Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2831

Verteilte Workflow-Engine: Ausführung verteilter Prozessmodelle auf Basis von Tuplespaces

Shenqiang Wu

Studiengang: Informatik

Prüfer: Prof. Dr. Frank Leymann

Betreuer: Dipl. –Inf. Daniel Martin
Dipl. –Inf. Daniel Wutke

beginn am: 01.Mai 2008

beendet am: 31.Oktober 2008

CR-Klassifikation: C.2.4, F.1.2, H.4.1

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Geschäftsprozesse, die mit BPEL4WS[12] definiert werden, dienen der Orchestrierung von Web Services innerhalb einer Serviceorientierten Architektur auf technischer Ebene. Diese Orchestrierung ist notwendig, um die meist unterschiedlichen Implementierungen einzelner Web-Services zu einem allgemein gültigen übergeordneten Geschäftsprozess zusammenzufassen. Die Ausführung der Prozesse übernimmt dabei die sogenannte BPEL Engine (man könnte auch von der Workflow Engine sprechen). Sie ist für die Steuerung der einzelnen Prozesse auf eine zentralisierte oder eine dezentralisierte Weise verantwortlich.

Bei der dezentralisierten Orchestrierung führen mehrere Engines jeweils ein Anteil der kompletten BPEL Spezifikation an verteilten Lokationen aus. Die Engines kommunizieren direkt synchron oder asynchron miteinander ohne zentralen Koordinator, um Daten und Kontrolle zu übertragen[8]. Verteilte Ausführung kann große Performance-Gewinne mit sich bringen. z.B. Der zentralisierte Koordinator, der ein Flaschenhals oder eine einzelne Fehlerstelle werden könnte, wird entfernt. Die Verteilung des Arbeitsaufwands zwischen den einzelnen Komponenten erhöht den Durchsatz.

Aus der Idee hat sich die sog. Executable Workflow Networks (EWFN) [7], ein auf Petrinetzen[13] basiertes Modell zur Ausführung von Workflows, entwickelt um eine verteilte Workflow-Engine aufbauen zu können. Nach dem Modell teilen sich Prozesse in viele einzelne Komponenten, die sich selbst koordinieren können. Beim eigentlichen Ausführen der vordefinierten Prozesse arbeitet die Workflow-Engine alle Komponenten und auch ihre Verbindungen, also die Übergänge von einer Komponente zur nächsten, ab.

EWFNs selbst basieren auf dem Konzept der Tuplespaces [2]. Tuplespaces stammen ursprünglich aus dem Bereich der verteilten Systeme und werden dort vornehmlich zur Prozesskoordination verwendet. Nicht nur wegen der Eleganz und Einfachheit ihrer Kommunikationsprimitive(read / write /take) verbunden mit Template-Mechanismen, sondern auch wegen der natürlichen Entkopplung von Kommunikationspartnern in den Dimensionen Raum, Zeit und Referenz, sind Tuplespaces für die Koordination von Prozessinteraktionen gut geeignet.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist der Entwurf und Implementierung einer Middleware auf Basis von Tuplespaces unter Berücksichtigung von Persistenz, Transaktion und Recovery. Diese Middleware stellt der verteilten Workflow-Engine eine stabile und zuverlässige und auch asynchrone

Kommunikationsplattform bereit, dadurch ist die Engine in der Lage Service-Orchestrierungen durch das Umtauschen von Daten und Kontrollen zwischen verteilten Prozessen zu erzielen.

Die Aufgabenstellung lässt sich in zwei Teile gliedern:

- Der Entwurf der Architektur und des API des Tuplespaces, basierend auf den Anforderungen und den Operationen der EWFNs. Des Weiteren soll die Implementierung von Optimierungen wie „XPath pushdown“ und Tuple Update vorbereitet werden.

- Die Implementierung des Tuplespaces mit besonderer Berücksichtigung nicht-funktionaler Eigenschaften wie Zuverlässigkeit und Recovery. Des Weiteren sollen einige Standard Workflow Patterns [6] wie Fork, Split, Merge, Choice, Sequence usw. Beispielhaft in Form von Test-Cases Umgesetzt werden. Dieser Teil beinhaltet auch den Entwurf und die Implementierung von Test-Szenarien für die nicht-funktionalen Eigenschaften des Tuplespaces.

2 Grundlage

2.1 Workflows

2.1.1 Geschäftsprozesse und Workflows

Ein Workflow ist eine vordefinierte Abfolge von Aktivitäten, die so organisiert sind, dass sie mit Unterstützung durch IT-Systeme einen Geschäftsprozess erfüllen. Jeder Aktivität sind typischerweise eine Tätigkeit, ausführende Ressourcen (Personen, Maschinen), zu benutzende Ressourcen (Werkzeuge, Maschinen, anderweitige Betriebsmittel) und zeitliche Abhängigkeit (Reihenfolge, Ausführungsdauer usw.) zugeordnet. Workflow ist der Teil eines Geschäftsprozesses, welcher IT-gestützt durchgeführt wird.

2.1.2 Workflow Management System

Workflow Managementsysteme (WfMS) werden eingesetzt, um den Ablauf der Durchführung der einzelnen Workflow-Aktivitäten zu steuern. Dabei interagieren sie mit den an den Prozessen beteiligten Diensten und übernehmen die Interaktionen zwischen den einzelnen Aktivitäten. Workflow Managementsysteme befreien somit workflowbasierte Anwendungen von der direkten Kommunikation mit dem Betriebssystem und den verwendeten Kommunikationsmechanismen für lokale oder entfernte Serviceaufrufe.

Die Workflow Management Coalition (WfMC) ist eine Vereinigung von Herstellern, Anwendern und Forschungseinrichtungen auf dem Gebiet der WfMS. Die WFMC definiert ein WfMS als ein System, das die Ausführung von Workflows durch den Einsatz von Software definiert und leitet. Diese Workflows werden auf einer oder mehreren Workflow Engines ausgeführt. Das WfMS ist dabei in der Lage, die Prozessdefinition zu interpretieren, mit den Teilnehmern des Workflows zu interagieren und die Verwendung von IT-Anwendungen zu initiieren.

Um heutige unterschiedene Workflow-Produkte zu vereinheitlichen, hat die WfMC ein Referenzmodell für ein WfMS entwickelt. Wie in Abbildung 2-1 gezeigt, besteht das Modell aus 6 wesentlichen Teilen. Im Zentrum befindet sich der Workflow Enactment Service, der von 5 Schnittstellen umkreist ist. Der Workflow Enactment Service ist wesentlich eine spezielle Implementierung eines WfMS. Dieser Service kann aus einer oder mehr Workflow-Engines bestehen, um Workflows zu erstellen, zu handhaben und durchzuführen. Anwendungen können zu diesem Service über die standardisierte Workflow API anschließen.

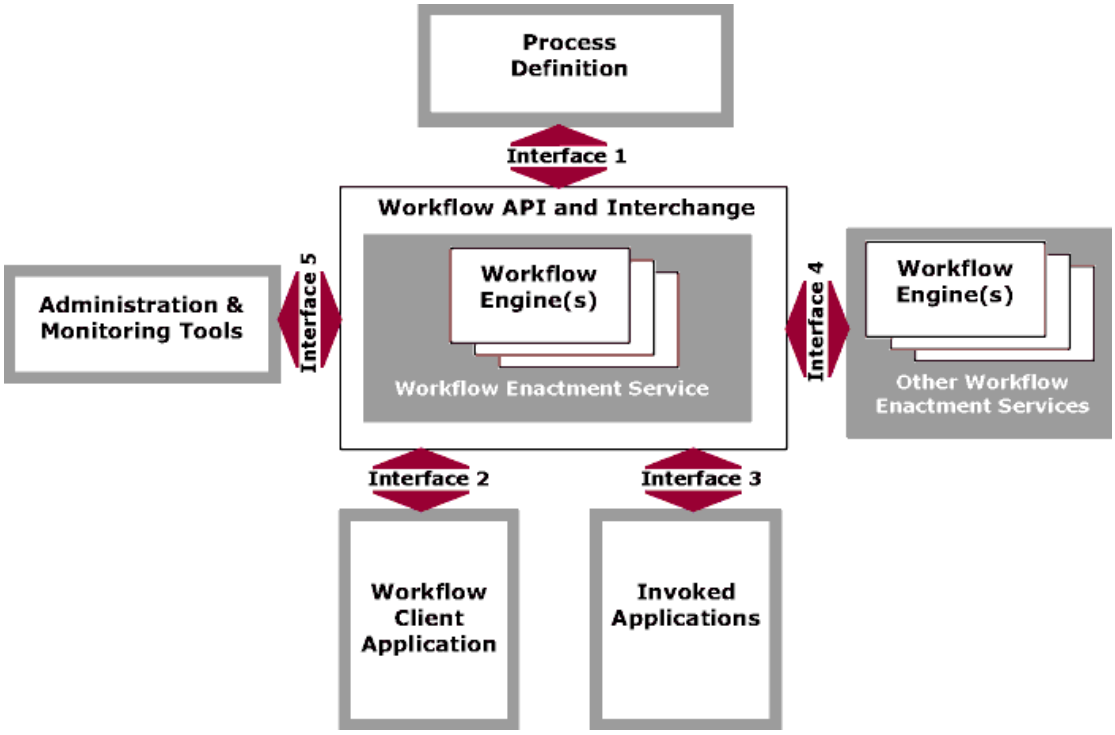

Abbildung 2-1: Workflowreferenzmodell der WfMC [30]

In einem Workflow Enactment Service können eine oder mehrere sog. Workflow Engines implementiert werden. Während die Systeme mit einer Workflow-Engine im Prinzip zentralisiert ist, bestehen nicht so groß verteilte „Workflow Enactment Service“ aus mehreren Workflow-Engines, wobei jede sich nur mit einem bestimmten Aufgabenkreis beschäftigt. Mittels spezieller Protokolle erfolgen Datenaustausch, Synchronisation und Ablaufkontrolle zwischen einzelnen Engines.

Workflow Engines bilden den Kernteil des Workflow Enactment Service und ermöglichen die Ausführung von Geschäftsprozessen. Sie Interpretieren zuerst die Definitionen von den auszuführenden Prozessen und kontrollieren den zur Ausführung benötigten Informationsfluss. Eine Workflow Engine ist im Wesentlichen ein Scheduler, der die Aktivitäten entweder sequentiell oder parallel im Voraus plant und einer ausführenden Ressource zuweist.

2.1.3 Prozessmodellierung mit BPEL

Geschäftsprozesse werden in der Regel durch ein definiertes Prozessmodell spezifiziert. In einem Modell sind die einzelnen Arbeitsschritte, ihre Reihenfolge sowie die Geschäfts- und Prozesslogikregeln enthalten. Diese Spezifikationen werden in für Menschen lesbarer Form, oft unstandardisiert oder anbieterspezifisch abgelegt.

Die Business Process Execution Language (BPEL) ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, deren einzelne

Aktivitäten durch Webservices implementiert sind. Dabei ist zu beachten, dass die direkte Interaktion mit Menschen nicht unterstürzt wird und die mit BPEL modellierten Prozesse ausschließlich mit Web Services kommunizieren. Die Koordination zwischen den einzelnen Aktivitäten entspricht somit der Orchestrierung von Web Services.

Ein BPEL-Prozess umfasst eine Gruppe von Aktivitäten und abhängigen Variablen sowie weiteren Informationen. Das Konstrukt des Prozess wird von seinen Aktivitäten und deren Linken, die alle Aktivitäten miteinander verbinden, dargestellt. Dabei spiegeln die Aktivitäten - Sequence, Flow, Switch und While [12] - die entsprechenden Workflow Patterns wider, die im Abschnitt 4.2 noch eingehend erläutert werden.

2.1.4 Ausführung von Workflows

Der Ablauf des Prozesses wird durch ein Kontroll-Token koordiniert. Deshalb besitzt jede Aktivität mindestens einen mit anderer Aktivität verbundenen Link, über den das Kontroll-Token übertragen wird. Die Ablauffolge von diesem Token während der Ausführung des Prozesses ist der sogenannte Kontrollfluss. Daneben entsteht auch der Datenfluss, der aus den verarbeiteten Daten in den definierten Variablen besteht. Durch den Kontroll- und Datenfluss werden die von den Aktivitäten repräsentieren Web Services logisch so zusammengeführt, dass ein global geschäftliches oder betriebliches Ziel zu erreichen ist.

Wie oben besprochen, der Kern der Ausführung von Workflows ist die Orchestrierung von zusammengesetzten Web Services. Dafür sind Koordinatoren unverzichtbar. Bei zentralisierter Orchestrierung läuft nur ein einzelner Koordinator, um die Anfrage zu bekommen, den entsprechenden Dienst aufzurufen und gegebenenfalls das Ergebnis weiterzuleiten. Er kümmert sich um die Koordination von allen Daten und Kontrollen zwischen Komponenten. So entsteht damit ein Performance-Flaschenhals. Um diesen zu vermeiden, zerlegen wir diese potentiell schwere Aufgabe in relativ kleine Anteile, für die mehrere flexibel und dynamisch verteilte Koordinatoren zuständig sind. Dieses Ausführungsmodell wird als dezentralisierte Orchestrierung angesehen. Doch bringt es auch zusätzliche Komplexitäten bei der Wiederherstellung und Fehlerbehandlung.

2.2 Petri-Netze

2.2.1 Allgemein

Bei Petri-Netzen handelt es sich um formale Konstrukte, die graphisch ausgestaltet und für die Modellierung und Analyse von Systemen und Prozessen geeignet sind. Ein Petri-Netz ist ein bipartiter und gerichteter Graph, der aus zwei verscheidenen Sorten von Knoten besteht: Stellen (Places) und Transitionen (Transitions). Eine Stelle entspricht einer Zwischenablage für Daten und wird als ein Kreis dargestellt, während eine Transition die Verarbeitung von Daten beschreibt und durch ein Rechteck

symbolisiert wird. Die bestehenden Kanten, die gerichtet und gewichtet sind, dürfen nur von einer Knotensorte zur andern führen.

Die Belegung der Stellen wird Markierung genannt und entspricht zugleich auch dem Zustand des Petri-Netzes. Transitionen sind aktiviert, nur wenn eine bestimmte Anzahl von Marken sich in allen Eingangsstellen befindet. Dann können aktivierte Transitionen zu einem beliebigen Zeitpunkt aus deren Eingangsstellen entsprechend den Kantengewichten Marken entnehmen und bei den Ausgangsstellen entsprechend den Kantengewichten Marken hinzufügen. In einem Petri-Netz werden Marken nicht bewegt. Sie werden entweder entfernt oder erzeugt!

Die Marken eines Petri-Netzes unterscheiden sich in ihrer einfachsten Form nicht voneinander. Markeneinfärbungen, Aktivierungszeiten und Hierarchien sind nur für komplexere, aussagekräftigere Petri-Netze definiert worden.

2.2.2 Anwendung zur Modellierung von Workflows

Die Geschäftsprozessmodellierung ist eine typische Anwendung von Petri-Netzen. Ein Petri-Netz, das ein Workflow modelliert, muss zwei Anforderungen erfüllen. Zuerst hat es eine Eingabenstelle (i) für die Marken, welche den zu verarbeitenden Instanzen entsprechen, und eine Ausgabenstelle (o) für die Marken, welche den verarbeiteten Cases entsprechen. Zweitens gibt es keine isolierte Stellen und Transitionen. Das heißt, jede Stelle und Transition muss sich auf einem Weg von der Stelle i zur Stelle o befinden und zur Bearbeitung von Instanzen beitragen. [13]

Das Routing-Konstrukt eines Workflows besteht aus den bestimmten Patterns, die ein Petri-Netz gut unterstützt. Folgend werden die Abbildungen von allen Patterns mit Petri-Netzen erläutert:

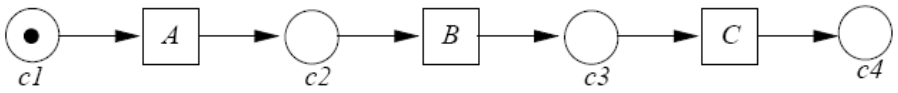

Abbildung 2-2: Sequenzielles Routing [13]

- **Sequenzielles Routing** dient zur Verarbeitung der kausalen Beziehungen von Aufgaben. Angenommen haben wir die Aufgaben A und B. Falls die Aufgabe B nach Beendigung von A ausgeführt wird, wurden A und B sequentiell ausgeführt. Die Abbildung 2-2 zeigt, dass das sequenzielle Routing sich durch die Stellen modellieren lässt. Zum Beispiel c2 repräsentiert eine Nachbedingung von A und eine Vorbedingung von B. c3 beinhaltet die kausale Beziehung zwischen den Aufgaben B und C.

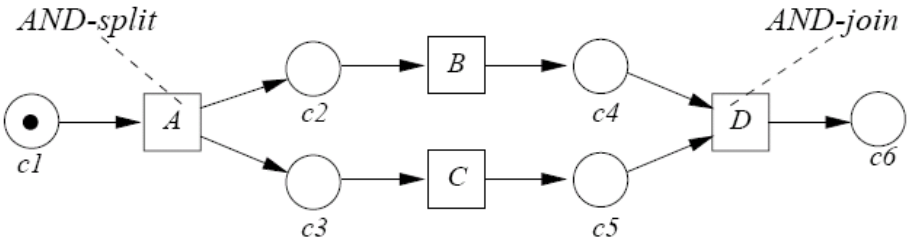

Abbildung 2-3: Paralleles Routing [13]

- **Paralleles Routing** wird verwendet in Situationen, in denen die Ausführungsreihfolge nicht streng geregelt ist. Beispielsweise können die Aufgaben B und C in einer beliebigen Reihfolge ausgeführt werden. Um dieses Routing zu modellieren, brauchen wir zwei Blöcke AND-split und AND-join. Die Abbildung 2-3 verdeutlicht, dass die beiden Blöcke durch übliche Transitionen zu modellieren sind. Die Ausführung von AND-split aktiviert zugleich die beiden Aufgaben B und C. Und nach Beendigung von B und C wird AND-join D aktiviert, sodass auch die zwei Kontrolleflüsse synchronisiert werden können.

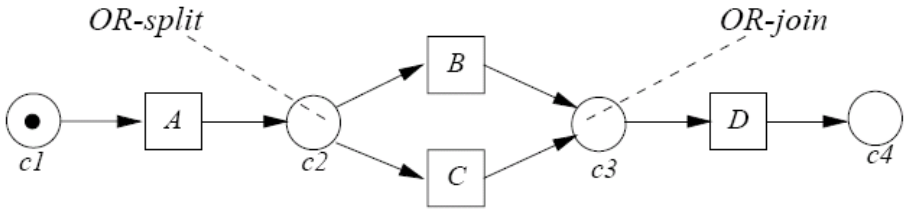

Abbildung 2-4: Konditionelles Routing [13]

- **Konditionelles Routing** wird dazu benutzt, ein sich mit Cases veränderndes Routing zu erlauben. In diesem Fall kann das Routing von vielen Variablen wie den Attributen eines Cases oder der Ausführungsumgebung, abhängig sein. Zur Modellierung solch eines Routings sind zwei Blöcke nötig: der OR-split Block, welcher eine Stelle mit mehreren Ausgängen modelliert, und der OR-join Block, welcher eine Stelle mit mehreren Eingängen modelliert. Die Abbildung 2-4 zeigt ein beispielhaftes konditionelles Routing mit den beiden Blöcken. Nachdem die Aufgabe A ausgeführt ist, wertet die Stelle c2 die beinhaltete Vorbedingung für B und C aus und trifft eine Entscheidung, welche Aufgabe aktiviert werden soll. Der Ausführung einer der beiden Aufgaben folgt die Ausführung von D.

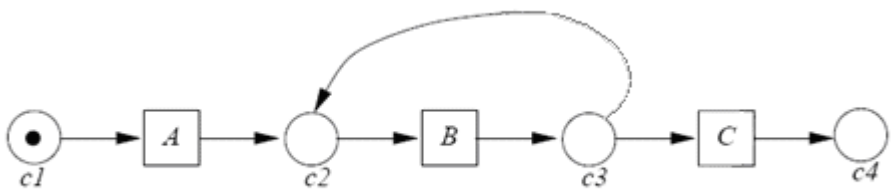

Abbildung 2-5: Iteratives Routing [13]

- **Iteratives Routing** dient zur Darstellung einer Schleife, wobei die Aufgaben iterativ auszuführen sind. Sie kann von den oben besprochen Blöcken modelliert werden. Die Abbildung 2-5 zeigt das iterative Routing, das OR-split und OR-join konstruieren. Nach der Auswertung der Bedingung im c3 könnte die Aufgabe B einmal oder mehrere Mal ausgeführt werden.

2.3 Tuple-Spaces

2.3.1 Linda und Tuple-Spaces

Linda ist eine Sprache zur Beschreibung der verteilten Programmierung, die 1982 von David Gelernter an der Yale University entwickelt wurde. Das Linda Programmiermodell beruht auf dem Konzept des Tupelraums (Tuple-Space), der als zentral zur Verfügung stehender Datenbehälter dient. Bei den Datenelementen des Tuple-Spaces handelt es sich um Sequenzen typisierter Datenwerte, diese Sequenzen heißen Tupel.

