesellschaften zu unterstützen. Konsistente Zustandsübergänge einer verteilten Geschäftsverbindung werden ermöglicht. 2PC-Protokolle können zu einer Blockierung der Transaktion führen, falls sich nicht alle Teilnehmer kooperativ verhalten. Daher geht BTP davon aus, dass alle am Koordinationsprozess beteiligten Softwarekomponenten innerhalb einer *trust domain* liegen und nach bestem Wissen implementiert wurden. Ob diese Annahme für den Einsatz des Protokolls im Internet realistisch ist, darf bezweifelt werden. Überlegungen zum Thema Sicherheit kommen in der Spezifikation nicht vor.

1.2.1 Business transactions

Eine *business transaction* wird als konsistenter Zustandsübergang in einer Geschäftsbeziehung zwischen zwei oder mehreren Gesellschaften definiert. Der Zustand wird notwendigerweise verteilt gespeichert. Die teilnehmenden Systeme verfügen über Anwendungen, die Zustandsinformationen speichern, modifizieren und geeignet kommunizieren. Durch Teilnahme an einer *business transaction* werden diese Operationen koordiniert, um Inkonsistenzen des Zustands der Geschäftsbeziehung zu vermeiden. Jede BTP-Anwendung lässt sich in ein *application element* und ein *BTP element* unterteilen:

- Das *application element* ist für die Abwicklung von geschäftlichen Operationen zuständig, wie zum Beispiel die Durchführung einer Bestellung oder die Reservierung eines Sitzplatzes. Dazu werden applikationsspezifische Nachrichten ausgetauscht, die nicht Gegenstand der BTP-Spezifikation sind.
- Das *BTP element* ermöglicht die Koordination der verteilten Anwendungen. Basierend auf dem klassischen 2PC-Protokoll werden Nachrichten spezifiziert, die unter den Teilnehmern einer *business transaction* ausgetauscht werden. Das *BTP element* ist in der Lage, diese Nachrichten zu senden, zu empfangen und geeignete Änderungen des lokalen Zustand zu veranlassen, um einen konsistenten Übergang des verteilten Gesamtzustandes zu erreichen.

1.2.2 Rollen in einer business transaction

Eine Anwendung, die sich an einer *business transaction* beteiligen kann, das heißt über ein *BTP element* verfügt und eindeutig adressiert werden kann, wird als *actor* bezeichnet. BTP definiert verschiedene *roles* (nachfolgend als Rollen bezeichnet), die ein *actor* im Rahmen einer *business transaction* ausfüllen kann. Dabei können mehrere Rollen gleichzeitig durch einen einzelnen *actor* gespielt werden.

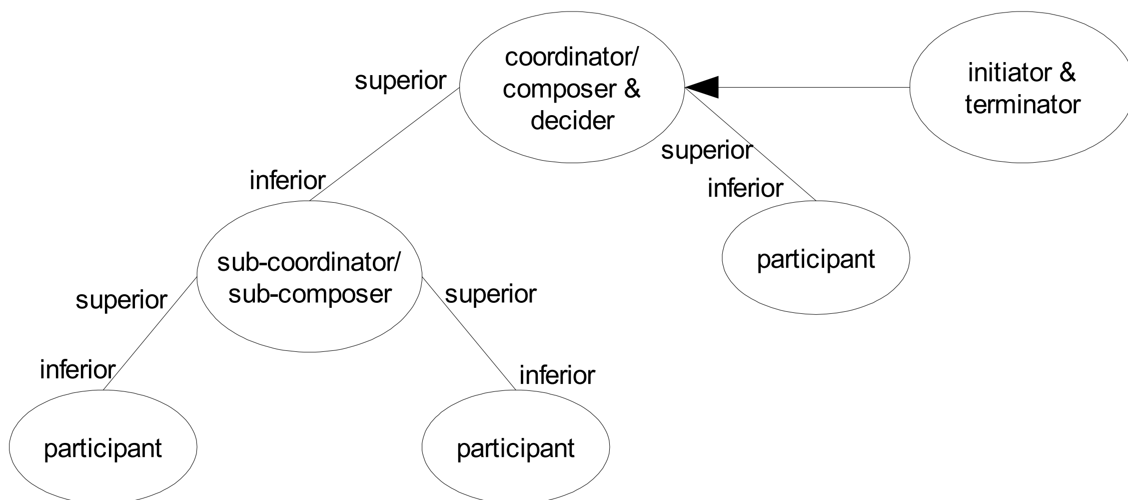

Abbildung 10: Rollen in einer business transaction

Eine Anwendung, die eine neue *business transaction* startet wird als *initiator* bezeichnet. Dazu sendet diese eine BEGIN-Nachricht an einen Dienst der *factory* genannt wird, um einen *coordinator* bzw. einen *composer* (der Unterschied zwischen diesen Rollen wird später in diesem Abschnitt beschrieben) und einen Kontext zu erzeugen. Die *factory* liefert den Kontext mittels einer BEGUN-Nachricht zurück. Dieser ist in Aufbau und Nutzung dem *coordination context*, der aus *WS-*

Coordination bekannt ist, sehr ähnlich. Er enthält Informationen über den Typ der *business transaction* und eine Referenz auf den *coordinator* bzw. *composer*.