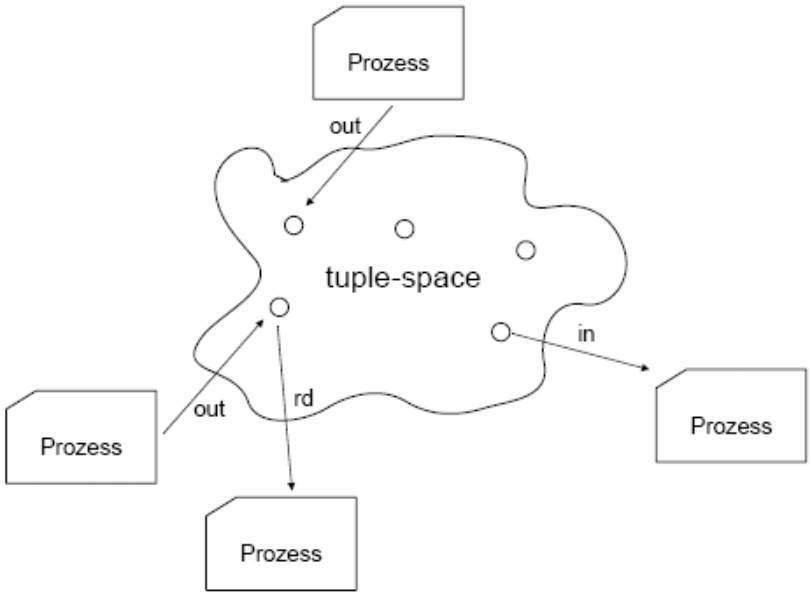

Abbildung 2-6: Das Tupelraum-Modell

Dem Kommunikationsaspekt zu Grunde liegt, dass es sich beim Tupelraum um eine verteilte Ressource (Tupel) handelt. Es werden keine Nachrichten mit einer expliziten Zieladresse versehen, sondern Datenelemente allen Prozessen zur Verfügung gestellt. Wie in Abbildung 2-6 gezeigt, läuft ein einzelner Kommunikationsvorgang so ab, dass ein Prozess einen Tupel innerhalb des Tupelraums generiert (*out*). Dieser Tupel ist somit verfügbar für alle Prozesse, die den Zugang zum Tupelraum haben. Ein Prozess davon empfängt diese Nachricht, indem er den gerade generierten Tupel ausliest (*rd*) oder entnimmt (*in*). Dabei

könnte die Entnehmens-Operation gegebenenfalls eine Blockierung verursachen. Nämlich wartet der Prozess solange bis ein gewünschtes Tupel im Tupelraum vorhanden ist und somit entnommen werden kann.

Beim Lesen oder Entnehmen muss der Prozess eine Tupel-Vorlage angeben, um eine Anfrage an den Tupelraum zu stellen. Die Vorlage beschreibt das gesuchte Tupel durch leer gebliebene Felder (dargestellt mit dem Sternzeichen „*“) oder konkrete Werte der Felder. Im Tupelraum wird auf Basis eines Wertevergleichs nach einem beliebigen Tupel gesucht, das mit der Vorlage übereinstimmt. Eine Übereinstimmung besteht, nur wenn

- die Anzahl und auch die Typen der Felder von dem Tupel und der Vorlage gleich sind und
- die Werte der Tupel-Felder denen der dazugehörigen Template-Felder entsprechen.

Dabei ist zu beachten, dass das Sternzeichen als Platzhalter für den Wert eines bestimmten Types dient. Zum Beispiel um nach dem Tupel (process1, 122, true) zu suchen, können wir die folgenden Vorlagen benutzen:
(prozess1, *:integer, *:boolean), (prozess1, 122, true).

Alle Prozesse haben Zugriff auf einen Tupelraum. Die Kommunikation geschieht durch Manipulation der zentralen Ressourcen des Tupelraums. Bei dieser Kommunikation über den Tupelraum sind zusammengefasst folgende Eigenschaften gewährleistet:

- Asynchronität: die Kommunikationspartner müssen nicht zu einem bestimmten Zeitpunkt miteinander kommunizieren. Da jeder Tupel im Tupelraum persistent gesichert ist und somit existiert, es ist ihnen erlaubt, zu beliebiger Zeit asynchron auf den Tupelraum zuzugreifen. Es findet keine Blockierung des Senders statt, weil er nicht auf die Antwort des Empfängers warten muss.
- Anonymität: der Gesprächspartner muss nicht wissen, wer der jeweils andere ist. Sie beide müssen lediglich den Tupelraum kennen.
- Räumliche Trennung: Sender und Empfänger können sich in unterschiedlichen Prozessen oder auf unterschiedlichen Rechnern befinden, solange sie denselben Tupelraum benutzen.

Deshalb sind die über den Tupelraum miteinander verbunden Prozesse lediglich schwach gekoppelt. Linda führte ein neues Paradigma von Integrationstechniken neben dem nachrichtenorientierten Middleware und dem Workflow-Management-System (WfMS) ein.

2.3.2 Verteilte Anwendung auf Basis von Tuple-Spaces

Eine verteilte Anwendung besteht aus mehreren Prozessen, die auf verschiedenen, miteinander vernetzten Rechner laufen. Um ihre gemeinsame Aufgabe zu erfüllen, müssen die verteilten Prozesse durch Datenaustausch miteinander interagieren. Eine typische Interaktions-Methode ist die Kooperation, wobei mehrere Prozesse auf einem

gemeinsamen Datenbereich arbeiten und dadurch in der Lage sind, Nachrichten auszutauschen. Der Tupelraum entspricht genau einem derartigen Datenbereich.

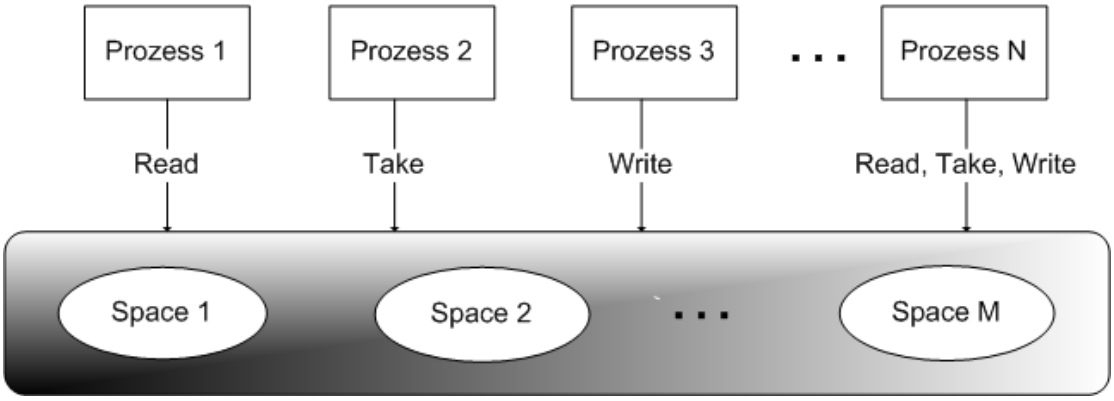

Abbildung 2-7: Architekturmodell der tupelraumbasierten verteilten Anwendung

Die Abbildung 2-7 veranschaulicht das Architekturmodell der verteilten Anwendung auf Basis von Tuple-Spaces. In diesem Modell spielen alle Prozesse ähnliche Rollen und arbeiten gleichrangig zusammen, um eine verteilte Aktivität durchzuführen, ohne dass zwischen Clients und Servern unterschieden wird. Die Kommunikation geschieht nicht direkt zwischen zwei beliebigen Prozessen, sondern dadurch, dass die Gesprächspartner innerhalb eines oder mehrerer Tuple-Spaces bestimmte Nachrichten austauschen. Um Tuple-Spaces voneinander zu unterscheiden, erweitern wir den traditionellen Tupelraum um einen eindeutigen Bezeichner, der ein URI sein darf. Alle Prozesse greifen immer mit derartigen Bezeichnern auf die Tuple-Spaces zu, sodass die Tuple-Spaces mit dem gebunden Hostnamen auf anderen Rechner umplatziert werden können, ohne die Zugriffe von den Prozessen zu beeinflussen.

2.3.3 Ähnlichkeiten zu Petri Netzen

Wenn wir uns eingehend das oben besprochene Modell ansehen, können viele Ähnlichkeiten zu Petri Netzen herausgefunden werden. Der Tupelraum nimmt die Tupel als Marken von Prozessen an und überträgt sie zu anderen Prozessen. Es verhält sich genau so wie die Stelle im Petri Netz. Beide dürfen nicht mit denselben Sorten von Elementen kommunizieren. Auch entspricht der Prozess der Transition, um neue Tupel zu erstellen, oder die vom Tupelraum empfangen Tupel zu bearbeiten und zum weiteren Tupelraum weiterzuleiten. Dabei übernehmen die Tupel die Rolle der Marken im Petrinetz. Die Transformation des Petrinetzes zum tuplespace-basierten Verteilungsmodell ist somit keine schwierige Aufgabe. Besonderes für die Petrinetze, die Workflows modellieren, weil ein Workflow selbst im Wesentlichen auch eine verteilte Anwendung ist.

2.3.4 Beschränkungen

2.3.4.1 Endloses Blockieren

Die Operationen *rd* und *in* blockieren, falls kein der Anfrage entsprechendes Tupel gefunden wird. Kann beim Programmentwurf nicht sichergestellt werden, dass ein solches existiert, besteht die Gefahr auf ein Ereignis zu warten, welches nie eintritt, und somit das Programm niemals terminiert. Möchte man jedoch eine gewisse Zeitspanne auf den Eintritt eines Tupels in den Tupelraum warten, so muss entweder auf busy-waiting zurückgegriffen werden oder der Clientprozess muss sich selbst für eine gewisse Zeitspanne suspendieren, in der selbstverständlich ein angekommenes Tupel schon wieder entfernt worden sein kann.

2.3.4.2 Problem mit mehrfachem Lesen

Aufgrund der alleinigen Zugriffsmöglichkeit durch die assoziativen Operationen an dem Tupelraum ist es schwer, innerhalb eines Prozesses eine Übersicht über die im Tupelraum enthaltenen Elemente zu erhalten.
Sind mehrere einer Vorlage entsprechende Tupel vorhanden, führt wiederholtes anwenden der *rd* Operation mit dieser Vorlage jedoch nicht zu dem gewünschten Ergebnis. Keine Forderungen in Bezug auf die Auswahlstrategie bestehen auf der Seite vom Lindamodell. So ist es möglich, mehrmals dasselbe Tupel zu erhalten.

Seitens des Clientprozesses lässt sich das Problem im Rahmen der ursprünglichen Linda Operationen dadurch lösen, dass der Client in einem Synchronisationsbereich alle vorlagenkonformen Tupel anhand *in* entnimmt. Damit erhält er eine vollständige Liste der Tupel die er danach wieder zurückschreibt. Diese Problemlösung wird mit erhöhter Codekomplexität erkauft. Der sonst sehr klare und ansprechende Charakter der Linda Programme wird von einer weniger gut verständlichen Darstellung beschädigt, da immer wieder Anweisungen zur Koordinierung des Zugriffes eingesetzt werden. Diese Beschädigung verhindert es auch, ein gut strukturiertes paralleles Programmierparadigma wie das nachrichtorientierte System aufzubauen. Somit ist eine Erweiterung des Linda Modells in dieser Hinsicht folgerichtig. Nach dem Vorschlag in [15] führen wir eine weitere Operation namens *scan* ein, die das Problem mit mehrfachem Lesen elegant lösen kann, ohne für die Synchronisation zwischen den beteiligten Prozessen zu sorgen. Hierbei handelt es sich um eine kollektive Operation, die alle Tupel kopiert, welche einer gewissen Vorlage entsprechen.

2.3.4.3 Aufstauung

Das eben beschriebene Problem macht es schwierig, eventuell vorhandene überflüssige und verwaiste Einträge zu finden, die zum Beispiel nach dem Ausfall eines Systems entstanden sind oder Hilfsergebnisse darstellen, die nach dem vorzeitigen Ende einer

Berechnung nicht weiter benötigt werden. Aufgrund des Fehlens kollektiver Operationen kann der Clientprozess nicht auf Tupel zugreifen, die gewissen Kriterien nicht genügen. Auch ist es nicht möglich, dass der Tupelraum selbst entscheidet, ob ein Tupel bereinigt werden soll oder nicht, da er unabhängig von den Anwendungen ist, die ihn nutzen, und somit überhaupt die Bedeutungen aller Tupel nicht kennt. Idealerweise existiert ein Prozess, der alle Clientprozesse des Tupelraums überwacht und periodisch eine Bereinigungsaufgabe für den Tupelraum durchführt. Alternativ lässt ein Timeout für jeden ankommenden Tupel einsetzen. Sobald eine gewisse Zeitspanne abgelaufen ist, entfernt der Tupelraum den entsprechenden Tupel automatisch.

3 Middleware für Kooperation zwischen verteilten Prozessen

Dieses Kapitel beschäftigt sich mit der Middleware auf Basis des Tupelraum-Konzeptes. Die aufzubauende Middleware stellt einen gemeinsam genutzten Speicherraum zur Verfügung, damit die verteilten Prozesse Nachrichten austauschen können, um zu einem gewissen Ziel zusammenzuarbeiten. Das ursprüngliche Tupelraum-Konzept kommt nicht mit den anspruchsvollen Anwendungen im Workflow Bereich zurecht. Auch aufgrund der in 2.3.4 besprochenen Beschränkungen muss das Lindamodell entsprechend erweitert werden.

3.1 Anforderungen

Ein BPEL-Prozess, der im verteilten Workflow-Engine ausgeführt wird, teilt sich in mehrere über ein Rechnernetz verteilte Teilprozesse. Sie müssen in der Lage sein, miteinander zu kooperieren, um ihre gemeinsame Aufgabe zu erfüllen. Wie im 2.3.2 besprochen, entsprechen Tupelräume diesem Anspruch. Daraus entwickelt sich eine Middleware, welche die Komplexität des zu Grunde liegenden Rechnernetzes verbergen und eine vereinheitlichte Schnittstelle für den Zugriff auf Tupelräume bieten kann. Die Ausführung von Workflows soll immer zuverlässig, effizient und auch unabhängig von der Verteilung von auszuführenden Prozessen sein. Deshalb muss die Middleware die folgenden Aspekte berücksichtigen:

Persistenz: Um die asynchronen Kommunikationen über einen Tupelraum zu gewährleisten, sollte der Zustand des Tupelraums im Speicher des ihn ausführenden Prozesses abgelegt sein. Persistente Tupelräume überdauern die Lebenszeit des Prozesses. Darum müssen Tupelräume ihren Zustand an einen nichtflüchtigen Speicher wie gebräuchliche Festplatten übertragen. Ist ihr Server abgestürzt, können sie aus dem Speicher wiederhergestellt werden.

Zuverlässigkeit: Eine zuverlässige Middleware sollte nicht nur die Kommunikation zwischen Clients und Server, sondern auch die Ausführung der angeforderten Operationen korrekt behandeln. Nach dem Auftreten von Fehlern in Hardware, Software oder Netzwerken sollte sie sich wiederherstellen können und korrekt weiterarbeiten, ohne die angebotenen Dienste zu stoppen.

Synchronisierung: Tupelräume bieten einen gemeinsam genutzten Speicherraum. Deshalb besteht die Möglichkeit, dass mehrere Prozesse gleichzeitig versuchen, auf dieselben darin existierenden Tupel zuzugreifen. Innerhalb eines Tupelraums müssen alle solche Operationen synchronisiert werden, um einen aktuellen Wert zu erhalten und auch um die Datenverluste zu vermeiden, die durch Konflikte beim mehrmaligen Schreiben verursacht werden . Außerdem könnten mehrere Tupel, die Kontrolltoken beinhalten, zugleich abgeholt werden, damit die Synchronisierungssemantik von WS-BPEL Joins implementiert wird.[22]

Diesen Fall sollte der Server durch eine zusätzliche Methode unterstürzen. Die Synchronisierung wird immer auf der Serverseite durchgeführt.

Transaktion: Bei den anspruchsvollen Anwendungen sollte eine Folge von Operationen für einen oder mehrere Tupelräume durch eine Transaktion zusammengefasst werden, sodass sie nach den ACID-Eigenschaften ausgeführt werden. Das bedeutet, dass eine Transaktion entweder vollständig oder gar nicht ausgeführt wird (Atomizität), sie von einem konsistenten Zustand zum nächsten führt (Konsistenz), sie von anderen Transaktionen isoliert ausgeführt wird (Isolation) und einmal ausgeführt, dauerhaft ist (Dauerhaftigkeit). Weiterhin könnten die Operationen innerhalb einer Transaktion mehrere Tupelräume betreffen, die auf unterschiedlichen Servern verteilt sind. Deshalb sind hierzu verteilte Transaktionen notwendig.

XML- und XPath- Unterstürzung: XML ist eine vom W3C standardisierte Auszeichnungsprache zur Textdarstellung strukturierter Daten und wird zurzeit überall zum Datenspeichern verwendet. Mit XPath kann man Teile von XML Dokumenten adressieren und dann auslesen oder ändern. Sind die Daten, welche ein Tupel beinhaltet, mit XML formuliert, kann man diesen Tupel mit XPath sehr effizient manipulieren. Damit gewinnt es sicherlich an Bedeutung, XML/XPath beim Zugriff auf einen Tupelraum zu unterstützen.

3.2 Architektur

Die Abbildung 3-1 zeigt die Architektur der zu entwerfenden Middleware. Das Kernmodul ist Tupelräume-Dienst, der sich um mehrere Tupelräume kümmert und damit den Clients mehrere gemeinsam genutzte Speicherräume zur Verfügung stellt. Die Clients verwenden die Tupelraum-API zum Hinzufügen, Lesen, Entnehmen und Ändern des Tupels. Um die Nebenläufigkeit aller Operationen zu kontrollieren, ist es auch möglich, dass die Clients ihre Operationen transaktionsorientiert ausführen. Der Transaktions-Manager kümmert sich um die korrekte Behandlung jeder Transaktion, die von ihm erstellt ist. Es können mehrere Transaktions-Manager gleichzeitig laufen, um einem Flaschenhals zu vermeiden. Der Persistenzdienst ermöglicht es den anderen Komponenten ihre Daten persistent zu speichern und beim erneuten Serverstarten sie wiederherzustellen. In den folgenden Abschnitten wird jede Komponente eingehend erläutert.

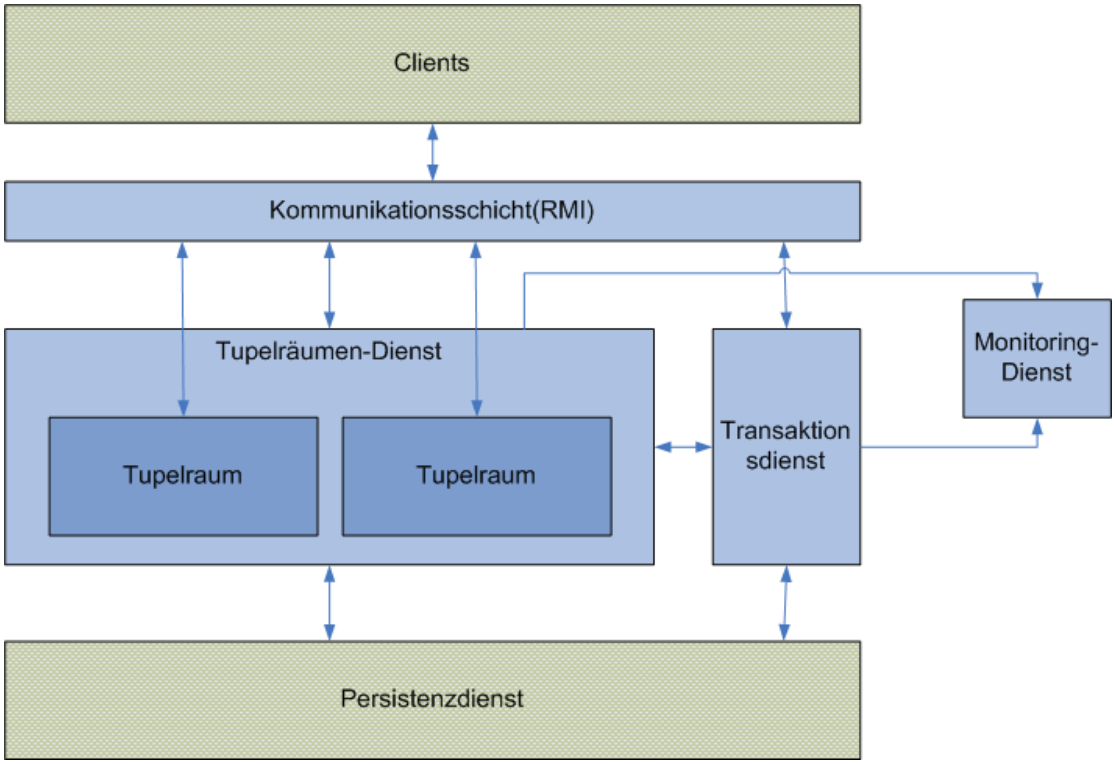

Abbildung 3-1: Middleware-Architektur

3.3 Persistenzdienst

Sowohl die Tupelräume als auch die Transaktionen sind als zustandsbehaftete Serverobjekte aufzufassen. Um die Forderung zu erfüllen, dass derartige Objekte Ihren Zustand nicht verlieren dürfen, wird es notwendig, den Zustand vor Prozessende in einem Sekundärspeicher abzulegen und ihn bei erneuerter Objektaktivierung von dort wieder zu laden. Die Implementierung dieses Speicherns und Wiederherstellens wird von einem Persistenzdienst unterstürzt.