In der Regel wird eine *business transaction* von der gleichen Anwendung beendet, die sich auch gestartet hat. Trotzdem wird in BTP die Rolle des *terminator* von der des *initiator* unterschieden, da die Kontrolle über die Beendigung einer *business transaction* weitergereicht werden kann. Der *terminator* hat die Wahl die Transaktion entweder erfolgreich zu beenden, falls dies möglich ist, oder diese zurückzusetzen. Die erste Möglichkeit wird in BTP „Confirm" genannt und entspricht dem Commit des klassischen Transaktionsparadigmas. Um die Zustandsänderungen einer *business transaction* ungeschehen zu machen, kann der *terminator* sich für „Cancel" entscheiden, was mit Rollback vergleichbar ist.

In der Beziehung zwischen einer koordinierenden und einer abhängigen Anwendung, wird erstere als *superior* und letztere als *inferior* bezeichnet. Die Protokollinstanz zwischen diesen Teilnehmern wird *superior:inferior relationship* genannt. Ein *actor* kann gleichzeitig die Rolle eines *superior* und eines *inferior* spielen, wodurch eine Hierarchie beliebiger Tiefe und Breite entstehen kann, die als *business transaction tree* bezeichnet wird.

Der Höchste *superior* in der Hierarchie des *business transaction tree* hat die Rolle des *decider* inne. Er sammelt die Prepare-Nachrichten, die im Rahmen des 2PC-Protokolls von allen Beteiligten an ihren *superior* weitergereicht werden, wertet diese aus und trifft die Entscheidung, ob die *business transaction* erfolgreich beendet werden kann oder zurückgesetzt werden muss.

Ein *superior* kann anhand von zweierlei Merkmalen unterschieden werden. Das erste Kriterium ist, ob er selbst auch ein *inferior* und damit nicht der Höchste in der Hierarchie ist. Das zweite, wie er sich gegenüber seinen *inferiors* verhält. Hier gibt es zwei Möglichkeiten: Er kann sich wie ein Koordinator im klassischen 2PC-Protokoll verhalten und ein Confirm nur dann gestatten, wenn alle *inferiors* dazu in der Lage sind. Oder er kann einer bestimmten Teilmenge der *inferiors* Confirm befehlen, den anderen Cancel und die Transaktion trotzdem erfolgreich beenden. Dadurch ergeben sich vier verschiedene *superior* Typen:

- Ein (*atom) coordinator* wird beim Start einer *business transaction* geschaffen und ist der Höchste in der Hierarchie des *business transaction tree*. Er sorgt dafür, dass entweder alle *inferiors* mit Confirm oder alle mit Cancel beendet werden. Diese Entscheidung liegt letztlich beim *terminator*.
- Ein (*cohesion) composer* wurde ebenfalls beim Beginn einer *business transaction* erschaffen. Der *terminator* hat jedoch das Recht, sich für eine Teilmenge der *inferiors* zu entscheiden, die schließlich erfolgreich beendet werden sollen, falls dies möglich ist. Diese Teilmenge wird als *confirm set* bezeichnet. Alle nicht im *confirm set* enthaltenen *inferiors* werden zum Cancel aufgefordert.
- Ein *(atomic) sub-coordinator* wurde in eine bestehende *business transaction* einbezogen. Er sorgt dafür, dass entweder alle *inferiors* zum Confirm oder alle zum Cancel aufgefordert werden. Die Entscheidung für eine dieser Möglichkeiten wird von seinem übergeordneten Koordinator getroffen.
- Ein *(cohesive) sub-composer* wurde in eine bestehende *business transaction* einbezogen. Die Entscheidung welche *inferiors* das *confirm set* enthalten soll, wird bis zum Beginn der Prepare-Phase verzögert. Die Kandidaten für das *confirm set* werden durch Anwendungslogik des *sub-composer* ausgewählt.

Eine business transaction, deren *decider* ein *atom coordinator* ist, wird als *atomic business transaction* bzw. als *atom* bezeichnet. Eine business transaction, deren *decider* ein *cohesion composer* ist, wird als *cohesive business transaction* bzw. als *cohesion* bezeichnet.

Eine *cohesion* hat große Ähnlichkeit mit einer *business activity*, die in Abschnitt 1.1.3 beschrieben wurde. Im Gegensatz zum *atom* wird bei einer *atomic transaction* (Abschnitt 1.1.2) die Isolationseigenschaft gefordert. Auf diesen Aspekt wird in Abschnitt 1.2.3 detailliert eingegangen.

Ein Teilnehmer an einer *business transaction*, der selbst kein *superior* ist, hat die Rolle eines *participant* inne.

Nachdem der *initiator* eine *business transaction* begonnen hat und einen Kontext sowie einen *decider* (der ein *coordinator* oder ein *composer* sein kann) erzeugt hat, kann er weitere *actors* einbeziehen, indem er den Kontext an diese weitergibt. Ob dieser als Header in applikationsspezifischen Nachrichten integriert wird oder als getrennte Nachricht gesendet wird, ist in der BTP-Spezifikationen nicht festgelegt. Der Kontext enthält die notwendigen Informationen, mit denen der neue Teilnehmer die *inferior:superior*-Beziehung eingehen kann. Dieser Vorgang ist dem Registrieren, das in der *WS-Coordination*-Spezifikation beschrieben wird, sehr ähnlich. Hier wird dieser jedoch als *enrol* bezeichnet. In BTP ist für diesen Vorgang eine eigene Rolle vorgesehen, die *Enroller* genannt wird. Dieser Dienst hat die Aufgabe den *superior* über die Existenz des *inferiors* zu informieren. Der *inferior* kann nun seinerseits weitere Dienste in seine Arbeit miteinbeziehen und dabei selbst die Rolle des *superior* inne haben. Mit einer CONTEXT_REPLY-Nachricht wird einem *superior* signalisiert, dass der Sender von nun an keine weiteren *inferiors* in die *business transaction* einbeziehen wird.

1.2.3 Zwei-Phasen-Commit im BTP

Eine *business transaction* garantiert einen konsistenten Zustandsübergang verteilter Systeme. Jeder Teilnehmer muss die Änderungen am lokalen Zustand im Falle eines Scheitern der Transaktion wieder rückgängig machen können. Zustandsänderungen werden von Nachrichten initiiert, die von den *application elements* der *actors* ausgetauscht werden. Deren Auswirkungen dürfen zunächst nur vorläufiger Natur sein; man spricht von einem *provisional effect*. Beim erfolgreichen Ende der Transaktion kann der vorläufige Zustand durch einen *final effect* dauerhaft gemacht werden. Ein *counter effect* ist in der Lage, den alten Zustand wiederherzustellen und wird im Falle des Scheiterns einer Transaktion genutzt.

Die BTP-Spezifikation überlässt es der jeweiligen Implementierung, wie diese Effekte realisiert werden. Es ist sowohl der Einsatz von Sperrverfahren, before- oder after-images sowie Kompensationsverfahren möglich. Insbesondere wird hier <u>nicht</u> die Isolationseigenschaft gefordert, wie sie von klassischen Transaktionen bekannt ist. Der sichtbare Zustand nach dem *provisional effect* kann dem Zustand nach dem *counter* oder *final effect* entsprechen, oder gar von beiden abweichen. Es besteht die Möglichkeit, dass Transaktionen ohne Wissen, Daten in ihre Verarbeitung miteinbeziehen, die in solch einem schwebenden Zustand sind. Diese abhängigen Transaktionen werden nicht zurückgesetzt, falls die gelesenen Daten beispielsweise durch einen *counter effect* ungültig werden. Das würde potentiell zu komplexen Cancel-Kaskaden führen.

Wie erwähnt werden die *provisional effects* durch den Austausch von applikationsspezifischen Nachrichten zwischen den *application elements* der Beteiligten einer *business transaction* hervorgerufen. Um die Transaktion konsistent zu beenden, wird nun in einer 2PC-Entscheidung zwischen den *BTP elements* der Beteiligten festgelegt, welche der *actors* den *final effect* und welche den *counter effect* anwenden sollen. Das Verhalten eines *actor* während des 2PC-Prozesses hängt von der Rolle ab, die er inne hat.

Um die Transaktion zurückzusetzen sendet der *terminator* CANCEL_TRANSACTION an den *decider*. Nach Erhalt dieser Nachricht sendet der *decider* CANCEL an alle seine *inferiors*. Falls es sich bei diesen gleichzeitig auch um *superiors* handelt, geben sie die CANCEL-Nachricht wiederum an ihre *inferiors* weiter. So wird schließlich jeder *actor* im *business transaction tree* informiert und kann einen *counter effect* zur Anwendung bringen, falls er seinen lokalen Zustand im Verlauf der

business transaction geändert hat.