Objektorientiertes Speichern

Ein zustandsbehaftetes Serverobjekt muss einige nicht flüchtig deklarierte Attribute haben, welche seine Zustandsinformation darstellen, die zwischen den Objektinstanzen unterscheiden kann. Das Speichern einer Objektinstanz ist damit im Wesentlichen ihren Zustand aus den Attributen zu sichern. Dabei generiert der Persistenzdienst eventuell ein entsprechendes Datenspeicherobjekt, aus dem die Objektinstanz wiederherzustellen ist.

Dateisysteme und Datenbanksysteme

Um Persistenz zu erreichen, können Persistenzdienste Dateisysteme oder Datenbanksysteme verwenden. Ein Dateisystem bietet bei der Organisation von Daten nur eingeschränkte Möglichkeiten. So ist z.B. ein Zugriff auf die einzelnen Daten nur über das Anwendungsprogramm

möglich und damit die Forderung nach Datenunabhängigkeit nicht erfüllt. Datenbanksysteme beseitigen bzw. vermeiden die typischen Probleme, die durch die Datenorganisation mit Hilfe von Dateisystemen entstehen. Diese Systeme integrieren die Daten und ermöglichen so eine zuverlässige, unabhängige und komfortable Verwaltung der Daten.

Allerdings sind die relationalen und auch objektorientierten Datenbanken in vielen Fällen für die Objektpersistenz zu kompliziert und die Leistung beim Zugriff auf die gespeicherten Daten wegen des großen Overhead stark eingeschränkt. Dagegen kann ein effizienter und leichtgewichtiger Persistenzdienst auf Basis von einem Dateisystem errichtet werden, obwohl er viele Aspekte selbst implementieren muss, um die gewünschten Forderungen wie z.B. nach Transaktionssicherheit und Zuverlässigkeit zu erfüllen. Ein gutes Beispiel ist das Prevayler System (Siehe 3.8.1.1).

Transparenz der Persistenz

Wir haben zwei Mechanismen zur Implementierung des Persistenzdienstes besprochen. Es lohnt sich darauf hinzuweisen, dass der konkrete Mechanismus, der für die Erreichung der Persistenz gewählt wurde, für manche Entwickler transparent sein kann, während er für andere nicht transparent ist. Die Verwendung eines Persistenzdienstes ist normalerweise für Client-Entwickler transparent. Der Clientprogrammierer betrachtet nur die veröffentlichte Schnittstelle des Serverobjekts und weiß nicht, wie Persistenz in der Implementierung der Schnittstelle erreicht wurde. Die Entwickler des zustandbehafteten Serverobjekts müssen sich im Allgemein über den Mechanismus bewusst sein, den er benutzt, um Persistenz zu implementieren.

3.4 Kommunikationsschicht

Diese Schicht dient der Kommunikation zwischen den verteilten Serverobjekten innerhalb der Middleware und den Clientobjekten. Sie basiert auf entfernte Methodenaufrufe (RMI, Remote Methode Invocation) und dem zu Grunde liegenden Anforderungs- und Antwort- Protokoll. In den folgenden Abschnitten werden die beiden Konzepte erklärt.

3.4.1 Anforderungs- und Antwort- Protokoll

Das nachfolgend beschriebene Protokoll, das die Middleware-Abstraktionen unterstürzen, ist unabhängig von den zu Grunde liegenden Transport-Protokollen. Es kann sowohl über UDP als auch über TCP implementiert werden. Die meisten RMI- und RPC- Systeme erwarten die Unterstützung durch ein ähnliches Protokoll.

Protokoll

Im Normalfall läuft die Kommunikation über das Protokoll ab, so wie in Abbildung 3-2 gezeigt. Der Client schickt eine Anforderungsnachricht, die Spezifikation der beim Server auszuführenden Operation und

gegebenenfalls Operationsparameter enthält, an den Server und wartet auf Antwort. Der Server führt die empfangene Operation aus und sendet eine Antwortnachricht, die gegebenenfalls Ergebnisse der Operation enthält, an den Client zurück. Diese Antwortnachricht dient auch einer Bestätigung für den Client, dass die Operation ausgeführt wurde. Dann setzt der Client seine Arbeit fort. Daraus ergibt sich, dass die Anforderung- /Antwort- Kommunikation synchron und zuverlässig ist.

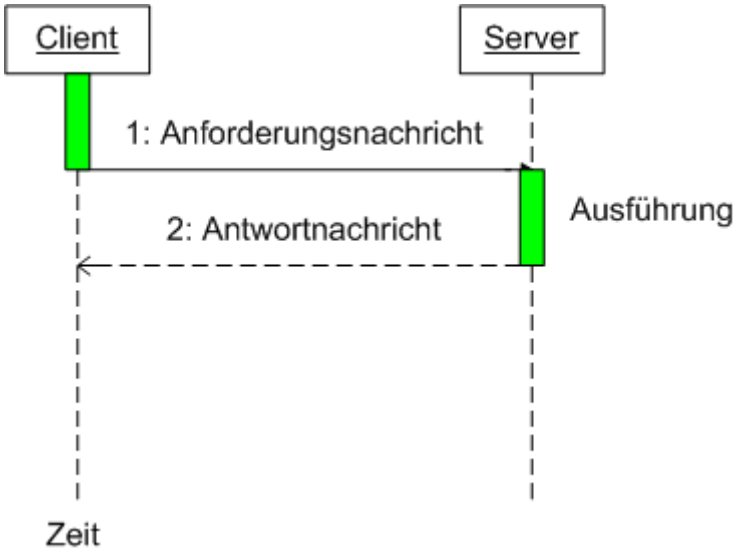

Abbildung 3-2: Anforderungs- / Antwort- Kommunikation

Fehlersituation und Fehlererkennung

Während jeder Phase des besprochenen Ablaufs könnten Fehler auftreten, welche die Kommunikation unterbrechen würden:

- *Verlust der Anforderung*: die Anforderungsnachricht geht verloren durch Auslassungs- oder Ausfallfehler der Netzverbindung oder durch Ausfall des Servers vor dem Empfang der Nachricht
- *Serverabsturz*: der Ausfallfehler des Servers tritt nach dem Empfang der Anforderungsnachricht und vor dem Senden der Antwortnachricht auf und wird durch Neustart des Servers maskiert. Es ergibt sich ein Auslassungsfehler beim Server. Gegebenenfalls werden Teile der Operation und das Senden der Antwort nicht durchgeführt.
- *Verlust der Antwort*: die Antwortnachricht geht verloren durch Auslassungs- oder Ausfallfehler der Netzverbindung.
- *Clientabsturz*: der Ausfallfehler des Clients geschieht nach dem Senden der Anforderungsnachricht. Normalerweise ist er kein Problem und kann durch Neustart des Clients verdeckt werden. Und gegebenenfalls resultiert daraus ein Auslassungsfehler beim Client. D.h. Die Antwortnachricht wird nicht mehr empfangen.

Um diese Fehlersituation zuzulassen, verwendet der Client ein Timeout beim Warten auf die Antwortnachricht des Servers. Wenn ein Timeout auftritt, wird ein Fehler, Nachrichtenverlust oder Serverausfall erkannt. Man kann aber nicht zwischen den Fehlersituationen unterscheiden.

Fehlerbehandlung

Werden Fehler durch Timeout erkannt, sendet der Client die Anforderungsnachricht wiederholt, bis er entweder eine Antwort erhält oder anderweitig ausreichend sicher ist, dass die Verzögerung auf Grund einer fehlenden Antwort vom Server und nicht durch verloren gegangene Nachrichten aufgetreten ist.

Mehrfache Anforderungsnachrichten entstehen ggf. durch die Behandlung von durch Timeout erkannten Ausfall- und Auslassungsfehlern, und durch Neustart des Clients zur Maskierung von Clientabstürzen, falls Anforderungen nicht protokolliert werden. Wenn die Anforderungsnachricht wiederholt übertragen wird, empfängt der Server sie möglicherweise auch mehrfach. Vielleicht hat der Server die erste Anforderungsnachricht erhalten, aber die Ausführung der Operation und die Rückgabe eines Ergebnisses dauerten länger als das Timeout des Clients. Das kann dazu führen, dass der Server eine Operation für dieselbe Anforderung mehrmals ausführt. Um dies zu vermeiden, wurde die Anforderungsnachricht um eine eindeutige Anforderungs-ID erweitert und der Server protokolliert alle empfangenen Anforderungs-IDs, sodass der Server Duplikate ausfiltern kann. Hat der Server die Antwort noch nicht gesendet, ist keine spezielle Aktion erforderlich. Er überträgt die Antwort, wenn er mit der Ausführung der Operation fertig ist.

Wenn der Server die Antwort bereits gesendet hat und dann eine doppelte Anforderung erhält, muss er die Operation erneut ausführen, um das Ergebnis zu erhalten, es sei denn, er hat das Ergebnis der ursprünglichen Ausführung gespeichert. Falls die Operation idempotent ist, kann sie wiederholt ausgeführt werden und hat jedes Mal dasselbe Ergebnis, als wäre sie genau einmal ausgeführt worden. Beispielsweise ist eine Operation, die einer Menge ohne Duplikate ein Element hinzufügt, eine idempotente Operation, weil sie für die Menge bei jeder Ausführung denselben Effekt erzeugt, während eine Operation, die einer Liste ein Element anfügt, nicht idempotent ist, weil sie die Liste bei jeder Ausführung erweitert. Ein Server, dessen Operationen alle idempotent sind, muss keine speziellen Maßnahmen ergreifen, um sicherzustellen, dass die Operationen nicht mehrfach ausgeführt werden.

Die Server, die eine erneute Übertragung von Antworten anfordern, ohne dass Operationen erneut ausgeführt werden, kann eine Historie verwenden. Der Begriff Historie beschreibt eine Struktur, die eine Aufzeichnung der Antwortnachrichten enthält, die übertragen wurden. Ein Eintrag in eine Historie enthält eine Anforderungs-ID, eine Nachricht sowie eine ID des Clients, an den sie gesendet wurde. Sie soll dem Server ermöglichen, Antwortnachrichten erneut zu übertragen, wenn der Client-Prozess sie anfordert. Ein Problem bei der Bereitstellung einer Historie ist der Speicheraufwand. Eine Historie kann sehr groß werden, es sein denn, der Server erkennt, welche Nachrichten nicht mehr benötigt werden, um sie erneut zu übertragen.

Das Anforderungs-Antwort-Protokoll kann somit darauf ausgelegt werden, eine Bestätigungsantwort vom Client an den Server zu senden, um zu

bestätigen, dass der Client die Antwort vom Server erhalten hat, sodass der Server den entsprechenden Eintrag aus der Historie löschen kann. Allerdings haben Client-Abstürze, die zum Verlust der Bestätigungsantwort führen, jetzt Auswirkung auf den Server. Um dieses Problem zu beseitigen, kann der Server jede Antwort als Bestätigung seiner vorhergehenden Anforderung betrachten, weil Clients immer nur eine Anforderung gleichzeitig vornehmen können. Die Historie muss also nur die letzte an jeden Client gesendete Antwortnachricht enthalten. Die Menge der Antwortnachrichten in der Historie eines Servers kann jedoch ein Problem darstellen, wenn es sehr viele Clients gibt. Insbesondere wenn ein Client-Prozess beendet wird, bestätigt er die letzte erhaltene Antwort nicht. Nachrichten in der Historie werden deshalb nach einer bestimmten Zeitdauer verworfen.

3.4.2 Entfernte Methodenaufrufe (RMI)

In einem verteilten objektorientierten System ist die Interaktion zwischen Objekten durch lokale oder entfernte Methodenaufrufe realisiert. Objekte, die entfernte Methodenaufrufe empfangen könnten, werden auch als entfernte Objekte bezeichnet und implementieren eine entfernte Schnittstelle. Weil Aufrufer und aufgerufene Objekte unabhängig voneinander fehlschlagen können, haben RMIs eine andere Semantik als lokale Aufrufe. Sie können lokalen Aufrufen sehr ähnlich gemacht werden, eine totale Transparenz ist nicht unbedingt wünschenswert. Der Code für das Mashalling und Un-Mashalling von Argumenten und das Senden von Anforderungs- und Antwort-Nachrichten kann automatisch von einem Schnittstellen-Compiler aus der Definition der entfernten Schnittstelle erzeugt werden.

Aufrufsemantik

Das Verhalten von Methodenaufrufen beim Server ist abhängig von den verwendeten Fehlerbehandlungsmaßnahmen. Die möglich vorkommenden Aufrufsemantiken sind wie folgt aufgelistet:

- *Vielleicht-Aufrufsemantik (maybe)*: Bei der Aufrufsemantik vielleicht kann der Aufrufer nicht erkennen, ob eine entfernte Methode einmal oder überhaupt nicht ausgeführt wurde. Die Vielleicht-Aufrufsemantik liegt vor, wenn keine der Fehlerbehandlungsmaßnahmen gegen Auslassungs- und Ausfallfehlern angewendet wird. Sie ist nur verwendbar, wenn die Auslassung der Aufrufe im Fehlerfall toleriert werden kann.

- *Mindestens-Einmal-Aufrufsemantik (at-least-once)*: Bei der Mindestens-Einmal-Aufrufsemantik empfängt der Aufrufer entweder ein Ergebnis, sodass er weiß, dass die Methode mindestens einmal ausgeführt wurde, oder eine Ausnahme, die ihn darüber informiert, dass kein Ergebnis empfangen wurde. Die Mindestens-Einmal-Aufrufsemantik kann durch die wiederholte Übertragung von Anforderungsnachrichten realisiert werden, wodurch Nachrichten-

verluste maskiert werden. Falls die Objekte in einem Server so entworfen werden können, dass alle Methoden ihrer entfernten Schnittstellen idempotente Operationen sind, ist die Mindestens-Einmal-Aufrufsemantik akzeptiert.

- *Höchstens-Einmal-Aufrufsemantik* (at-most-once): Bei einer Höchstens-Einmal-Aufrufsemantik empfängt der Aufrufer entweder ein Ergebnis, sodass er weiß, dass die Methode genau einmal ausgeführt wurde, oder eine Ausnahme, sodass davon ausgegangen werden kann, dass die Methode entweder einmal oder überhaupt nicht ausgeführt wurde. Die Höchstens-Einmal-Aufrufsemantik wird erreicht, indem alle Fehlerbehandlungsmaßnahmen eingesetzt werden. Wie im vorigen Fall maskiert die Verwendung von Wiederholungen der Anforderungs-nachrichten alle Nachrichtenverluste. Mit den zusätzlichen Fehlerbehandlungsmaßnahmen wie Duplikatfilterung und Historie wird sichergestellt, dass eine Methode für jede RMI nur höchstens einmal ausgeführt wird. Die Höchstens-Einmal-Aufrufsemantik wird dazu verwendet, nicht idempotente Operationen sicher auszuführen.

Idealerweise wird eine vierte Aufrufsemantik, nämlich Exakt-Einmal-Aufrufsemantik, erwartet. Dabei würde die aufgerufene Methode immer genau einmal ausgeführt. Jedoch ist es in der Tat unerreichbar, da weder der Server noch das verbundene Netz sicherstellen kann, dass alle möglichen Fehler zu tolerieren sind. Sowohl in Java RMI als auch in CORBA liegt somit eine Höchstens-Einmal-Aufrufsemantik vor, welche in den meisten Fällen ausreicht.

Entfernte Objektreferenz

Beim entfernten Methodenaufruf muss die Anforderungsnachricht angeben, welches bestimmte Objekt die Methode aufrufen soll. Eine entfernte Objektreferenz ist ein Bezeichner für ein entferntes Objekt, das innerhalb des gesamten verteilten Systems gültig ist. Eine entfernte Objektreferenz wird in der Anforderungsnachricht übergeben, um damit anzugeben, welches Objekt aufgerufen werden soll.

Die entfernte Objektreferenz ist stets einer lokalen Objektreferenz in einem Serverprozess zugeordnet. Fällt der Server aus, wird die entfernte Objektreferenz bei der statischen Zuordnung ungültig, auch wenn der Server erneut gestartet ist. Um den Serverausfallfehler zu tolerieren, soll der abgestürzte Server wieder aktiviert und die entfernte Objektreferenz automatisch der neuen lokalen Objektreferenz zugeordnet werden können, sodass die entfernte Objektreferenz für den Client immer verfügbar zu sein scheint.

3.5 Transaktionsmanager

3.5.1 Verteilte Transaktion

Unter dem Begriff „verteilte Transaktion“ versteht man eine Transaktion, die auf Objekte zugreift, die von mehreren Servern verwaltet werden.

Wird eine verteilte Transaktion abgeschlossen, ist es für die Atomarität der Transaktion erforderlich, dass entweder alle beteiligten Server die Transaktion festschreiben oder dass alle sie abbrechen. Um dies zu erreichen, übernimmt einer der Server die Rolle eines Koordinators, in der er sicherstellt, dass auf allen Servern dieselbe Maßnahme ergriffen wird. Zur Realisierung des Koordinators wählen wir das gebräuchlichste Protokoll, nämlich das so genannte Zwei-Phasen-Commit-Protokoll. Dieses Protokoll erlaubt den Servern, miteinander zu kommunizieren, um eine gemeinsame Entscheidung zu treffen, ob die Transaktion festgeschrieben oder abgebrochen werden soll. Verteilte Transaktionen erhalten mithilfe dieses Protokolls die ACID-Eigenschaften, die im Folgenden besprochen werden:

- *Atomarität*: Eine Transaktion wird entweder erfolgreich vollständig oder überhaupt nicht ausgeführt. Tritt inmitten einer Folge von Operationen ein Fehler auf, können einige Operationen bereits ausgeführt sein, während andere aufgrund des Fehlers nicht mehr bearbeitet werden können. Wird diese Folge von Operationen als eine Transaktion ausgeführt, sogt der Transaktionsmanager dafür, dass die sich durch die bereits ausgeführten Operationen ergebende Wirkungen rückgängig gemacht werden und der Zustand des verteilten Systems auf den Punkt vor Start der Transaktion zurückgesetzt wird.

- *Konsistenz*: Eine Transaktion bewahrt die Konsistenz, indem sie einen konsistenten Zustand des Systems in einen anderen konsistenten Zustand überführt und somit die existierenden anwendungsspezifischen Konsistenzbedingungen nicht verletzt. Das bedeutet nicht, dass Inkonsistenz nicht vorkommen kann, sondern sie innerhalb einer Transaktion eingeschlossen ist. Kann eine Transaktion keinen konsistenten Zustand erreichen, darf sie zuletzt nicht festgeschrieben werden. Die Transaktion muss abgebrochen und alle bisherigen Änderungen rückgängig gemacht werden.

- *Isolierung*: Eine Transaktion wird isoliert von anderen nebenläufigen Transaktionen ausgeführt. Das bedeutet, dass es keine Beeinflussungen zwischen zwei beliebig nebenläufigen Transaktionen geben kann. Änderungen, die eine Transaktion am System vornimmt, können anderen Transaktionen erst nach Beendigung der Transaktion bekannt gemacht werden.

- *Dauerhaftigkeit*: Nach der Festschreibung einer Transaktion sind die vorgenommenen Änderungen persistent und können nicht mehr zurückgenommen werden. Diese Dauerhaftigkeit wird üblicherweise durch Integration der verteilten Objekte mit dem Persistenzdienst erreicht, der im Abschnitt 3.3 besprochen wird. Der Transaktionsmanager nutzt den Dienst, die Zustände aller an der Transaktion beteiligten Objekte persistent zu speichern.

Die ACID-Eigenschaften verteilter Transaktionen sind abhängig von der Implementierung der Teilnehmer. Um die ACID-Eigenschaften zu erfüllen, muss das System sicherstellen, dass kein Teilnehmer gegen irgendeine

Eigenschaft verstoßen kann. Im nächsten Abschnitt wird das Zwei-Phasen-Commit-Protokoll näher vorgestellt.

3.5.2 Das Zwei-Phasen-Commit-Protokoll (2PC)

Rollen in verteilter Transaktion

Bei einer verteilten Transaktion, die mit dem Zwei-Phasen-Commit-Protokoll ausgeführt wird, sind üblicherweise die folgenden Rollen beteiligt:

- *Transaktionsclient*: Die Anwendung, die eine neue Transaktion vom Transaktionsdienst erhält und sie verwendet und schließlich beendet, wird allgemein als Transaktionsclient bezeichnet.