1.2.3.1 Prepare phase

Der *terminator* beginnt die Prepare-Phase, indem er eine CONFIRM_TRANSACTION-Nachricht an den *decider* sendet. Handelt es sich bei der *business transaction* um eine *cohesion*, hat der *terminator* die Möglichkeit, die Prepare-Phase für einzelne *inferiors* gesondert zu beginnen und vom *composer* über das Ergebins informiert zu werden (BTP spezifiziert hierfür die PREPARE_INFERIORS- und INFERIORS_STATUS-Nachricht). Die Zusammensetzung des *confirm set* kann durch das Resultat dieser vorgezogenen Prepare-Phase beeinflusst werden. Wie diese Informationen genutzt werden, wird jedoch von Anwendungslogik bestimmt.

Entscheidet sich der *terminator* für ein erfolgreiches Ende der Transaktion, sendet er CONFIRM_TRANSACTION an den *decider*. Falls es sich bei der *business transaction* um eine *cohesion* handelt, muss sich der *terminator* im Rahmen dieser Nachricht auf ein *confirm set* festlegen. Im Fall eines *atom* besteht das *confirm set* stets aus allen *inferiors* des *coordinator*. Der *decider* beginnt die Prepare-Phase, indem er PREPARE an alle seine *inferiors* propagiert. Jeder *actor* im *business transaction tree,* der die Aufforderung zum PREPARE erhält, verhält sich entsprechend seiner Rolle:

- Ein *sub-coordinator* sendet PREPARE an alle seine *inferiors*. Nur wenn er von <u>allen</u> eine PREPARED-Nachricht zurückerhält, antwortet er seinem *superior* ebenfalls mit PREPARED. Falls der *sub-coordinator* eine CANCELLED-Nachricht von einem *inferior* erhält oder beim Auftreten eines Time Out, befiehlt er allen *inferiors* die Transaktion zurückzusetzen (durch senden von CANCEL-Nachrichten) und meldet CANCELLED an seinen *superior* weiter.
- Ein *sub-composer* sendet PREPARE an alle seine *inferiors*. Das *confirm set* wird durch Anwendungslogik des *sub-composer* determiniert, der *terminator* hat keinen Einfluss darauf. Falls der *sub-composer* von allen im *confirm set* enthaltenen *actors* eine PREPARED-Nachricht erhält, sendet dieser ebenfalls PREPARED an seinen *superior*. Ist dies nicht der Fall, kann das den *sub-composer* kontrollierende *application element* eingreifen und neue *inferiors* einbeziehen, vorhandene zurücksetzen oder die Verarbeitung trotz des Fehlers fortsetzen – die Möglichkeiten sind vielfältig. Entscheidet sich das *application element* jedoch endgültig, die Transaktion abzubrechen, informiert es alle *inferiors* durch senden von CANCEL-Nachrichten und propagiert seinerseits CANCELLED an den eigenen *superior*.
- Ein *participant*, aber auch jeder andere *actor*, der seinen Zustand im Verlauf der *business transaction* verändert hat, überprüft nach Erhalt einer PREPARE-Nachricht, ob er in der Lage ist, sowohl den *final* als auch den *counter effect* zu erzielen. Nur in diesem Fall antwortet er mit PREPARED, ansonsten mit CANCELLED.

Falls ein *inferior* über Anwendungslogik verfügt, die in der Lage ist festzustellen, dass dessen Arbeit im Rahmen der *business transaction* getan ist, kann er PREPARED auch ohne entsprechende Aufforderung an seinen *superior* melden. Dieser Vorgang wird als *spontaneous prepare* bezeichnet. Auch ein *sub-coordinator* oder *sub-composer* kann spontan PREPARED melden, falls das gesamte *confirm set* das bereits getan hat.

Wenn ein *coordinator* oder ein *composer* nur einen einzigen *inferior* hat, stellt das 2PC-Protokoll einen unnötigen Aufwand dar. In diesem Fall kann der *superior* die Entscheidung über den Ausgang der Transaktion an seinen *inferior* abtreten, indem er CONFIRM_ONE_PHASE sendet und auf dessen Antwort wartet.

Sobald alle PREPARED bzw. CANCELLED-Nachrichten beim *decider* eingetroffen sind, ist dieser in der Lage, über das Schicksal der Transaktion zu entscheiden und kann gegebenenfalls die Confirm-Phase beginnen oder die Transaktion abbrechen. In beiden Fällen wird der *terminator* über

das Resultat der Transaktion informiert.

1.2.3.2 Confirm phase

Hat das gesamte *confirm set* des *decider* mit PREPARED geantwortet, so kann die Transaktion erfolgreich beendet werden. Falls es sich bei der *business transaction* um eine *cohesion* handelt, sendet der *composer* CONFIRM-Nachrichten an alle *inferiors* im *confirm set* (das vom *terminator* determiniert wird), alle anderen erhalten CANCEL. Handelt es sich um ein *atom*, erhalten alle *inferiors* eine CONFIRM-Nachricht vom *coordinator*.

Jeder superior im *business transaction tree* reicht das CONFIRM an sein *confirm set* weiter und löst damit gegebenenfalls *final effects* aus. Die korrespondierenden CONFIRMED-Nachrichten werden dann in Gegenrichtung von den Blättern des Baumes bis zum *decider* gesendet, woraufhin dieser den *terminator* über den Ausgang der Transaktion informiert.

1.2.4 Abschließendes Beispiel

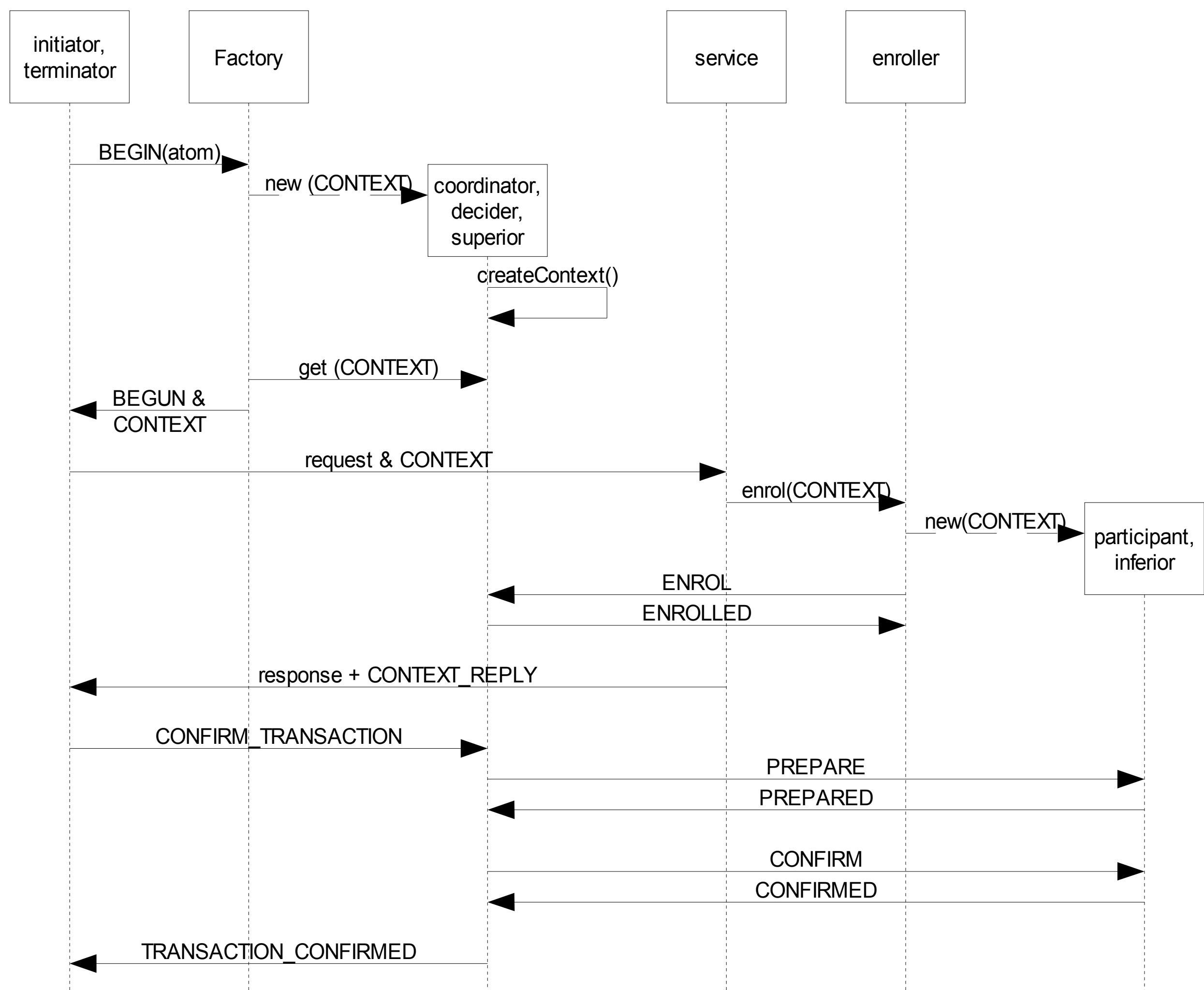

Abbildung 11: einfache business transaction

Abbildung 11 zeigt den Nachrichtenfluss einer einfachen *business transaction*, die nur eine *superior:inferior*-Beziehung enthält. Die Kommunikation findet zwischen zwei *application elements* statt; das erste vereinigt die Rollen des *initiator* und *terminator* in sich, das zweite ist der

genutzte (Web) Service kombiniert mit einem *enroller*. Alle applikationsspezifischen Nachrichten, die nicht in BTP spezifiziert sind, sind in diesem Beispiel durch Kleinschreibung kenntlich gemacht und könnten auch in anderer Form erfolgen bzw. wären nicht nötig, falls mehrere Rollen in eine Applikation integriert würden. Die ebenfalls nicht standardisierten Interaktionen von *participant* und *service* tauchen in diesem Beispiel nicht auf, wären jedoch in der Praxis notwendigerweise vorhanden.

2. Transaktionen in Grid-Umgebungen

Die Forschung auf dem Gebiet des Grid Computing schritt in den letzten Jahren schnell voran, das Thema Transaktionen spielte dabei eine eher untergeordnete Rolle. Die bisherigen Bemühungen konzentrierten sich vor allem auf die Integration von Servern, Datenhaltungssystemen und Netzwerken in einen virtuellen Supercomputer – dem Grid. In diesem können Systeme zusammenarbeiten, die über den gesamten Globus verteilt sind und unterschiedliche Hardware-Plattformen oder Betriebssystemen verwenden. Die Enorme ungenutzte Rechenleistung vieler Systeme (z.B. nutzten Desktop-Computer durchschnittlich nur 5% ihrer Rechenkapazität) kann produktiv eingesetzt werden, indem Prozesse von überlasteten System auf andere migrieren. Das Projekt *grid.org* konzentriert sich ausschließlich auf dem Aufbau eines solchen weltweiten Rechenlastverbundes via Internet. Transaktionen spielen hier keine Rolle.