- *Transaktionsserver*: Ein Transaktionsserver ist anwendungsspezifisch und häufig zustandsabhängig. Wird eine Operation vom Transaktionsserver innerhalb einer Transaktion ausgeführt, muss der Transaktionsserver sich an der Transaktion beteiligen, indem er sich beim Transaktionskoordinator anmeldet. So bezeichnen wir diesen als Teilnehmer. Jeder Teilnehmer ist dafür verantwortlich, alle wiederherstellbaren Objekte auf diesem Server zu verwalten, die an der Transaktion beteiligt sind. Er kümmert sich auch darum, mit dem Koordinator zusammenzuarbeiten, um das Zwei-Phasen-Commit-Protokoll auszuführen.

- *Transaktionskoordinator*: Der Transaktionskoordinator ist dafür verantwortlich, neue Transaktionen zu erstellen und sie nach der Ausführung festzuschreiben oder abzubrechen. In verteilten Systemen gibt es normalerweise viele Transaktionskoordinatorobjekte, weil ansonsten ein neuer Engpass entstehen würde.

Ablauf einer verteilten Transaktion

Der Transaktionsclient startet eine Transaktion, indem ein Transaktionskoordinator diese Transaktion öffnet. Unter Verwendung dieser Transaktion führt der Client eine Folge von Operationen aus, wobei betroffene Transaktionsserver sich an der Transaktion beteiligen. Inzwischen gibt es keine Kommunikation zwischen dem Koordinator und den Teilnehmern, außer dass die Teilnehmer den Koordinator darüber informieren, wenn sie der Transaktion beitreten. Die Anforderung des Clients, diese Transaktion festzuschreiben oder abzubrechen, wird an den Koordinator weitergeleitet. Fordert der Client „abort“ an, benachrichtigt der Koordinator die Teilnehmer sofort. Fordert der Client „commit“ an, kommt das Zwei-Phasen-Commit-Protokoll ins Spiel.

Das Zwei-Phasen-Commit-Protokoll besteht aus einer Abstimmphase und einer Abschlussphase. In der ersten Phase wird festgestellt, ob eine Transaktion beendet werden kann oder nicht. Dies erfolgt durch Abstimmung. Dabei wird vorausgesetzt, dass während der Ausführung der Transaktion der Koordinator eine Liste aller Verweise auf die Teilnehmer und jeder Teilnehmer einen Verweis auf den Koordinator aufzeichnet. Um

die Abstimmungsphase zu implementieren, befragt der Koordinator alle Teilnehmer an der Transaktion nach ihrer Stimme. Bevor ein Teilnehmer für ein Commit stimmt, bereitet er sich auf das Festschreiben vor, indem er Objekte im permanenten Speicher ablegt. Hat jeder einzelne beteiligte Transaktionsserver dem Festschreiben zugestimmt, kann die Transaktion erfolgreich beendet werden. Dagegen bricht die Gegenstimme irgendeines Teilnehmers die Transaktion ab.

Zunächst wird der Abschluss gemäß des Abstimmergebnisses implementiert. Nachdem der Transaktionskoordinator von allen Teilnehmern ihre Stimmen gesammelt hat, vergleicht er die Stimmabgaben. Wenn alle Teilnehmer für das Commit gestimmt haben, entscheidet der Transaktionskoordinator, die Transaktion festzuschreiben und stellt an jeden der Teilnehmer eine Commit-Anfrage. Andernfalls entscheidet der Koordinator, die Transaktion abzubrechen und sendet Abbruch-Anforderungen an alle Teilnehmer, die für das Commit gestimmt haben. Teilnehmer, die für das Commit gestimmt haben, warten auf eine Commit- oder Abbruch-Anforderung vom Transaktionskoordinator. Empfängt ein Teilnehmer eine dieser Nachrichten, handelt er entsprechend.

Beispielweise stellt das Sequenzdiagramm, wie in Abbildung 3-3 gezeigt, das Zwei-Phasen-Commit-Protokoll einer verteilten Transaktion dar, die sich auf die Verlegung eines Tupels zwischen zwei Tupelräumen bezieht, die von verschiedenen Servern verwaltet werden. Zunächst fordert der Transaktionsclient den Start einer neuen Transaktion beim Transaktionskoordinator an. Dann ruft der Transaktionsclient die Operation *in()* und *out()* der beiden Tupelraum-Objekte auf, welche die Transaktionsserver sind. Die Implementierungen dieser beiden Operationen sind transaktionsfähig gestaltet und registrieren sich daher beim Transaktionskoordinator mit der Operation *join()*. Danach fragt der Transaktionsclient das Beenden der Transaktion an. Die Implementierung des Commit sieht vor, dass der Transaktionskoordinator zunächst anhand der Operation *prepare()* die beiden Tupelraum-Objekte nach ihrer Stimmabgabe bezüglich eines Commit befragt. In diesem Beispiel stimmen beide Transaktionsserver für ein Commit und der Transaktionskoordinator fordert von ihnen die Operation *commit()* an. Anschließend zeigt der Transaktionskoordinator dem Client den erfolgreichen Abschluss der Transaktion an und gibt die Kontrolle an diesen zurück.

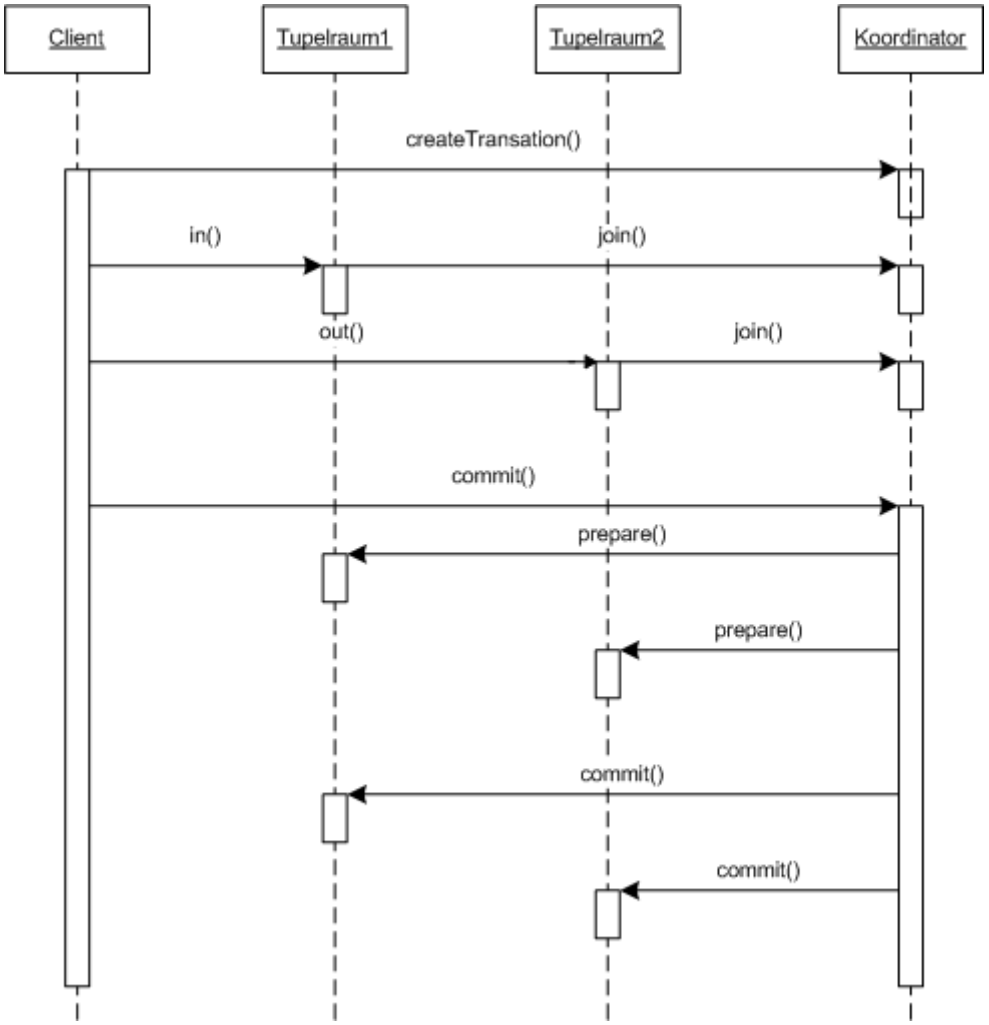

Abbildung 3-3: Sequenzdiagramm einer beispielhaften Transaktion

Das obige Beispiel zeigt, dass die Implementierung der Transaktion für den Client transparent ist. Er muss die Transaktion nur starten und beenden. Für die Entwickler der Transaktionsserver ist die Implementierung der Transaktion jedoch nicht transparent. Sie müssen die für das Zwei-Phasen-Commit-Protokoll notwendigen Operationen wie *prepare()* und *commit()* implementieren.

Wiederherstellung

Treten die Fehler während einer Transaktion auf, muss das Zwei-Phasen-Commit-Protokoll darauf richtig reagieren und ggf. eine Wiederherstellung durchführen können. Im Folgenden werden die verschiedenen Fehlerfälle und die entsprechend ergriffen Maßnahmen erläutert:

- *Fehler vor dem Beenden*: Wenn irgendein Teilnehmer an einer Transaktion, einschließlich des Transaktionsclients, vor der Commit-Anfrage fehlschlägt, wird der Transaktionskoordinator die Transaktion schließlich abbrechen, indem er explizit Abbruch-Anforderung an die beteiligen Server sendet.

- *Fehler bei den Teilnehmern vor der Abstimmphase*: Fällt irgendein Teilnehmer vor der Abstimmphase aus, interpretiert der Transaktionskoordinator den Ausfallsfehler der Stimme des Teilnehmers als Gegenstimme und bricht die Transaktion ab. Wenn der fehlschlagende beteiligte Server vor der Abstimm-Anfrage wiederhergestellt ist, ist der Ausfallsfehler für den Transaktionskoordinator zu maskieren.

- *Fehler beim Transaktionskoordinator während der Abstimmphase*: Stürzt der Koordinator vor oder während der Abstimmphase ab, empfängt die Transaktion niemals eine Commit-Anfrage und bricht schließlich ab, es sei denn, dass der Koordinator wiederhergestellt wird, bevor für den Transaktionsclient oder für die Transaktion ein Timeout auftritt.

- *Fehler bei den Teilnehmern nach der Abstimmphase*: Fällt ein Teilnehmer nach der Abgabe seiner Stimme aus, fragt dieser nach einem Neustart den Transaktionskoordinator nach der getroffenen Entscheidung bezüglich des Transaktionsabschlusses. War die Entscheidung für das Commit, wird dieses mit den wiederhergestellten Daten durchgeführt. War die Entscheidung gegen das Commit, wird er abbrechen.

- *Fehler beim Transaktionskoordinator während der Abschlussphase*: Stürz der Koordinator während der Abschlussphase ab, muss er die persistent gespeicherten Informationen verwenden, um die Commit-Anforderungen an alle Teilnehmer erneut zu senden.

3.6 Kooperationsmodul: Tupelräume-Dienst

Wir haben im Abschnitt 2.3 besprochen, dass das Tupelraum-Konzept eine einfache und flexible Lösung zur Kooperation zwischen verteilten Prozessen ist. Ein Tupelraum bietet wie ein schwarzes Brett einen gemeinsam genutzten Platz, wo man seine zu veröffentlichenden Nachrichten aufschreiben, die einem interessierenden Nachrichten erfahren und ggf. diese entfernen kann, dadurch sind zwei beliebige miteinander unbekannte Prozesse in der Lage, für die Kommunikation genutzte Nachrichten untereinander auszutauschen und somit eine gemeinsame Aufgabe zu koordinieren.

Jede Nachricht hat potenzielle Empfänger, die normalerweise nur auf eine bestimmte Menge von Prozessen beschränkt sind. Umgekehrt hat ein Prozess vielleicht nur Interesse an gewissen Nachrichten. So ist es sinnvoll, mehrere Tupelräume, welche die eingegangen Nachrichten verteilen können, zur Verfügung stellen zu können. Jedoch sind die Verteilung der Nachrichten und die Organisation aller Tupelräume, welche Tupelräume sich um welche Arten von Nachrichten kümmern, total anwendungsspezifisch.

Der Tupelräumen-Dienst ist verantwortlich für das Erstellen und Löschen eines Tupelraums. Jeder erstellte Tupelraum ist einen im Dienst eindeutigen Namen zugeordnet und als ein selbstständiger Server verfügbar. Über diesen Namen kann der Client den Tupelraum finden und auf den Tupelraum mit den angegeben Operationen zugreifen, dies wird im nächste Abschnitt erläutert.

3.6.1 Operationen an einen Tupelraum

3.6.1.1 Grundoperationen

3.6.1.1.1 Schreiben (out)

Die Schreiboperation ist dafür verantwortlich, den vom Client übertragenden Tupel in den Tupelraum abzulegen. Der Tupelraum überprüft nicht, ob der eingegangene Tupel bereits vorhanden ist. Deshalb könnten mehrere Tupel mit gleichem Inhalt existieren. Verbrauchen keine Clients diesen Tupel, behält der Tupelraum den Tupel für immer.

Haben die Clients sich angemeldet, um bestimmte Tupel zu erwarten, benachrichtigt der Tupelraum nach der Schreiboperation diejenigen Clients, die genau auf den eben abgelegten Tupel warten.

3.6.1.1.2 Lesen (read und in)

Zu verschiedenen Zwecken sind zwei Leseoperationen verfügbar. „read“ durchsucht den Tupelraum nach einem Tupel, welcher der gegebenen Vorlage entspricht. Wenn ein passender Tupel gefunden ist, wird eine Kopie davon zurückgegeben. Die Operation „in“ führt die gleiche Aktion aus und entfernt aber noch den gefundenen Tupel vor der Zurückgabe.

Die Operation „**read**“ liest einen passenden Tupel aus dem Tupelraum, wobei der Tupelraum sich nicht ändert. Jedoch ist „read“ keine idempotente Operation, da der Tupelraum nicht garantieren kann, ob sich derselbe Tupel ergibt, wenn die Operation „read“ zweimal nacheinander ausgeführt wird und es mehrere passende Tupel gibt, obwohl der Tupelraum inzwischen vielleicht überhaupt keine Änderung vorgenommen hat.

Für einen Tupel, der mit XML dargestellte riesige Daten enthält, benötigt der Client häufig nur ein Teil davon. Um die Kommunikationsdaten zu verringern, macht es Sinn, das partielle Lesen zu unterstürzen. Zu diesem Zweck wird XPath in der Vorlage verwendet. Anhand eines XPath-Ausdrucks ist beliebig ein Teil des XML-Dokuments lesbar.

Die Operation „**in**“ entnimmt einen passenden Tupel aus dem Tupelraum. Aufgrund des ähnlichen Problems wie oben diskutiert erhält der Client die gewünschten Tupel nicht immer in derselben Reihfolge, wenn er die gleiche Operation ausführt und es mehrere Tupel gibt, welche der gegebenen Vorlage entsprechen.

Alle Leseoperationen können ggf. ihren Clientprozess blockieren, wenn die erwarteten Tupel im Tupelraum nicht existieren. In diesem Fall wird der Proxy auf der Clientseite sich automatisch beim Tupelraum-Server registrieren und auf eine Nachricht vom Tupelraum warten. Diese Nachricht gibt an, dass die erwarteten Tupel vorhanden sind. Tritt ein Timeout auf, erhält der Client einen leeren Verweis und setzt seine Arbeit fort.

3.6.1.2 Erweiterte Operationen

Neben den Grundoperationen, die das Linda-Modell unterstürzt, wird dies Tupelraum noch um einige Operationen erweitert, um die im Abschnitt 2.3.4 besprochenen Beschränkungen des Linda-Modells zu beseitigen.

3.6.1.2.1 Synchronisiertes Lesen (sync)

Die besprochenen Leseoperationen ermöglichen den Zugriff auf einem einzelnen Tupel im Tupelraum. Auf der Clientseite kann man durch Verwendung einer Transaktion einige solche atomare Leseoperationen miteinander verknüpfen, um mehrere Tupel synchronisiert zu lesen. Allerdings wenn mehrere Clientprozesse derartige Transaktion ausführen, muss es mit einer Überlappung der Transaktionsanforderungen wie Abbildung 3-4 gezeigt, gerechnet werden.

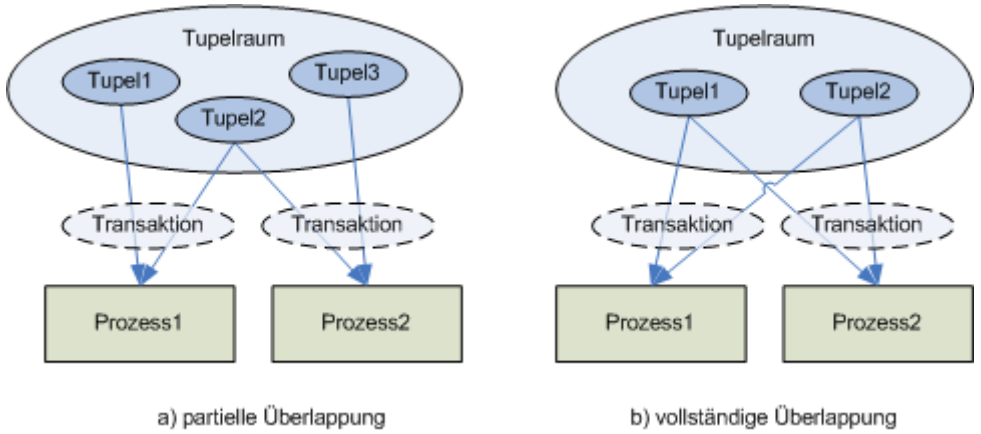

Abbildung 3-4: Überlappung der Transaktionsanforderungen

- **Einzelne Überlappung** wird in Abbildung 3-4 a) veranschaulicht. Die beiden Prozesse fordern den Tupel2 innerhalb der eigenen Transaktion an. Wenn der Tupel2 ankommt, verbraucht entweder der Prozess1 oder der Prozess2 ihn. So entsteht ein Überlappungspunkt, der dazu führt, dass nur ein Prozess seine Transaktion erfolgreich festschreiben kann. Die Transaktion des anderen Prozesses ist nicht mehr ausführbar und muss dann ablaufen und ein Rollback vornehmen.
- **Mehrfache Überlappung** Falls mehrere Überlappungspunkte existieren, könnten die beiden Prozesse jeweils ein Teil der sich überlappenden Tupel verbrauchen, sodass sie beide ihre Transaktionen nicht beenden können. Es erfolgt ein Deadlock, das durch zufällige Timeout gelöst wird. Die vollständige Überlappung wird in Abbildung 3-4 b) gezeigt und ist bloß ein Sonderfall der mehrfachen Überlappung, außer dass alle Vorlagen sich überlappen.

Das Überlappungsproblem kann man durch Timeout und wiederholte Versuche zwar auf der Clientseite lösen, aber aufgrund der Zunahme des Netzverkehrs zwischen dem Client und dem Server erkauft diese Lösung viele Performanceverluste. Zur Optimierung sollte der Server eine zusätzliche Operation bieten, die mehrere atomare Leseoperationen ausführen kann, damit die Synchronisierungssemantik auf der Serverseite implementiert wird. Die von den Überlappungen verursachten Konflikte sind nach einem gewissen Algorithmus zentral zu lösen.

Außerdem kann eine Gruppe atomarer Leseoperationen nicht eine Menge Tupeln finden, die über einen oder mehrere unbekannten Attributen zusammenhängen. Zum Beispiel wollte der Client zwei Tupel, die zum selben Prozessinstanz gehören, lesen, aber für den Client ist der Prozessinstanz noch unbekannt. Durch zwei getrennte Leseoperationen ist es unmöglich solchen Zusammenhang der erhaltenen Ergebnisse festzustellen.

Deshalb erweitern wir die Tupelraum-Schnittstelle um eine spezielle Operation *sync*, [22] die im Hintergrund angegebene Leseoperationen aufruft und im Wesentlichen ausschließlich die Synchronisierung auf der Serverseite implementiert. Mit dieser Operation können mehrere Clientprozesse gleichzeitig versuchen, ihre gewünschte Tupel synchronisch zu lesen oder zu entnehmen, ohne wegen den Zugriffskonflikten zum Deadlock zu führen.

Der Algorithmus teilt sich in zwei Phasen: Anfragephase und Wartephase. Der Client übergibt eine Liste von synchronisiert auszuführenden Leseoperationen als Parameter an *sync*. Die angegebenen Vorlagen könnten über ein oder mehrere Attributen zusammenhängen. Beispielweise betrachten wir die zwei Vorlagen („abc“, 123, null?) und („def“, 123, null?), die über das letzte Attribut zusammenhängen sollen. Dann muss das dritte Attribut des Tupels, der der ersten Vorlage entspricht, sich mit dem des Tupels, der der zweiten Vorlage entspricht, übereinstimmen.

Nachdem der Server diese Anfrage akzeptiert hat, fängt er mit der Anfragephase an und versucht er zuerst, den ganzen Tupelraum zu durchsuchen. Daraus erfolgen zwei Fälle:

- Werden alle gesuchte Tupel gefunden, führt der Server gleich die übergebenen Operationen aus und der Client erhält deren Ergebnisse als Antwort. In diesem Fall wird nur die erste Phase durchlebt.