Es gibt Ansätze, die auf die Überwindung von Heterogenität mittels Grid Computing auch im Bereich der Datenhaltung abzielen. Ein so genanntes Data Grid bietet eine einheitliche, plattformunabhängige Schnittstelle für verteilte Speichersysteme. Die *Database Access and Integration Services Working Group (DAIS)* des *Global Grid Forum (GGF)* beschäftigt sich mit der Integration von bestehenden Datenbanken in Grid-Systeme [7]. Bisher werden die Datenbanken jedoch als reine Informationsspeicher verstanden, die Möglichkeit verteilter Transaktionen wird nicht genutzt. Zu diesem Aspekt wurden von der DAIS bis jetzt nur grundlegende Überlegungen angestellt. Es wurden Probleme identifiziert, die die Übertragung des klassischen Transaktionsparadigmas in die Welt der Grids birgt, um im nächsten Schritt ein angepasstes Transaktionskonzept für Grid-Umgebungen zu entwickeln. Diese Probleme sind denen sehr ähnlich, die bei der Übertragung von Transaktionen in eine Web-Service-Umgebungen auftreten. Auch bei einem Grid handelt es sich um einen losen, weitläufig verteilten Verbund von Systemen, die unter Kontrolle verschiedener Gesellschaften stehen können. Klassische Transaktionen sind kurzlebig und nutzen Ressourcen in der Regel exklusiv. Wie bei Web Services wären Transaktionen im Grid von längerer Dauer und der Einsatz von Sperrverfahren sehr problematisch. Die DAIS kommt daher zu dem Ergebnis, dass ACID-Transaktionen nicht für eine Grid-Umgebung geeignet sind und neue Konzepte erdacht werden müssen. In diese Überlegungen sollen *WS-Coordination* und *WS-BusinessActivity* einbezogen werden.

In jüngster Zeit zeichnet sich ein aufkommendes Interesse für Transaktionen in Grid-Umgebungen ab:

- Das GGF zieht die Gründung einer neuen Forschungsgruppe in Erwägung, der *Grid Transaction Research Group.* Sie wird die erste Arbeitsgruppe des GGF sein, die sich mit Transaktionen im Allgemeinen beschäftigt. Bisher wurden Forschungen in dieser Richtung, wenn überhaupt, von jeder Arbeitsgruppe gesondert betrieben, vor dem Hintergrund spezifischer Problemstellungen.
  Zu den Aufgaben der *Grid Transaction RG* wird die Entwicklung von Transaktionskonzepten gehören, die der Grid-Umgebung Rechnung tragen. Es ist vorgesehen zu prüfen, inwiefern Aspekte bestehender Spezifikationen aus dem Bereich der Web Services, wie beispielsweise *WS-Coordination*, übernommen werden können.

- Auf einer Tagung der GGF *Area on Data Management (Data Area)* am 6. Oktober 2003 in Chicago wurden die größten Herausforderungen aufgelistet, die sich der Data Area zukünftig stellen werden [8]. Die Liste enthält unter anderem die Punkte:
    - Integration von Transaktionsverarbeitung in Grids (als kurzfristiges Ziel)
    - Klärung der Beziehung von Web-Service-Spezifikationen zur Transaktionsverarbeitung zu Grid-Applikationen.
- In einem Projekt der Shanghai Jiao Tong University wurde ein System names GridTP erarbeitet, das Transaktionsverarbeitung in Grids ermöglicht [9]. GridTP ist unabhängig von Web-Service-Standards, es bedient sich stattdessen des X/Open DTP Models und integriert es in eine Grid-Umgebung. X/Open DTP basiert auf dem 2PC-Konzept. Es werden folgende Schnittstellen für verteilte Transaktionsverarbeitung spezifiziert: Die TX API dient der Steuerung des *Transaction Managers (TM)* (entspricht dem Koordinator) durch eine Applikation, das XA-Interface wird zur Kommunikation des TM mit den *Resource Managers (RM)* (den Agenten) genutzt (siehe Abbildung 12). X/Open DTP wird bereits von vielen bestehenden Datenbanksystemen unterstützt.

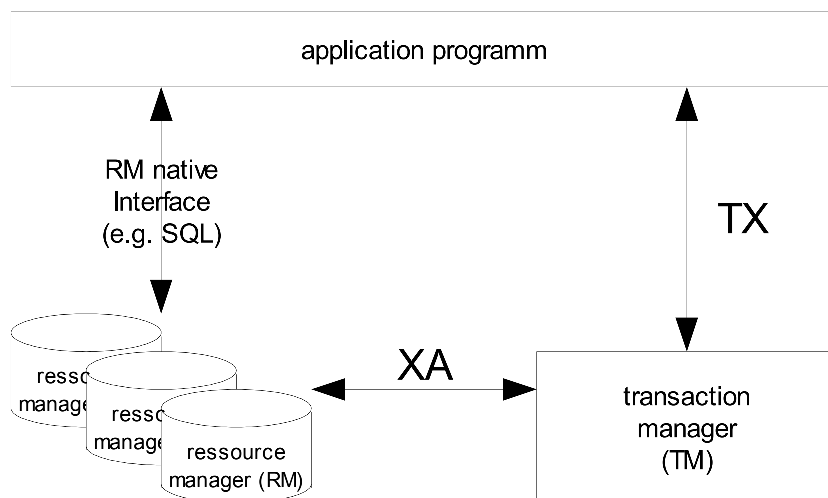

Abbildung 12: Komponenten und Interfaces des X/Open DTP Modells

GridTP ist eine Middlewarekomponente, die referentielle Integrität zwischen unterschiedlichen Datenbanksystemen sicherstellt und verteilte Transaktionsverarbeitung ermöglicht. Allerdings wird das klassische Transaktionskonzept ohne jegliche Änderung zur Anwendung gebracht. Dieses passt, wie bereits erwähnt, nicht optimal in eine Grid-Umgebung. Diese Tatsache ist auch den Entwicklern bewusst – sie haben sich das Ziel gesetzt GridTP geeignet weiterzuentwickeln, um zukünftig Unterstützung für bestehende Web-Service-Standards wie *WS-Transaction* zu bieten.

Die Forschungen zur Integration von Transaktionen in Grid-Systeme stehen momentan an ihrem Anfang. Welche Standards sich hier in Zukunft durchsetzten werden ist noch nicht abzusehen. Interessant ist, dass man sich ähnlichen, wenn nicht gar den gleichen Problemen gegenüber gestellt sieht, die bereits aus dem Bereich der Web Services bekannt sind. Anleihen aus bestehenden Spezifikationen wie WS-*Coordination* oder WS-*BusinessActivity* werden eventuell die Basis für eine Lösung bilden. Möglicherweise können diese Protokolle sogar ohne Änderungen in Grids zur Anwendung kommen, da durch die *Open Grid Services Architecture (OGSA)* der Brückenschlag

zwischen der Grid- und Web-Service-Welt gelungen ist. OGSA ermöglicht die Nutzung von WSDL, um so genannte *grid services* zu definieren, die somit wie herkömmliche Web Services genutzt werden können.