- Werden keine oder nur ein Teil davon gefunden, stellt der Server den Clientprozess in eine Warteschlange und dann der Client wartet auf die Benachrichtigung vom Server. Damit wird die Wartephase gestartet. Da jede nachher auszuführende Schreiboperation einen vorlagenkonformen Tupel hinzufügen könnte, sucht der Server alle Anforderungen der Clientprozesse in der Warteschlange nach der Ausführung jeder Schreiboperation durch. Sobald die Anforderung an den gesuchten Tupeln erfüllt und der Clientprozess ausgewählt ist,

führt der Server die vorher mit gespeicherten Operationen aus und benachrichtigt den Clientprozess mit den Ergebnissen.

Mehrere Clientprozesse könnten nach der Ausführung von *sync* synchron blockiert werden. Überlappen sich die gesuchten Tupel ganz oder teilweise, muss der Server diese Prozesse nach einem gewissen Prinzip sortieren, sodass sie nacheinander ausgewählt werden, wenn ihre Anforderungen erfüllt sind. Der Einfachheit halber übernehmen wir die Ausführungszeit als den Sortierungsschlüssel. Stehen die gesuchten Tupel zur Verfügung, bearbeitet der Server vorrangig die Anforderung des Clientprozesses, dessen sync Operation am frühsten beim Server ausgeführt wird.

3.6.1.2.2 Modifizieren (update)

In dem traditionellen Tupelraum-Konzept gibt es keine Modifikationsoperation. Um einen vorhanden Tupel zu bearbeiten, muss der Client erst den Tupel entnehmen und nach der Bearbeitung diesen wieder in den Tupelraum ablegen. Dabei ist viele Kommunikation mit dem Tupelraum nötig, die eine Menge Zeit verbraucht. Deshalb erweitern wir die Tupelraum-Schnittstelle um eine Modifikationsoperation „update“, die direkt mit den gegebenen Parametern den entsprechenden Tupel auf der Serverseite bearbeiten kann.

Ähnlich wie „read“ unterstürzt „update“ auch die partielle Modifizierung durch XPath. Anstatt eines vollständigen Tupels muss der Client nur die entsprechende Vorlage mit dem XPath-Ausdruck und den Tupel mit dem zu ersetzenden Teil übergeben. Zum Beispiel wird es gewünscht, dass das Tupel („name“, 1203, „<note><to>Vetter</to><body>Wichtige Nachrichten</body></note>") auf („name“, 1203, „<note><to>David</to><body>Wichtige Nachrichten</body></note>") geändert wird. Dann muss nur die Operation „update“ mit den Parametern, nämlich der Vorlage („name“, 1203, „/note/to“) und dem Tupel („name“, 1203, „<to>David</to>“), ausgeführt werden.

Die Änderungen, welche die Operation „update“ verursacht hat, könnten auch die Benachrichtigung aktivieren, wenn die bearbeiteten Tupel dadurch den vom Client Erwarteten entsprechen.

3.6.1.2.3 Untersuchen (scan)

Aufgrund des Problems in 2.3.4.2 erweitert sich die Schnittstelle um die Operation „scan“. Sie sucht in einem Tupelraum nach den Tupeln, die der gegebenen Vorlage entsprechen, und erhält deren Anzahl. Sie wird üblicherweise ausgeführt, wenn der Client sich vor der Ausführung anderer Operationen darüber informieren will, wie viele gewünschte Tupel im Tupelraum vorhanden sind. Das zeigt deutlich, dass sie eine idempotente Operation ist.

3.6.1.3 Zeitkomplexität

Die Zeitkomplexität aller Operationen kann auf der Anzahl von RMI Aufrufen, und der Ausführungszeit im Server zurückgeführt werden. Nächstens werden die beiden Aspekte jeweils diskutiert.

Anzahl von RMI Aufrufen

Zuerst wird es angenommen, dass weder der Netzausfall noch der Serverabsturz geschieht, da im Fehlerfall aufgrund der Wiederholung der Anforderungsnachrichten die Anzahl der notwendigen RMI Aufrufe nicht mehr bestimmt werden kann. Wie in Tabelle 3-1 gezeigt, brauchen alle Operationen genau einen RMI Aufruf ohne die Rückmeldung und Blockierung zu berücksichtigen. Die Rückmeldung erfordert einen zusätzlichen Aufruf.

	Ohne BLK Ohne ACK	Ohne BLK Mit ACK	Mit BLK Ohne ACK	Mit BLK Mit ACK
out	1	2	-	-
in	1	2	5+	7+
read	1	2	5+	7+
sync	1	2	2	4
update	1	2	-	-
scan	1	2	-	-

BLK: Blockierung ACK: Rückmeldung

Tabelle 3-1: Anzahl der notwendigen RMI Aufrufe für jede Tupelraum-Operation

Während der Ausführung von „out“, „update“ und „scan“ wird die Blockierung nie auftreten. Dagegen benötigen alle Leseoperationen ggf. die Blockierung zum Warten auf die gewünschten Tupel. Wie oben besprochen, registrieren „in“ und „read“ sich beim Server durch die entsprechende Methode, falls die vorlagenkonformen Tupel nicht existieren. Nach dem Aufwachen durch die Benachrichtigung vom Server führt der Client sie noch mal aus. Jedoch könnte der angekommene Tupel wieder entfernt werden. Wenn der Client endlich seinen gewünschten Tupel bekommen hat, muss er sich noch beim Server abmelden. Zusammengefasst brauchen die beiden Leseoperationen mindestens fünf RMI Aufrufe.

„sync“ verhält sich anderes als „in und „read“. Während der erstmaligen Ausführung im Server meldet sie sich automatisch an, falls keine angepassten Tupelsatz gefunden sind. Dann wartet der Client auf die Benachrichtigung vom Server. Sobald die gewünschten Tupelsatz ankommen, wird die Operation „sync“ auf der Serverseite ausgeführt und die Ergebnisse werden an dem Client zurückgegeben. Im Vergleich zum Fall ohne Blockierung ist bloß ein zusätzlicher Aufruf zur Benachrichtigung erforderlich.

Ausführungszeit im Server

Hier wird unter der Ausführungszeit die Zeit zur Erledigung einer Operation im Server verstanden. Die Tabelle 3-2 zeigt den Komplexitätsgrad der Ausführungszeit aller Tupelraum-Operationen. Die wichtigen Variablen sind die Anzahl der vorhandenen Tupel und die Anzahl der vom Client registrierten Vorlagen. Außer den Operationen „read“ und „scan“, die das Tupelraum nur lesen, können alle anderen Operationen

den Zustand des Tupelraums ändern. Jede Änderung aktiviert den Überprüfungsvorgang, der alle registrierten Vorlagen durchprobiert, ob der geänderte Tupel einer entspricht.

	Ausführungszeit
out	c1 + c2 * p
in	c2 * n + c2 * p
read	c2 * n
sync	c2 * n * m1 + c2 * p * m2
update	c2 * n + c2 * p
scan	c2 * n

n: die Anzahl der vorhandenen Tupel
m1: die Anzahl der von „sync“ beinhalteten Operationen
m2: die Anzahl der von „sync“ beinhalteten Operationen „in“
p: die Anzahl der vom Client registrierten Vorlagen
c1: die durchschnittliche Zeit zum Hinzufügen eines Tupels
c2: die durchschnittliche Zeit zum Tupel-Matching

Tabelle 3-2: Ausführungszeit aller Tupelraum-Operationen

„out” fügt einen Tupel mit einer konstanten Zeit c1 zu und durchsucht alle registrierten Vorlagen. Die Ausführungszeit steigt linear mit p an.
„in” und „update“ durchsuchen erst das ganze Tupelraum und weiterhin noch alle registrierten Vorlagen. Üblicherweise ist n viel größer als p. Im Vergleich zu n kann p ignoriert werden. So steigt die Ausführungszeit linear mit n an.
„read” und „scan“ durchsuchen bloß das ganze Tupelraum. Die Ausführungszeit steigt linear mit n an.
„sync” beinhaltet m1 atomare Leseoperationen, darin es m2 „in“ gibt. Durch Kombination der Ausführungszeiten aller atomaren Operationen ergibt sich der obere Ausdruck c2*n*m1+c2*p*m2. Normalerweise sind m1 und m2 sowie p relativ nur kleine Zahlen und können als Konstant betrachtet werden. Damit steigt die Ausführungszeit auch linear mit n an.

3.6.2 Transaktion

3.6.2.1 Zweiphasen-Sperrverfahren

Das Zweiphasen-Sperrverfahren ist das verbreiteteste Verfahren zur Nebenläufigkeitskontrolle. Die Grundidee dahinter lautet, dass Transaktionen eine gemeinsame Ressource beim Nebenläufigkeitsmanager anfragen. Sie erwerben Sperren, bevor sie eine Ressource nutzen. Hier verstehen wir unter Ressourcen die Tupel und unter Nebenläufigkeits-Manager den Tupelraum-Server. Die Nebenläufigkeitskontrolle bewilligt eine Sperre nur dann, wenn die Verwendung des Tupels nicht mit Sperren in Konflikt gerät, die zur anderen nebenläufigen Transaktion bewilligt worden sind. Wird ein Tupel von der Transaktion nicht mehr benötigt, wird die Sperre aufgehoben, so dass sie von anderen Transaktionen erworben werden kann.

Sperrprofil

Das Zweiphasen-Sperrverfahren fordert, dass eine Transaktion keine weiteren Sperren erwerben kann, sobald sie eine ihrer Sperren zurückgegeben hat. Die Transaktion kann die Sperre eines Tupels erst aufheben, wenn es alle benötigten Sperren bereits besitzt und der Tupel nicht mehr benötigt wird. Es ist wünschenswert, Sperren so früh wie möglich aufzuheben, damit nebenläufige Transaktionen, welche die Tupel benötigen, die entsprechenden Sperren erwerben können. Weil die Entscheidung, ob das Aufheben einer Sperre sicher ist, schwer zu treffen ist, behalten Transaktionen hier ihre Sperren bis zum Schluss und heben sie erst gemeinsam auf.

3.6.2.2 Operationen unter Transaktionen

Jede Operation ohne Transaktion verhält sich genau sowie dieselbe Operation unter einer Transaktion, die schließlich festgeschrieben ist. Beispielweise wird eine Operation „out“ ohne Transaktion genauso ausgeführt, wie als ob sie nachdem eine Transaktion erstellt wurde, unter dieser ausgeführt wird.

- *out:* Wird diese Operation innerhalb einer Transaktion ausgeführt, ist der zu schreibende Tupel sichtbar, erst wenn die Transaktion festgeschrieben ist. Ist die Transaktion wegen eines Fehlerfalls abgebrochen, wird der Tupel schließlich nicht in den Tupelraum hinzugefügt. Die mögliche Benachrichtigung wird nach dem Festschreiben der Transaktion aktiviert.

- *read:* Unter einer Transaktion wird der gelesene Tupel solange blockiert, bis die Transaktion abgebrochen oder festgeschrieben ist. Inzwischen dürfen nur die Prozesse innerhalb der Transaktion auf diesen Tupel zugreifen.

- *in:* Den innerhalb einer Transaktion bereits entnommenen Tupel dürfen andere Prozesse außerhalb der Transaktion nicht sehen, es sei denn, dass die Transaktion abgebrochen ist.

- *update:* Während der Ausführung einer Transaktion sind die Änderungen, die „update“ vorgenommen hat, für die Prozesse außerhalb der Transaktion unsichtbar. Sie werden gültig, nachdem die Transaktion festgeschrieben ist. Danach erfolgt die Benachrichtigung, falls die geänderten Tupel die Erwartung des Clients erfüllen.

3.7 Monitoring

In dieser Arbeit wird unter Monitoring der Vorgang verstanden, die Zustände der Dienste innerhalb dieser Middleware zur Laufzeit zu überwachen. Damit kann festgestellt werden, ob die Middleware richtig funktioniert.

Die Monitore können passiv oder aktiv die gewünschten Daten von den entsprechenden Diensten erhalten. Bei passivem Monitoring muss der Zustand des überwachten Diensts erfragt werden, bei aktivem Monitoring propagiert der überwachte Dienst selbständig seinen Zustand. Die Monitore präsentieren die erhaltenen Daten, indem sie diese Daten in Textform oder graphischer Form abbilden.

Die Middleware umfasst den Transaktionsdienst und den Tupelräume-Dienst. In jeden Dienst wurde ein aktiver Monitor integriert, durch den der Dienst selbständig seinen Zustand propagieren kann.

Transaktionsdienst

Um den Ablauf einer Transaktion, die vom Transaktionsdienst erstellt ist, zu verfolgen, muss der Zustand der Transaktion jederzeit während der Ausführung abrufbar sein. Der Übergang des Transaktionszustandes verkörpert eine Operation der Transaktion. Damit ist der Lebenszyklus der Transaktion zu überwachen. Übrigens ist die Anzahl der laufenden Transaktionen ein wichtiger Parameter zur Belastungsrechnung für den Transaktionsdienst.

Üblicherweise darf der Ablauf einer Transaktion außer ihrem Client und ihren Teilnehmern sowie dem Transaktionsdienst nicht beeinflusst werden, ansonst würden die ACID-Eigenschaften beschädigt. Deshalb kann der Monitor die Zustände aller Transaktionen zeigen und sie aber nicht ändern.

Tupelräume-Dienst

Der Zustand eines Tupelräume-Dienstes umfasst die Liste aller aktuell vorhandenen Tupelräume und den Inhalt jedes Tupelraums. Üblicherweise ist die Erste selten zu ändern und nur der Administrator kann neuen Tupelraum erstellen und existierenden Tupelraum entfernen. Was den normalen Client interessiert, ist die Änderung des Inhalts vom Tupelraum. Im Fehlerfall spielt dieser Monitor eine speziell wichtige Rolle, wenn er noch läuft. Er hilft dabei, rechtzeitig das Problem zu entdecken und weiterhin festzustellen, woran es liegen könnte.

3.8 Realisierung

3.8.1 Verwendete Technologien

3.8.1.1 Prevayler

Prevayler ist eine java-basierte Bibliothek für Objektpersistenz. Es ist die erste Implementierung der Konzepte der Object Prevalence, die im Jahre 2001 von Klaus Wuestefeld entwickelt wurde. Prevayler ermöglicht ohne Verwendung von Datenbanken einen effizienten Persistenzdienst anzubieten. Die auf der Webseite [18] veröffentlichten Geschwindigkeits-Vergleiche zwischen Prevayler und Relationalen Datenbank Management-Systemen (RBDMS) erregen große Aufmerksamkeit. Im Vergleich wird Prevayler 3000- bis 9000-mal schneller als bekannte RBDMS wie MySQL oder Oracle ausgewiesen.

Der Grundidee von Prevayler ist hierbei, dass aktuelle Daten im RAM aufbewahrt und ihre Änderungen in Log-Dateien protokolliert werden. Die Geschäftsobjekte der Anwendung werden in einem Java Objekt, auch als Prevalent-System bezeichnet, verkapselt. Durch Aggregation von Unterobjekten entspricht die Datenstruktur eines Prevalent-Systems einem Baum. Jedes Objekt auf dem Baum muss serialisierbar sein, sodass das gesamte Prevalent-System serialisiert werden kann.

Prevayler dient als eine transaktionale Absperrung der Geschäftsobjekte. Für den Client sind die Objektzugriffe möglich nicht direct durch verfügbare Operationen, sondern indirekt durch Nutzung des Command Pattern realisiert. Die Idee dieses Patterns liegt darin, dass jede Operation in einem einfachen Objekt verkapselt wird. Entsprechend existieren zwei Arten vom Commands. Query Commands sind zum lesenden Zugriff bestimmt während Transaction Commands zum modifizierenden Zugriff genutzt werden können. Anderes als SQL-Ausdrücke im RBDMS werden Commands durch Zugriffscode implementiert, die für Prevayler in Java programmiert wird. Als statisch typisierte Sprache ergibt sich für Java ein Vorteil, dass man die Syntaxüberprüfung zur Kompilationszeit ausführen kann, während Fehler in der SQL-Syntax oft nur zur Laufzeit in Sicht kommen. Anderseits bietet SQL als dynamische und für Datenzugriff bestimmte Sprache mehr Flexibilität bei den Administrationsaufgaben. Bei komplexen Abfragen, die Aggregation, Sortierung oder Joins beinhalten, erfordert die Programmierung in Java zumeist auch erheblich mehr Code als bei Verwendung der Sprache SQL der Fall ist [19].

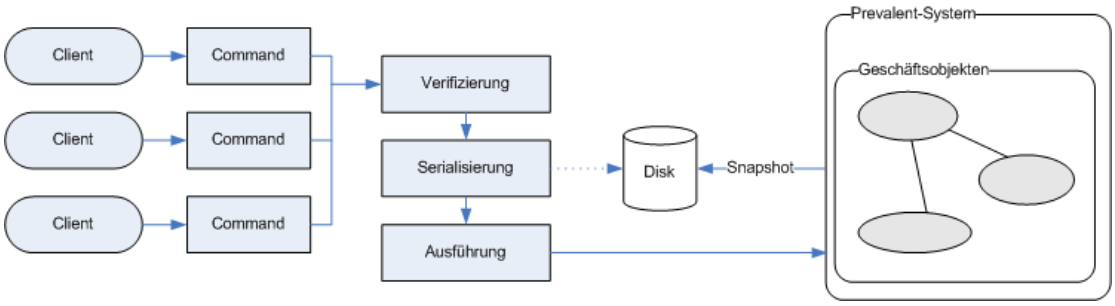

Abbildung 3-5: Funktionsweise von Prevayler

In der Abbildung 3-5 wird die Funktionsweise von Prevayler erläutert. Um eine Operation an einem oder mehreren Geschäftsobjekten durchzuführen, muss der Client das Command, in dem die Operation verkapselt wird, ausführen. Nachdem Prevayler das Command erhalten hat, verifiziert es zuerst deterministisches Verhalten des Commands durch die sogenannte Probe-Ausführung. Dabei ist es zu bemerken, dass nach der Probe-Ausführung das Prevalent-System unverändert bleibt, indem es auf seinen letzten Zustand zurückgeführt wird. Allerdings wenn die Implementierung des Commands eine Operation an externen Ressourcen aufruft, ist dieser Aufruf nicht zurücksetzbar und bei der normalen Ausführung des Commands erfolgt dann zweiter Aufruf der Operation, was zur unerwarteten Konsequenz führen könnte. Deshalb ergibt sich eine Anforderung an der Implementierung des Commands, dass die

Datenzugriffe sich nur auf das entsprechende Prevalent-System beschränken sollen.

Der Fehlschlag der Verifizierung würde zur Beendigung der Ausführung des Commands mit einer Ausnahme führen. Ansonst protokolliert Prevayler das Command auf einer Log-Datei, bevor es wirklich ausgeführt wird, damit die Daten nicht verloren gehen, wenn das System abstürzt. Prevayler kann also jederzeit ein Snapshot des Prevalent-Systems abspeichern. Beim erneuten Start des Servers benutzt Prevayler das aktuellste Snapshot mit der Log-Datei dazu, die Geschäftsobjekte automatisch wiederherzustellen, dadurch, dass die nach der Aufnahme des letzten Snapshot aufgezeichneten Commands noch mal ausgeführt werden. [18]

3.8.1.2 Java/RMI

3.8.1.2.1 Architektur

Der Java-spezifische Ansatz der *Remote Method Invocation* (RMI) zielt darauf einen lokationstransparenten Aufruf innerhalb von (verteilten) Java-Anwendungen zu ermöglichen. Die wesentliche Idee besteht darin, Kommunikation innerhalb einer verteilten Anwendung als Methodenaufrufe an entfernten Objekten zu modellieren. Idealerweise gestaltet sich ein entfernter Methodenaufruf dabei genauso wie ein lokaler Methodenaufruf.

Der Methodenfernaufruf ist ein asymmetrischer Mechanismus: ein Server stellt ein entferntes Objekt bereit, während ein Client an diesem Objekt Methoden aufruft.

Wenn ein Client eine Methode am entfernten Server aufruft, wendet er sich in Wirklichkeit an ein lokales Stellvertreterobjekt des Servers, den sogenannten Serverproxy oder auch Serverstub. Dieses Objekt befindet sich auf der Clientmaschine und bietet dieselbe Schnittstelle an wie das entfernte Serverobjekt. Der Serverproxy konvertiert die Methodenparameter in ein geeignetes Transportformat – der Java-RMI-Mechanismus verwendet dafür Objektserialisierung – und schickt sie zusammen mit dem Methodennamen zur Servermaschine. Das Konvertieren der Parameter nennt man auch Marshalling.

Auf der Serverseite wird der Aufruf von einem sogenannten Serverskeleton empfangen. Der Skeleton konvertiert die Parameter zurück in ihr Ausgangsformat, indem es sie deserialisiert. Dieser Vorgang wird als Demarshalling bezeichnet. Der Serverskeleton reicht den Aufruf dann weiter an das eigentliche Serverobjekt, das ihn entgegennimmt und das nach erfolgter Bearbeitung das Ergebnis zurückliefert an den Skeleton. Nun übernimmt der Serverskeleton das Marshalling des Ergebnisses und verschickt das so serialisierte Ergebnis zurück an den Serverproxy auf der Clientseite.

Der Proxy führt das Demarshalling des Ergebnisses durch und reicht es weiter an den Client. Sowohl Server als auch Client haben beide nur mit

lokalen Objekten kommuniziert; Der Serverproxy simuliert dabei das Serverobjekt auf der Clientseite, während das Serverskeleton den Client auf der Serverseite simuliert.

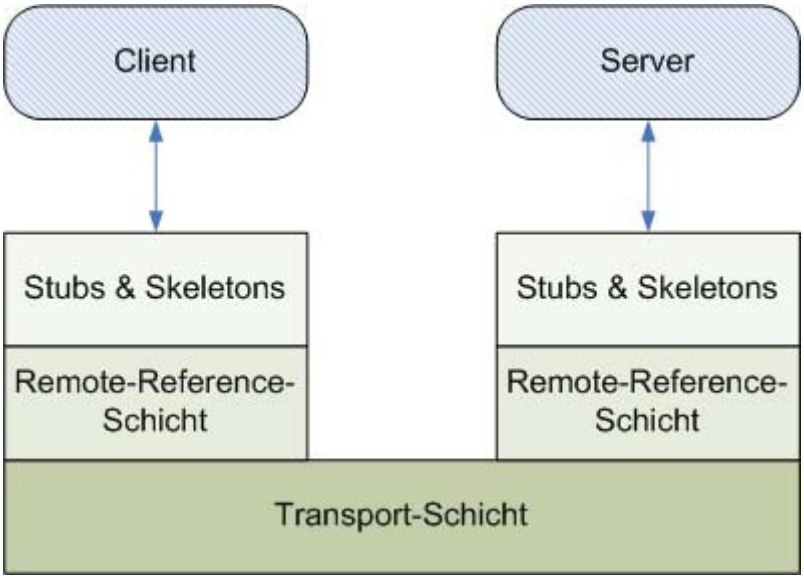

Abbildung 3-6: RMI Schichtenarchitektur

Die in Abbildung 3-6 gezeigte RMI-Schichtenarchitektur repräsentiert den Aufbau, der dieses Vorgehen ermöglicht:

- Die ***Transportschicht*** – Java RMI verwendet dafür TCP/IP – übernimmt die Übertragung des serialisierten Methodenaufrufs bzw. des Methodenergebnisses zwischen den entfernten Server- und Clientmaschinen.

- Die ***Remote-Refenrence-Schicht*** kontrolliert die Aufrufsemantik. Sie kann beispielsweise entscheiden, ob das entfernte Serverobjekt aktiviert werden muss, oder sie kann entscheiden, an welches von mehreren replizierten Serverobjekten sie einen gegebenen Methodenaufruf weiterleitet.

- Die ***Stubs & Skeletons-Schicht*** enthält die oben beschriebenen Serverstubs und Skeletons. Seit Java 5 werden diese vom System automatisch, ohne Zutun des Entwicklers, generiert.

Immer wenn ein Client eine Referenz auf ein entferntes Objekt erhält, eine sogenannte entfernte Objektreferenz, bekommt er in Wirklichkeit einen Serverproxy geschickt. Jedes Mal wenn nun der Client eine Methode an dieser entfernten Objektreferenz aufruft, wendet er sich an das lokale Proxyobjekt, das dann intern die Kommunikation mit dem entfernten Serverobjekt übernimmt.

3.8.1.2.2 RMI Elemente und Ablauf

Die Abbildung 3-7 zeigt die RMI Elemente und den Ablauf eines RMI-Aufrufes. Für die Kommunikation zwischen Client und Server sind vier Elemente unentbehrlich, die im Folgenden beschrieben werden:

- Das **Remote-Interface** definiert die auf dem Server zur Verfügung stehenden Funktionen und beschreibt damit das Verhalten der entfernten Funktionen.
- Die **Remote-Objekte** implementieren das obige *Remote-Interface*. Vom Server lässt sich eine oder mehrere Instanzen des *Remote-Objekts* erstellen.

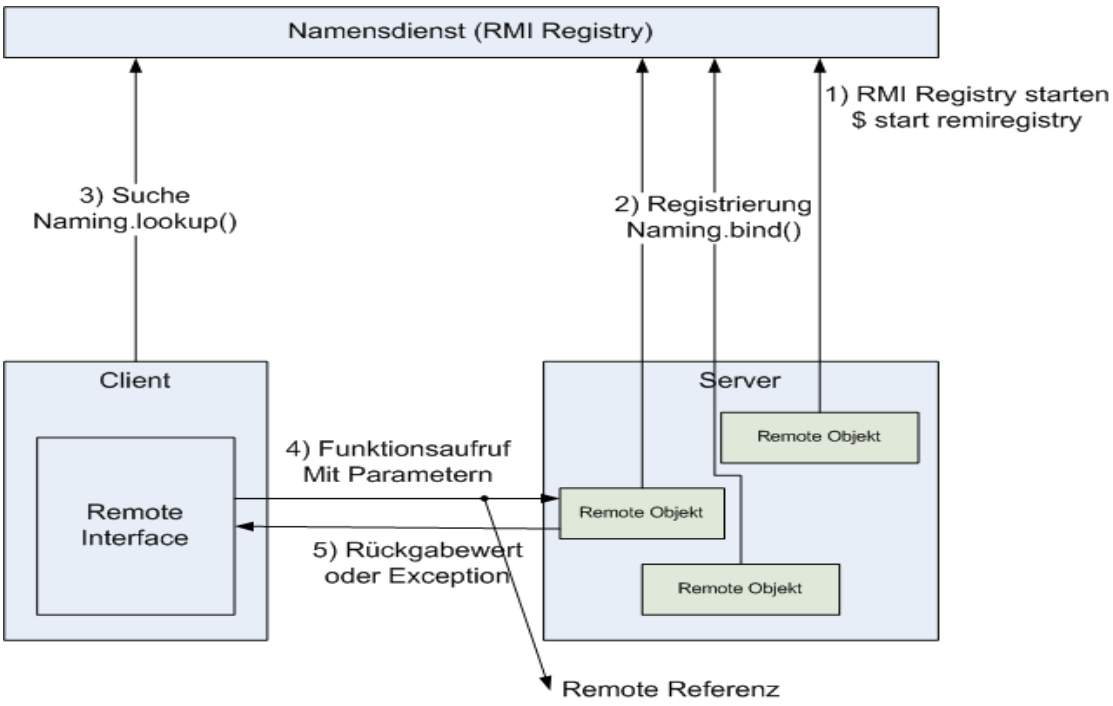

Abbildung 3-7: RMI Elemente und Ablauf

- Beim **Namensdienst** *(RMI Registry)* werden die *Remote-Objekte* vom Server registriert und die Referenzen auf diese *Remote-Objekte* sind von den Clients abzufragen. *RMI Registry* ist eine Implementierung eines einfachen Namensdienstes und ein Bestandteil des RMI Pakets.
- Als **Remote-Referenzen** werden die Referenzen auf *Remote-Objekte* bezeichnet.

Der Ablauf kann damit wie folgt beschrieben werden:

1. Zuerst startet man die RMI Registry mit dem Befehl *$ start rmiregistry* über die Befehlskonsole auf dem Server.
2. Danach instanziiert der Server ein oder mehrere Remote-Objekte und meldet diese mit der Funktion *Naming.bind()* bei der RMI Registry an.
3. Auf der Clientseite muss vor allem eine Referenz auf das entfernte Objekt mit der Funktion *Naming.lookup()* von der RMI Registry erfragt werden und danach kann der Client auf dieses zugreifen.

4. Jetzt ist der Client in der Lage die entfernte Funktion aufzurufen.
5. Letztlich erhält der Client entweder den Rückgabewert oder eine Ausnahme zurück, falls bei der Kommunikation ein Fehler aufgetreten ist.

3.8.1.2.3 Client Callbacks

In vielen Architekturen möchte der Server Aufrufe (Callbacks) im Client durchführen. Die Clients werden Server und der Server wird ein Client. Callbacks sind nützlich, falls mehrere Clients sich rechtzeitig über Änderungen auf dem Server informieren müssen. Zum Beispiel, stellen Sie sich einen Auktionsserver vor. Wenn ein Client ein höheres Angebot macht, müssen die anderen Clients darüber benachrichtigt werden. Ein weiteres Beispiel könnte eine Patientenakte in einem Hospital sein. Testergebnisse dürften von unterschiedlichen Personen an unterschiedlichen Stellen eingegeben werden. Inzwischen könnten andere Akten desselben Patienten an anderer Stelle angeschaut werden. Es wäre notwendig, sicherzustellen, dass dieser Client immer eine aktuelle Ansicht der Patientenakte hat. Eine einfache Lösung dazu ist eine Technologie, die *Polling* genannt wird, wobei der Client periodisch den Server nach den neuesten Informationen fragt. Dagegen sind Callbacks von Ereignissen gesteuert. Nur wenn eine Änderung auf dem Server entsteht, wird eine Callback Methode vom Server aufgerufen. Das ist eine deutlich effizientere Lösung als die Letzte.

Um die Callbacks zu erzeugen müssen Clients zuerst eine Benachrichtigungsmethode beim Server registrieren, die aufgerufen wird wenn ein passendes Ereignis geschieht. Dann warten die Clients auf den Aufruf der registrierten Methode durch den Server, sodass auf das Ereignis reagiert werden kann.

3.8.1.2.4 RMI und Threads

Wie in Abbildung 3-8 gezeigt, erzeugt jeder RMI-Aufruf auf der Serverseite einen neuen Thread. Das heißt, dass in der Java VM des Servers mehrere solcher Threads zugleich laufen können, zusammen eventuell mit weiteren dauerhaft laufenden Threads des Servers (z.B. Garbage Collector, Computer-Server,...).

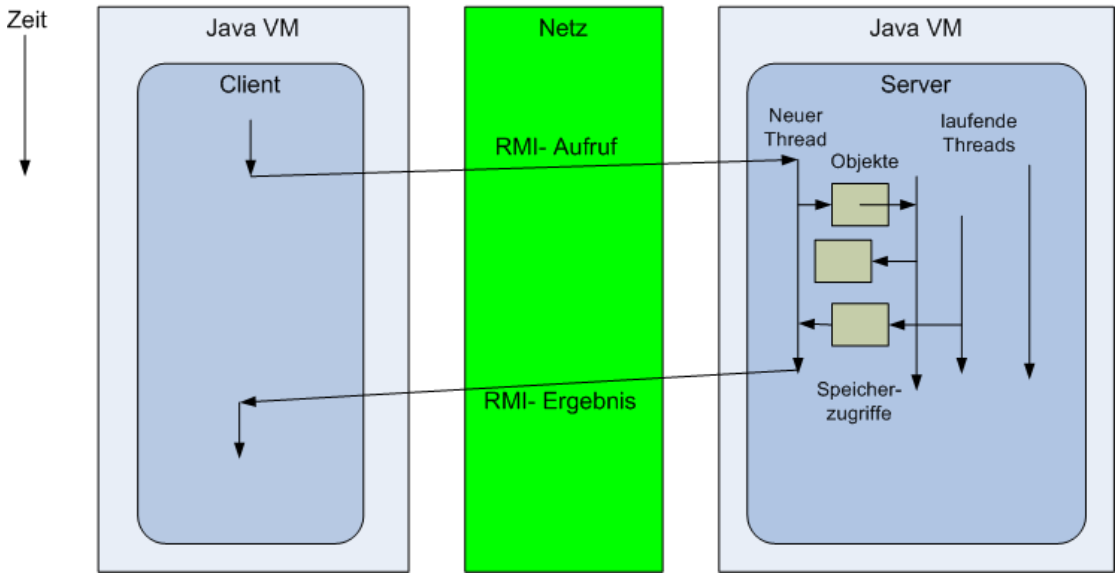

Abbildung 3-8: RMI und Threads

In jedem Fall muss man mit einer notwendigen serverseitigen Synchronisation rechnen (mit synchronized-Blocken oder mit wait() und notify()). Clientseitiges Sperren eines Remote-Objekts durch synchronized-Blocken ist unmöglich, da nur der lokale Stub gesperrt wird.

3.8.1.2.5 Serverobjektaktivierung

RMI unterstützt eine Technik namens Remote Object Activation (ROA). Neben der Möglichkeit, ein RMI-Serverobjekt ständig laufen zu lassen, so dass es auf einkommende Methodenaufrufe hören kann, gibt es außerdem die Option, das Serverobjekt erst bei Bedarf dynamischen zu starten und nach Abarbeitung eines oder mehrerer Methodenaufrufe wieder zu beenden. Damit ist vermeidbar, dass Serverobjekte andauernd im Speicher vorgehalten werden müssen.

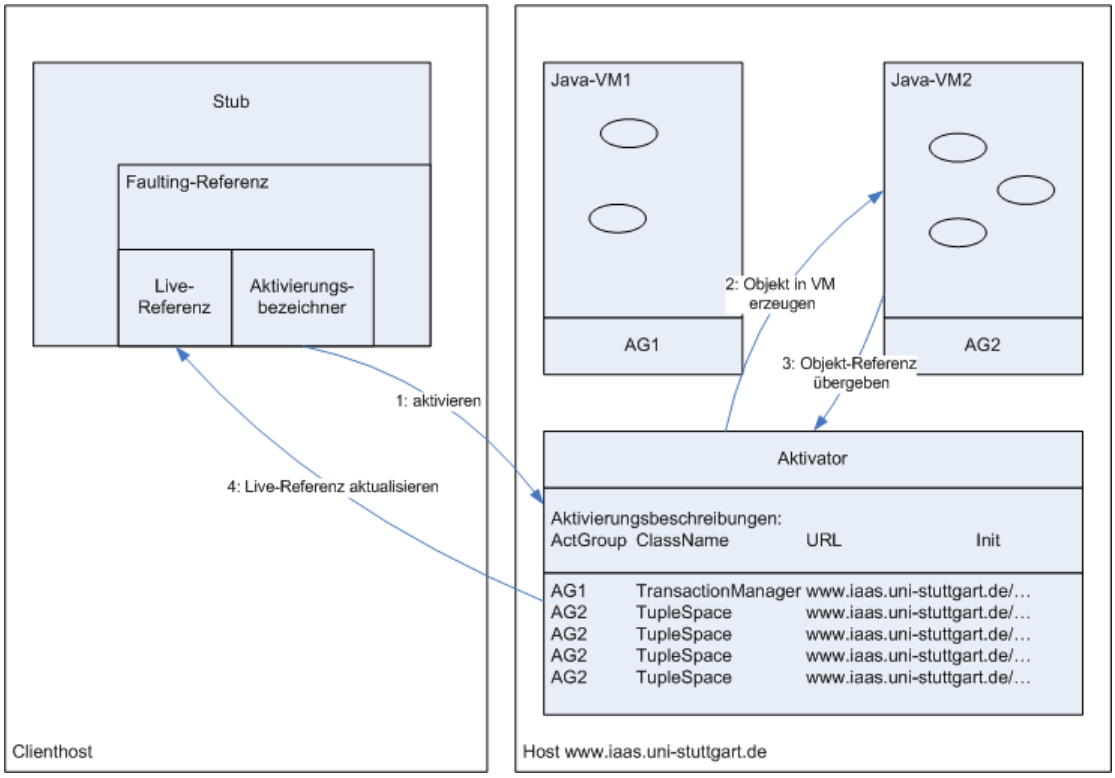

Abbildung 3-9: Aktivierung entfernter Objekte in Java/RMI

Diese Technik erfordert ein anderes Vorgehen auf der Serverseite. Für die Clientseite ist die Aktivierung vollständig transparent, wie in Abbildung 3-9 gezeigt. Auf der Clientseite erscheinen Serverobjekte immer zugänglich und verfügbar, auch wenn der Server nach dem Fehlerfall erneut gestartet ist. Ein zugrunde liegendes Konzept ist die Faulting-Referenz. Sie stellt eine intelligente Referenz auf ein entferntes Objekt dar, die in einem Stub verwaltet wird. Eine Faulting-Referenz enthält einen Aktivierungsbezeichner, der genug Informationen umfasst, um das Objekt zu aktivieren, und eine Live-Referenz auf das entfernte Objekt, möglicherweise ungültig. Wird die Faulting-Referenz in einem entfernten Methodenaufruf verwendet, prüft diese, ob die Live-Referenz aktuell auf ein aktiviertes Objekt verweist. Ist dies der Fall, wird die Live-Referenz dazu benutzt, den Methodenaufruf an das entfernte Objekt weiterzureichen. Ist die Live-Referenz ungültig, wird der Aktivierungsbezeichner dazu verwendet, den entfernten Host zu finden, auf dem das Objekt aktiviert werden muss. Jeder Host verfügt über einen Aktivator. Dieser erhält den Aktivierungsbezeichner. Der Aktivator benutzt den Bezeichner, um eine Aktivierungsbeschreibung zu ermitteln. Diese bestimmt, in welcher Java-VM das Objekt zu aktivieren ist, den Klassennamen des Objektes, eine URL für den Bytecode der Klasse und Initialisierungsdaten für das Objekt. Läuft die gewünschte VM nicht, startet der Aktivator sie. Dann wird die Aktivierungsbeschreibung an eine Aktivierungsgruppe weitergegeben, die es in jeder Java-VM gibt. Die Aktivierungsgruppe lädt dann den Bytecode von der URL, baut das Objekt auf und initialisiert es mit den bereitgestellten Daten, welche den Objektzustand beinhalten könnte. Sie liefert eine entfernte Referenz an den Aktivator zurück, der dann vermerkt, dass das Objekt nun läuft. Der

Aktivator wiederum liefert die Objektreferenz an die Faulting-Referenz im Stub des Clients zurück, der seine Live-Referenz aktualisiert.

3.8.1.3 JMX

Die Java Management Extensions (JMX), die von dem Java Community Process (JCP) entwickelt wurde, spezifiziert Schnittstellen und Protokolle zum Monitoring und Management von Java-Anwendungen. Seit Java 5 ist JMX in J2SE und J2EE integriert.

Die Architektur von JMX ist wie in Abbildung 3-10 gezeigt in drei Levels unterteilt.

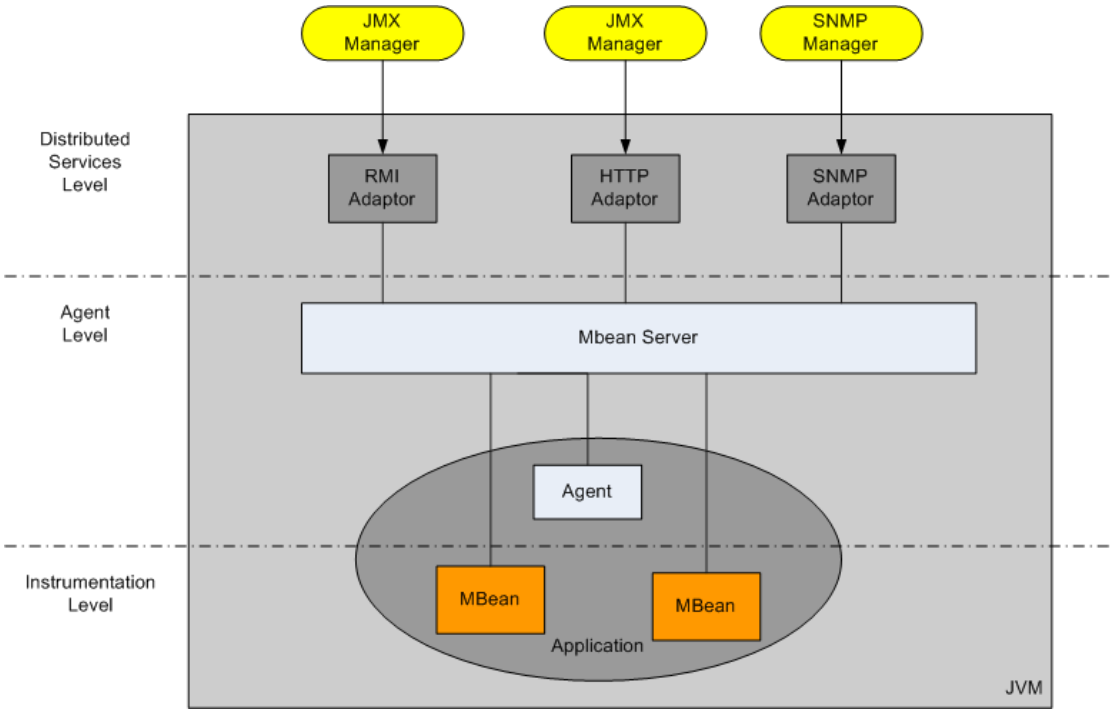

Abbildung 3-10: JMX Architektur

Instrumentation Level

Dieser Level enthält Managed Beans(MBeans) und die mit ihnen verwalteten Ressourcen. Es spezifiziert vier verschiedene Arten von solchen MBeans:

- Standard MBeans: sie gehöhren dem einfachsten Managed Bean Typ. Bei diesem Typ müssen gewisse Design-Pattern eingehalten werden. Dadurch, dass Getter- und Setter-Methoden zur Verfügung gestellt werden, sind die Attribute eines MBean zugreifbar. Zweitens braucht es mindestens ein öffentlicher Konstruktor. Es muss noch eine statische Schnittstelle mit den Attributen und Operationen vorhanden sein. Das Suffix der Namen dieser Schnittstelle ist zwangsläufig MBean. Ansonst könnte der MBean Server sie nicht erkennen. Beispielweise verwendet der Transaktionsmonitor diesen MBean. Die Schnittstelle *TransactionMonitorMBean* definiert eine Get-Operation an der Anzahl

der Transaktionen. Die Klasse *TransactionMonitor* implementiert nicht nur diese Schnittstelle, sondern auch die andere Schnittstelle *TransactionListener,* mit der sie sich über das Hinzufügen und Entfernen der Transaktionen informieren und entsprechende Änderungen vornehmen kann.

```
public interface TransactionMonitorMBean {
       int getTransactionSum();
}

public class TransactionMonitor implements
TransactionMonitorMBean, TransactionListener {
       private int transacitonSum;

       public TransactionMonitor(int transacitonSum) {
              super();
              this.transacitonSum = transacitonSum;
       }

       public int getTransactionSum() {
              return transacitonSum;
       }

       public void transactionCreated(Transaction transaction) {
              transacitonSum++;
       }

       public void transactionRemoved(Transaction transaction) {
              transacitonSum--;
       }
}
```

- Dynamic MBeans: Sie implementieren eine spezielle Schnittstelle DynamicMBean, die mehr Flexibilität zur Laufzeit anbietet.