Zusammenfassung

Web Services sind Dienste, die von verschiedenen Gesellschaften im Internet angeboten werden und vollautomatisch genutzt werden können. Bestehende ERP- und Workflow-Systeme können durch Nutzung von Web Services plattformübergreifend, über Unternehmensgrenzen hinweg zusammenarbeiten. Es besteht die Möglichkeit, ganze Geschäftsprozesse, wie z.B. den Einkauf, weitgehend zu automatisieren. Das Transaktionskonzept bietet ein Höchstmaß an Zuverlässigkeit und ist eine der Voraussetzungen für den Vertragsabschluss im Internet. Eine Übertragung in die Welt der Web Services wird nötig. Hierzu wurden in dieser Ausarbeitung zwei unterschiedliche Frameworks vorgestellt:

- *WS-Coordination*, *WS-AtomicTransaction* und *WS-BusinessActivity* entwickelt von BEA, IBM und Microsoft. Referenzen auf diese Spezifikationen tauchen in der Literatur am häufigsten auf, sie bilden eine Art Standardwerk.
- BTP ist eine Spezifikation des OASIS Committee, die ebenfalls die Koordination verteilter Anwendungen ermöglicht. Diese ist sehr allgemein gehalten und wurde nicht ausschließlich zur Nutzung in einer Web-Service-Umgebung geschaffen.

Keine der Spezifikationen ist bis zum heutigen Zeitpunkt standardisiert.

In beiden Frameworks werden neue Konzepte entwickelt (*business activity* bzw. *cohesion*), die dem Umstand Rechnung tragen, dass der Einsatz des klassischen Transaktionparadigmas in einer Web-Service-Umgebung erhebliche Probleme mit sich bringt. Dazu sind die Unterschiede zu einem herkömmlichen Datenbanksystem zu groß. Insbesondere die lange Laufzeit von Web-Service-Transaktionen und die große Zahl an autonomen Teilnehmern sind problematisch. Daher werden Verfahren entwickelt, die die Isolationseigenschaft der Transaktionen aufweichen und beispielsweise Kompensation statt Sperren verwenden. Anwendungslogik wird einbezogen, um einen erfolgreichen Abschluss der Transaktion zu ermöglichen, selbst wenn einige der Teilnehmer scheitern.

Im Bereich des Grid Computing spielte das Transaktionskonzept bisher eine untergeordnete Rolle. Erst seit kurzer Zeit zeichnet sich ein aufkommendes Interesse ab. Es hat den Anschein, dass die Entwicklungen aus dem Web-Services-Bereich, auch für Grid-Systeme geeignet sind und eventuell übernommen werden können. Die Forschungsaktivitäten stehen diesbezüglich jedoch noch am Anfang.

Literaturverzeichnis

[1] F. Cabrera et al.,
„Web Services Coordination (WS-Coordination)“,
September 2003,
http://msdn.microsoft.com/ws/2003/09/wscoor/

[2] F. Cabrera et al.,
„Web Services Atomic Transaction (WS-AtomicTransaction)“,
September 2003,
http://msdn.microsoft.com/ws/2003/09/wsat/

[3] F. Cabrera et al.,
„Web Services Business Activity Framework (WS-BusinessActivity)“,
January 2004
http://msdn.microsoft.com/ws/2004/01/wsba/

[4] T. Härder et al.,
„Datenbanksysteme. Konzepte und Techniken der Implementierung“
11. September 2001
Springer-Verlag Berlin Heidelberg

[5] F. Cabrera et al.,
„Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity“,
28. Januar 2004
http://msdn.microsoft.com/library/en-us/dnwebsrv/html/wsacoord.asp

[6] OASIS Committee Specification,
„Business Transaction Protocol“,
Version 1.0,
3. Juni 2002

[7] Database Access and Integration Services Working Group,
„Grid Database Access and Integration: Requirements and Functionalities“,
13. März 2003
http://www.ggf.org/documents/GWD-I-E/GFD-I.013.pdf

[8] Susan Malaika et al.,
„GGF Data Area Structure and Function Analysis“,
27. Februar 2004
http://www.gridforum.org/Meetings/ggf10/GGF10 Documents/ggfdataarea.structureandfunctionanalysis.Feb2004.pdf

[9] Zhengwei Qi et al.,
Shanghai Jiao Tong University,
„Integrating X/Open DTP into Grid Services for Grid Transaction Processing“,
Mai 2004