- Open MBean: Darunter versteht man Dynamic MBeans, deren Typenwahl auf den Grunddatentypen beschränkt werden.

- Model MBeans: Dies Typ ist eine Erweiterung des Dynamic MBean. Ein Model MBean muss die Schnittstelle ModelMBean implementieren, die eine Ressource schnell verwaltbar machen kann.

Auf dieser Ebene gibt es noch eine wichtige Komponente, nämlich Notifikationsmodell. Mit dem Modell kann eine Notifikation bei einem bestimmten Event wie Methodeaufruf gesendet werden.

Agent Level

Dieser Level besteht aus MBean Server und Agent Services. Der MBean Server bildet den Kernmodul einer JMX Infrastruktur. Jede verwaltbare Ressource muss von MBeans im MBean Server registriert werden, damit sie über die Management Konsolen zugreifbar werden. Zum Beispiel meldet sich das eben besprochene *TransactionMonitorMBean* wie folgt beim MBean Server an. Die Klasse *ObjectName* enthält einen eindeutigen Namen, unter dem das TransactionMonitorMBean im Server registeriert wurde.

```
TransactionMonitor monitor = new TransactionMonitor(0);
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
ObjectName monitorName = new
       ObjectName("ServerSetup:name=TransactionMonitor");
mbs.registerMBean(monitor, monitorName);
```

Durch die Agent Services stehen neue Operationen an den registrierten MBeans im Server zur Verfügung. Beispielweise wird anhand des Timers ein Mechanismus eingeführt, mit dem in bestimmten Intervallen Notifikationen eines MBeans auszuführen sind. Es existiert noch weitere solche Dienste, die bei dem Aufbau einer komplizierten JMX Anwendung helfen.

Distributed Services Level

Dieser Level beinhaltet die Komponenten, die Management-Applikationen ermöglichen, mit JMX Agenten zu kommunizieren. Er bietet die Schnittestelle zur Implementierung von JMX Managers und definiert die Schnittestellen und Komponenten zum Zugriff auf den Agenten. Es kann über verschiedene sogenannte Protokoll-Adapter (RMI, HTML, usw.) zugegriffen werden.

Die JConsole ist ein JMX-kompatibeles Monitoring Programm und nutzt diesen Level zur Überwachung der Ressourcen einer Java-Anwendung. Die Informationen aller registrierten MBeans sind auch aufbereitet. Wie in Abbildung 3-11 gezeigt, kann man das Attribut „TransactionSum“ vom *TransactionMonitorMBean* aus der JConsole lesen.

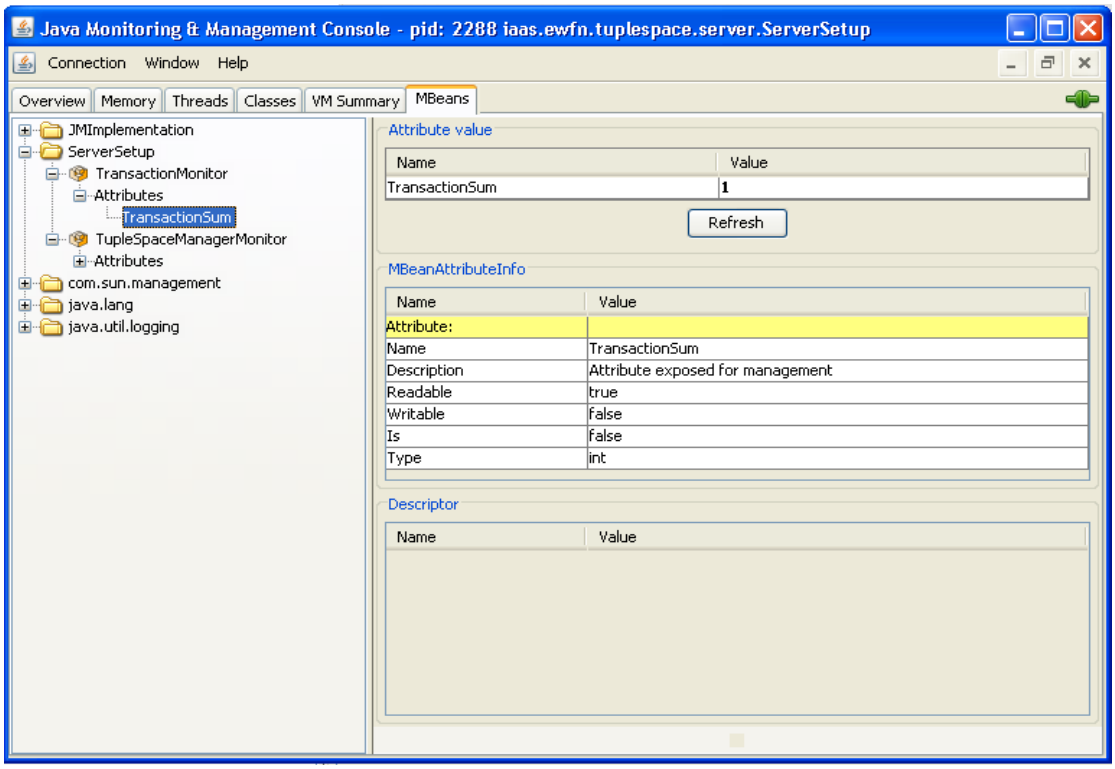

Abbildung 3-11: TransactionMonitorMBean in der JConsole

3.8.2 Übersicht

Die Abbildung 3-12 veranschaulicht die Übersicht der Implementierung auf Basis der oben besprochen Technologien. Die Middleware teilt sich in drei Schichten: Persistenzschicht, Logikschicht und Dienstschicht. Wie im Abschnitt 3.8.1.1 diskutiert, bei der Ausführung der Operationen, die die Dienste auf oberster Schicht zur Verfügung stellen, wird immer die transaktionsorientierten Kommanden auf der Zwischenschicht dazu benutzt, die Daten, die sich im entsprechenden Prevalent-System befinden, zu manipulieren.

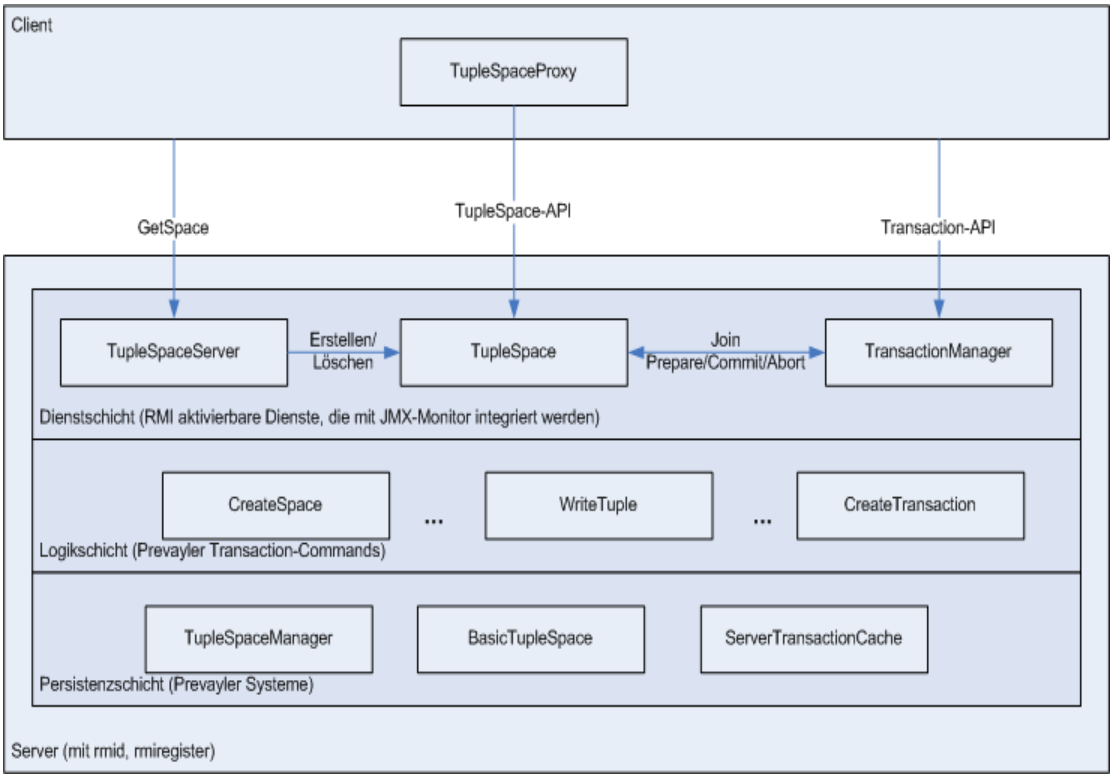

Abbildung 3-12: Implementierungsübersicht der Middleware

3.8.2.1 Umgebung zur Laufzeit

Alle Dienste innerhalb der Middleware sind als RMI aktivierbare Serverobjekte veröffentlicht, welche einen Namensdienst und ein Aktivierungssystem brauchen. Damit kann der entfernte Client über eine URI, die sich bereits beim Namendienst registriert hat, eine entfernte Objektreferenz bekommen, wobei noch inaktive Serverobjekte vom Aktivierungssystem automatisch erstellt und zur Verfügung gestellt werden. Deshalb müssen zwei RMI Daemonprogramme *rmiregistry* und *rmid* bereits aktiv sein, bevor der Middleware-Server startet. Fällt *rmiregistry* oder *rmid* aus, muss auch der Middleware-Server erneut gestartet werden, um alle Dienste noch mal beim Namensdienst und Aktivierungssystem zu registrieren, weil nach dem Neustart des Namensdienstes alle vorher registrierten Namen verloren sind und das Aktivierungssystem die Informationen über die angemeldeten

Serverobjekte nicht finden kann. Stürzt der Middleware-Server ab, muss rmiregistry und rmid nicht neu gestartet werden.

3.8.2.2 Dienstschicht

Die Middleware stellt hauptsächlich den Tupelräume-Dienst und den Transaktionsdienst zur Verfügung. Der erste besteht weiterhin aus einem Verwaltungsdienst und mehreren einzelnen Tupelraumdiensten. Der Verwaltungsdienst ist dafür verantwortlich, neue Tupelraumdienste einzurichten, laufende Tupelraumdienste auszuschalten und zu entfernen. Die eingerichteten Tupelraumdienste sind jeweils mit einer eindeutigen URI versehen, die vom Tupelräume-Dienst zugeordnet wird. Über diese URI kann der Client auf den relativen Tupelraum zugreifen.

Der Transaktionsdienst behält alle erstellten Transaktionen und deren Teilnehmer. Während der Ausführung einer Transaktion dient er im Wesentlichen als ein Transaktionskoordinator, der das Zwei-Phasen-Commit-Protokoll implementiert. Übrigens ist ein Scheduler integriert, um die abgelaufenen Transaktionen rechtzeitig und automatisch abzubrechen. Die Details werden noch im Abschnitt 3.8.3 erläutert.

Alle hier besprochenen Dienste registrieren sich als aktivierbare Serverobjekte beim Aktivierungssystem unter einer Gruppe, damit die neu generierten Objektinstanzen stets im selben Java VM laufen und die Informationen vom Logging an zentraler Stelle gesammelt werden können, was das Debugging, die Überprüfung und auch die Überwachung begünstigt. Alles ist aber für den Client transparent. Er sieht nur eine für immer gültige entfernte Objektreferenz, welche die Aktivierung des Servers gut verbirgt.

3.8.2.3 Logikschicht

Diese Schicht dient zur Trennung der sich in einem Prevayler System verkapselnden Geschäftsobjekte von der Dienstschicht. Nach der Prevayler Grundidee darf der Dienst nur durch sogenannte Kommanden, die als Transaktionen ausgeführt werden, die Geschäftsobjekte bearbeiten. Das bedeutet, dass jede Operation von einem Objekt, die das Objekt ändern könnte, sich in einem Kommando einkapseln muss. Das Prevayler System zeichnet alle Kommanden zum Objekt sequenziell auf, bevor sie tatsächlich ausgeführt werden. Beim Neustarten des Servers ist es in der Lage, den Objektzustand wiederherzustellen, indem die aufgezeichneten Kommanden nach der ursprünglichen Reihfolge noch mal ausgeführt werden.

3.8.2.4 Persistenzschicht

In der Middleware gibt es drei Geschäftsobjekte, die persistent gespeichert werden müssen, nämlich *TupleSpaceManager*, *BasicTupleSpace* und *ServerTransactionCache*. *TupleSpaceManager* umfasst die Liste der vorhandenen Tupelräume und deren relativen URIs, damit der Client auf die Tupelräume zugreifen kann. Eine Objektinstanz von „*BasicTupleSpace*“ entspricht einem Tupelraum und bewahrt alle abgegebenen Tupel und auch die Daten, die sich auf Transaktion und Benachrichtigung beziehen. Es bietet nicht nur die zum Zugriff auf dem Tupelraum notwendigen Operationen, sondern auch die transaktionsbezogenen Operationen wie commit, abort (Siehe 3.5.2). Der *ServerTransactionCache* speichert alle Transaktionen, die der zugehörige Transaktionsmanager erstellt hat, und die Teilnehmer, die sich an den Transaktionen beteiligt haben.

In der aktuellen Version kann das Prevalent-System noch nicht über die Prozessgrenze hinweg die Geschäftsobjekte verwalten. Da ein aktivierter Dienst nur in seinem eigenen Prozessraum laufen kann, so muss für jeden Dienst ein unabhängiges Prevalent-System eingerichtet werden. Alle einzelnen Tupelraumdienste, die ein Tupelräume-Dienst erstellt hat, verwenden damit jeweils eigene Prevayler Systeme.

Das Prevalent-System basiert auf dem Dateisystem und benötigt ein Verzeichnis zum Speichern seiner protokollierten Daten. Soll dieses System vollständig entfernt werden, muss das relative Verzeichnis manuell oder durch Dateioperationen auch gelöscht werden, denn das Prevayler System selbst stellt noch keine entsprechende Operation zur Verfügung.

3.8.2.5 Tupelraum-Proxy

Um auf Tupelräume zuzugreifen, kann der Client direkt über die Tupelraum-Schnittstelle oder durch einen Proxy mit den eingerichteten Tupelraumdiensten kommunizieren. Der Unterschied besteht darin, dass der Proxy, der die Operationen an den Tupelräumen verkapselt, noch den Teil des Anforderungs- und Antwort-Protokolls (Siehe 3.4.1) auf der Clientseite implementiert. Sollte der Client seiner Anwendung Höchstens-Einmal-Aufrufsemantik gewährleisten und aber nicht selbst dafür sorgen, muss der Proxy zur Anwendung kommen, da das Protokoll als Ergänzung zum Kommunikationsprotokoll vom RMI realisiert ist.

Anfangs erstellt der Proxy zuerst eindeutige Identifizierer für einige Methodenaufrufe, und diese IDs werden mit allen wiederholten Anforderungsnachrichten an den Server gesendet, um die entsprechenden Aufrufe zu identifizieren. Ist ein Aufruf erfolgreich ausgeführt, sendet der Proxy ggf. eine Bestätigungsantwort an den Server, damit der Server das vorher aufgezeichnete Ergebnis für den entsprechenden Aufruf entfernen kann.

Empfängt der Proxy beim Aufruf eine Ausnahme, stellt er eine erneute Anforderungsabfrage an den Server, solange bis der vorgegebene Timeout auftritt. Üblicherweise ist es nicht sinnvoll, gleich nach einem

fehlschlagenden Aufruf noch einmal den Selben zu probieren. Der Fehler, der die Ausnahme verursacht hat, könnte in einer kurzen Zeit immer noch aktiv sein. Somit gewinnt es an Bedeutung, eine bestimmte Zeitspanne zwischen zwei nacheinander auszuführenden Aufrufen abzuwarten. Es gibt viele Algorithmen, um diese Zeitspanne zu berechnen. Die einfachste Lösung ist immer ein konstantes Zeitintervall zu benutzen. Sie kann aber durch massive Anforderungen den Netzverkehr stark belasten. Wir verwenden hier den gebräuchlichsten Algorithmus, darin wächst das Zeitintervall nach jedem Versuch exponentiell. Um zu vermeiden, dass ein riesiges Zeitintervall den Proxy lange blockiert, sollte das Auftreten eines Timeouts vor Erhöhung des Zeitintervalls überprüft werden.

Für die Leseoperationen wie in, read und sync kann der Client dem Proxy eine bestimmte Wartezeit übergeben, die angibt, dass wenn kein Ergebnis beim ersten erfolgreichen Aufruf zurückgegeben wird, der Proxy sich beim Server anmelden muss. Kommt ein gesuchter Tupel im Tupelraum vor, benachrichtigt der Server den Proxy sofort, der noch mal dieselbe Leseoperation ausführt. Hat ein nebenläufiger Clientprozess inzwischen den Tupel weggenommen, könnte die erneute Ausführung wiederum zum leeren Ergebnis führen und dann muss der Proxy noch mal auf die Benachrichtigung durch den Server warten. So läuft der Proxy rekursiv solange, bis er einen gesuchten Tupel bekommt. Ist die Wartezeit abgelaufen, gibt der Proxy einen leeren Verweis zurück.

3.8.3 Transaktion durch die Middleware

3.8.3.1 Erstellung und Verwendung

Der Client startet eine neue Transaktion durch das Erstellen mit Hilfe von *TransaktionFactory*, der die Anforderung zum *TransactionManager* weiterleitet und die erstellte Transaktion dem Client zurückgibt. Wie in Abbildung 3-13 gezeigt, wird die Transaktion als Parameter bei der Ausführung von Operationen an einen Tupelraumdienst mit übertragen. Wird der Dienst diese Transaktion akzeptieren und deren Operationen überwachen, muss er sich der Transaktion als Teilnehmer anschließen. Somit muss der Tupelraumdienst die Schnittstelle *TransactionParticipant* implementieren. Der Client, der die Transaktion gestartet hat, kann auch ein Teilnehmer an der Transaktion sein.

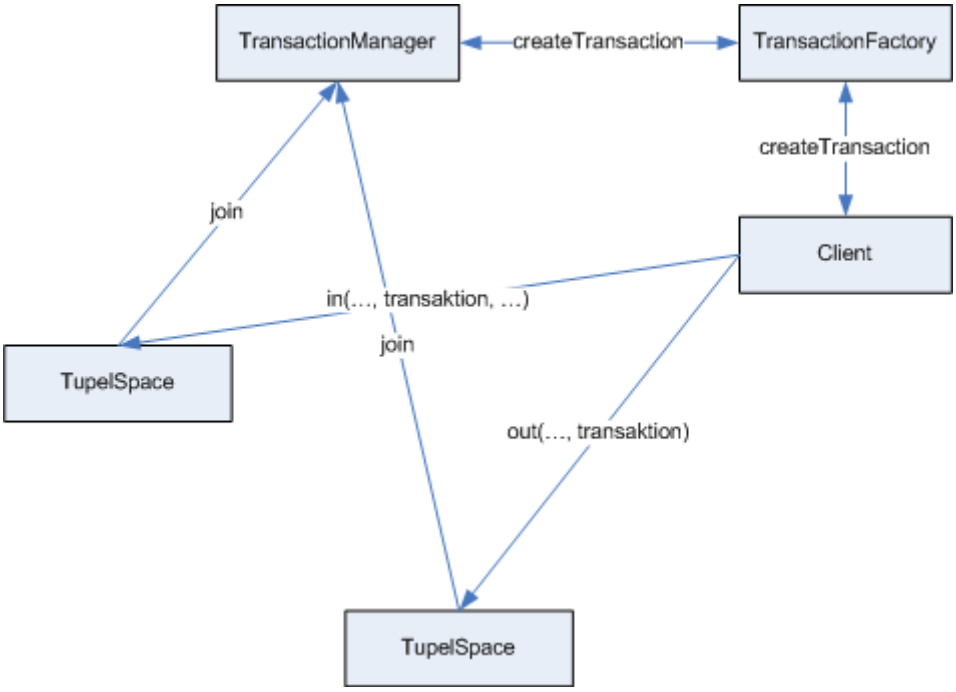

Abbildung 3-13: Erstellung und Verwendung von Transaktionen

Die Transaktion wird beendet wenn der Client sie festschreibt (*commit*) oder abbricht (*abort*). Ist die Transaktion erfolgreich festgeschrieben, werden alle Operationen, die unter der Transaktion ausgeführt sind, erledigt. Der Abbruch der Transaktion bedeutet, dass alle unter der Transaktion auszuführenden Operationen schließlich nicht stattfinden.

3.8.3.2 Transaktionszustände

Die folgenden Zustände sind definiert zur Kommunikation zwischen Manager und Teilnehmern:

- ACTIVE: Der Zustand der neu erstellten Transaktion
- PREPARING: In diesem Zeitpunkt sammelt der Manager noch die Stimmenabgaben für die Transaktion ein.
- UNJOINED: Der Teilnehmer benachrichtigt den Manager, dass er sich an der Abstimmung nicht beteiligt.
- PREPARED: Der Teilnehmer ist bereit, die Transaktion festzuschreiben.
- COMMITTED: Die Transaktion ist festgeschrieben.
- ABORTED: Die Transaktion ist abgebrochen.

3.8.3.3 Transaktionsbeendigung

Für den Client startet die Transaktion mit ACTIVE, sobald die Erstellungsmethode *create* einen Verweis auf die erstellte Transaktion zurückgibt. Der Client beendet die Transaktion mit Ausführung von *commit* oder *abort* durch den Transaktionsmanager.

Empfängt der Manager die Commit-Anforderung vom Client, setzt er das Zwei-Phasen-Commit-Protokoll (Siehe 3.5.2) ein. Dabei geht die

Transaktion gleich in den Zustand PREPARING über. Nach einem erfolgreichen Commit erreicht die Transaktion den Zustand COMMITTED. Ansonsten wird die Transaktion abgebrochen als wie wenn der Manager die Abort-Anforderung bekommen hätte. Tritt ein Timeout während der Transaktion auf, wechselt sie auch ihren Zustand nach ABORTED. Die Abbildung 3-14 veranschaulicht die Zustandsübergänge aus der Managersicht und der Clientsicht.

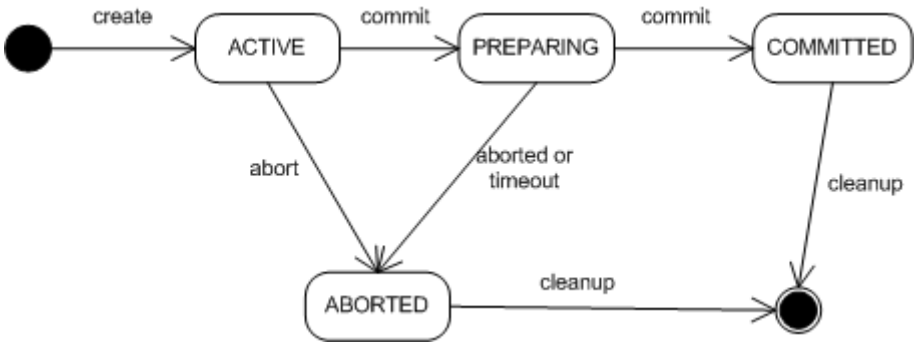

Abbildung 3-14: Zustandsdiagramm aus der Managersicht und der Clientsicht

Beim Teilnehmer ist der Zustandsübergang komplizierter. Wie in Abbildung 3-15 gezeigt, startet die Transaktion beim Teilnehmer, erst wenn er sich an der Transaktion beteiligt. Der Teilnehmer kann die Transaktion abbrechen, bevor der Manager die Abstimmungsanforderung *prepare* an ihn sendet. Wenn nicht, fängt der Teilnehmer mit der Abstimmungsphase an. Zum nächsten Schritt gibt es drei mögliche Rückantworte für *prepare*.

- Ist an dem Teilnehmerzustand keine Änderung unter der Transaktion vorgenommen worden, kann der Teilnehmer mit der Antwort UNJOINED den Manager benachrichtigen, dass er auf seine Abstimmung verzichtet. Denn seine Stimme soll die endgültige Entscheidung vom Manager nicht beeinflussen. Der Teilnehmer erreicht dann den Zustand UNJOINED.

- Haben die Operationen unter der Transaktion den Teilnehmerzustand geändert, muss der Teilnehmer versuchen, sich auf die Änderungen vorzubereiten, die beim zukünftigen Aufruf von *commit* geschehen werden. Wenn der Teilnehmer diese Vorbereitung erfolgreich durchgeführt hat, gibt er PREPARED an den Manager zurück. Zugleich erreicht er den Zustand PREPARED.

- Kann der Teilnehmer die obige Vorbereitung nicht durchführen, signalisiert er das dem Manager mit der Antwort ABORTED. Dann wechselt er in den Zustand ABORTED.

Befindet sich der Teilnehmer nun in dem Zustand PREPARED, wartet er auf die letzte Entscheidung des Managers. Bei der Entscheidung *commit* nimmt der Teilnehmer alle vorbereiteten Änderungen vor und erreicht damit den Zustand COMMITTED. Bei der Entscheidung *abort* muss der Teilnehmer ggf. seinen Zustand zurücksetzen.

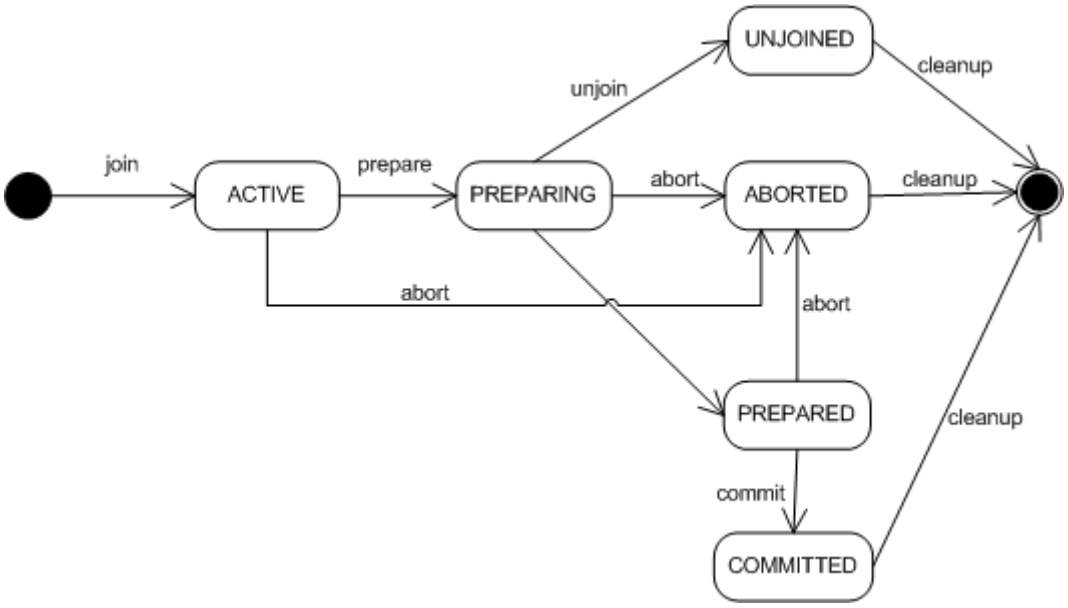

Abbildung 3-15: Zustandsdiagramm aus der Teilnehmersicht

4 Ausführung verteilter Prozessmodelle

In diesem Kapitel werden wir darüber diskutieren, wie verteilte Prozessmodelle mithilfe der oben entworfenen Middleware zur Ausführung gebracht werden. Ein modelliertes Prozessmodell lässt sich nach gewissen Prinzipien in ein verteiltes Prozessmodell transformieren. Die Middleware kommt zur Verwendung, um verschiedene Workflow Patterns zu unterstürzen und damit zur Koordinierung des Ablaufs eines Workflows beizutragen.

4.1 Transformation in verteilte Prozessmodelle

Die meisten Prozessmodellierungstools richten sich nach den Anforderungen der zentralisierten Workflow-Engine. Um das erstellte Prozessmodell in der verteilten Workflow-Engine ablaufen zu lassen, muss es in ein verteiltes Prozessmodell transformiert werden. Nach den Prinzipien der dezentralisierten Orchestrierung sind mehrere Koordinatoren dafür verantwortlich, den Ablauf eines Workflows zu steuern. Jeder Koordinator umfasst üblicherweise nur einen Teil vom gesamten Prozess. Die Koordination zwischen den Teilprozessen entspricht der Kommunikation zwischen den Koordinatoren.

Der erste Schritt der Transformation ist, das ursprüngliche Prozessmodell aufzuteilen und jeder Partition einen neuen Koordinator zuzuordnen. Die Aufteilung kann nach den bestimmten Attributen wie Ausführungsrollen und Ausführungsabteilung der Aktivitäten oder nach den potentiellen Belastungen der Aktivitäten durchgeführt werden. Welche Verfahren zum tatsächlichen Ablauf des Prozesses passen, ist oft anwendungsspezifisch. Das grundsätzliche Prinzip lautet, dass der wegen der Aufteilung auftretende Overhead die Ausführung des Prozessmodells möglichst wenig beeinflusst.

Zunächst verteilen wir alle Koordinatoren auf ein Netz, in dem die Workflow-Engine laufen soll. Um die direkten Verbindungen zwischen Koordinatoren zu unterbrechen, soll dem einzelnen Koordinator verborgen bleiben, wo sich die anderen Koordinatoren befinden. Er beinhaltet aber die Adressen der Vermittler, mit deren Hilfe er mit anderen Koordinatoren kommunizieren kann. Jedoch verlieren wir wegen der verteilten Ausführung die Übersicht über den Workflow-Zustand und die globale Kontrolle über den Prozess. Um dies zu vermeiden, macht es Sinn, einen speziellen Koordinator einzuführen, der sich mit allen anderen Koordinatoren verbindet. Mit diesem Koordinator können wir einen Befehl an die anderen senden, um eine globale Aufgabe zu erledigen, beispielweise den aktuellen Prozesszustand zu erhalten.

Der nächste Abschnitt wird zeigen, wie das transformierte Prozessmodell mithilfe der als Vermittler dienenden Middleware zur Ausführung gebracht wird.

4.2 Workflow Patterns

Vor der weiteren Besprechung wird zuerst angenommen, dass das Prozessmodell nach den einzelnen Aktivitäten unter den Koordinatoren aufgeteilt ist. Jeder Koordinator hat genau eine Aktivität. Dadurch, dass gewisse Information zwischen zwei Koordinatoren ausgetauscht werden, wird der Übergang einer Aktivität auf eine andere Aktivität implementiert. Bei einer Serie dieser Informationen spricht man von einem Kontrollfluss. Im Workflow gibt es viele Kontrollflusskonstrukte (Workflow Patterns). Im Folgenden diskutieren wir nur darüber, wie die Basiskonstrukte - Sequenz, Branche und Schleife – mithilfe von Tupelräumen implementiert werden. Die anderen Kontrollflusskonstrukte sind daraus zusammenzusetzen.

4.2.1 Kontrolltoken

Eben haben wir den Kontrollfluss besprochen, der in der Tat eine Menge von bestimmten Informationen enthält. Diese Information wird als Kontrolltoken bezeichnet. Von dem Kontrolltoken kann man sowohl die zugehörige Prozessinstanz also auch die Ausgangs- und Eingangsaktivität erfahren. Die zur Entscheidung benutzten Daten sind Alle Informationen werden in einer wie folgt vordefinierten Struktur verkapselt, die als Tupel durch einen Tupelraum zu übertragen ist.

```
public class ControlTokenTuple {
      public String processId;
      public String instanceId;
      public String fromActivityId;
      public String toActivityId;
}
```

Zum Beispiel versteht man unter dem Token (“process1”, “instance1”, “activity1”, “activity2”) ein Kontrollfluss von der Aktivität „activity1“ nach der Aktivität „activity2“ innerhalb des Instanzes “instance1” des Prozesses „process1“.

4.2.2 Sequenz

Die Sequenz meint die Abfolge von Aktivitäten, die nacheinander ausgeführt werden. Eine Aktivität davon wird erst ausgeführt, nachdem ihr Vorgänger erfolgreich beendet wurde und sie das passende Kontrolltoken empfangen hat. Die Abbildung 4-1 zeigt die auf einem Tupelraum basierende Implementierung eines sequenziellen Routings. Hat der Prozess1 den Task1 beendet, schreibt er mit der Operation „out“ das Kontrolltoken („process1“, „instance1“, „task1“, „task2“) in einen Tupelraum hinein, der für den Prozess2 auch zugreifbar ist, während der Prozess2 auf das Kontrolltoken wartet. Sobald der Prozess2 mit der

Operation „in“ das Kontrolltoken aus dem Tupelraum entnimmt, fängt er damit an, den Task2 auszuführen.

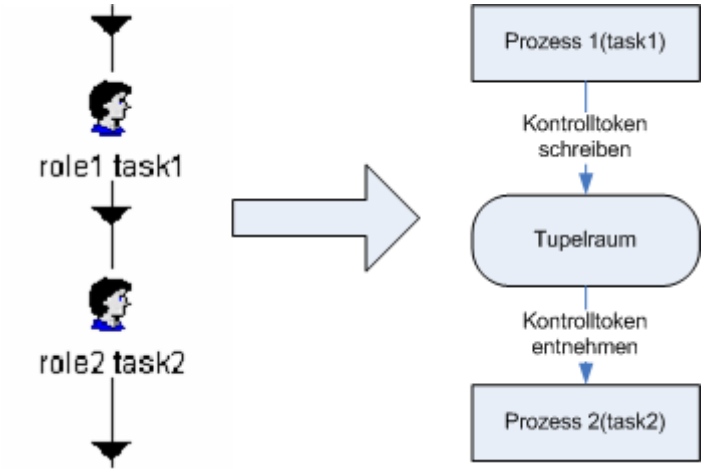

Abbildung 4-1: Implementierung eines sequenziellen Routings

4.2.3 Parallel Split und Synchronisation

Unter dem Pattern *Parallel Split* (AND-split) versteht man eine Stelle im Workflow-Prozess, an der sich eine einzelne Prozedur in mehrere Prozeduren teilt, die parallel auszuführen sind, sodass Aktivitäten erlaubt sind, welche gleichzeitig oder in beliebiger Reihfolge ausgeführt werden können. Und das Pattern *Synchronisation* (AND-join) beschreibt eine Stelle im Workflow-Prozess, an der mehrere parallele Prozeduren in einer einzelnen Prozedur zusammenlaufen, um mehrere Aktivitäten zu synchronisieren.

Die beiden Patterns werden dazu benutzt, ein paralleles Routing zu spezifizieren. Die Abbildung 4-2 veranschaulicht die Implementierung eines parallelen Routings. *Parallel Split* und *Synchronisation* spiegeln tatsächlich die Verteilung und die synchronisierte Einsammlung der für die parallel auszuführenden Aktivitäten notwendigen Kontrolltoken wieder. Im gezeigten Beispiel generiert der Prozess1 zwei Kontrolltoken („process1“, „instance1“, „task1“, „task2“) und („process1“, „instance1“, „task1“, „task3“) und legt sie in einen Tupelraum ab. Danach holen der Prozess2 und der Prozess3 jeweils den für sich bestimmten Token vom Tupelraum ab. So können die beiden Prozesse unabhängig voneinander ausgeführt werden. Nach der Ausführung legen sie zwei neue Tokens („process1“, „instance1“, „task2“, „task4“) und („process1“, „instance1“, „task3“, „task4“) in einen Tupelraum ab.

Da mehrere Instanzen gleichzeitig laufen könnten, kann der Prozess4 nicht vorhersagen, von welchen Instanzen die Kontrolltoken vorkommen. Andererseits muss der Prozess4 noch sicherstellen, dass die empfangenen beiden Kontrolltoken aus dem Prozess2 und Prozess3 zu derselben Instanz gehören. Deshalb wird die Operation „sync“ verwendet, die zwei Leseoperationen synchronisiert, die jeweils die vom Prozess2 und Prozess3 abgelegte Kontrolltoken durch die Vorlagen („process1“, null?,

„task2“, „task4“) und („process1“, null?, „task3“, „task4“) entnehmen. Die zwei Vorlagen hängen über den Attribute „instanceId“ zusammen, damit es sicherzustellen ist, dass die gefundenen Kontrolltoken sich auf derselben Instanz befinden. Nach der Ausführung dieser Operation wird der Prozess4 solange blockiert, bis die beiden Token im Tupelraum verfügbar sind, dadurch ist die Synchronisation vom Prozess2 und Prozess3 implementiert.

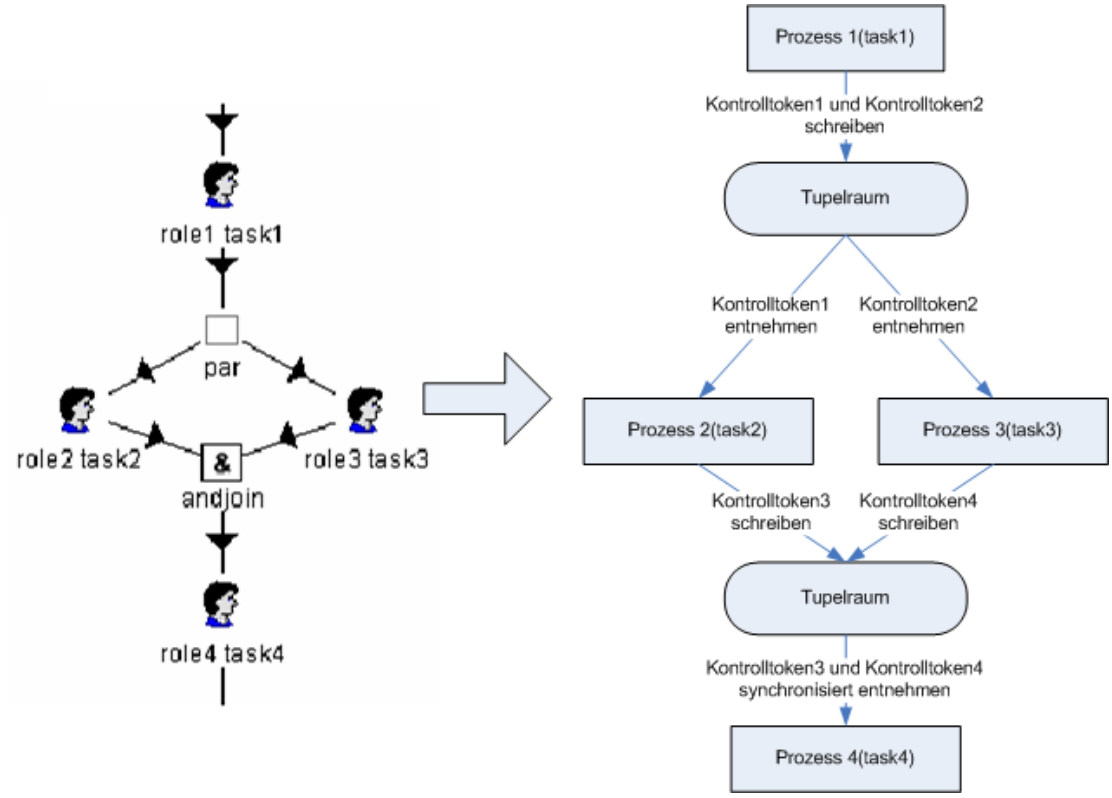

Abbildung 4-2: Implementierung eines parallelen Routings

4.2.4 Exclusive Choice und Simple Merge

Im Workflow-Prozess kommt oft solch eine Stelle vor, an der auf einer Entscheidung oder auf Kontrolldaten basierend, eine von mehreren Branchen ausgewählt wird. Später laufen diese alternativen Branchen an einer Stelle zusammen ohne Synchronisation, aber mit der Voraussetzung, dass nur eine der alternativen Branchen ausgeführt wird. Die beiden Stellen entsprechen jeweils den Workflow Patterns *Exclusive Choice* (XOR-split) und *Simple Merge* (XOR-join). Die Zusammenarbeit der beiden Patterns kann konditionelles Routing spezifizieren.

Wie in Abbildung 4-3 gezeigt, hat die Implementierung eines konditionellen Routings ein ähnliches Konstrukt wie im letzten Abschnitt. Der Unterschied besteht darin, dass der Prozess1 erst eine einzelne Entscheidung an Hand der beinhalteten Bedingung treffen muss und dann ein der Entscheidung entsprechendes Kontrolltoken („process1“, „instance1“, „task0“, „task1“) oder („process1“, „instance1“, „task0“,

„task2“) in den Tupelraum ablegt. Der ausgewählte Prozess wird ausgeführt, während der andere Prozess weiterhin blockiert wird. Sobald ein Kontrolltoken entweder aus dem Prozess2 oder aus dem Prozess3 im Tupelraum ankommt und der Prozess4 den Token aus dem Tupelraum erhalten hat, kommt das konditionelle Routing zum Ende.

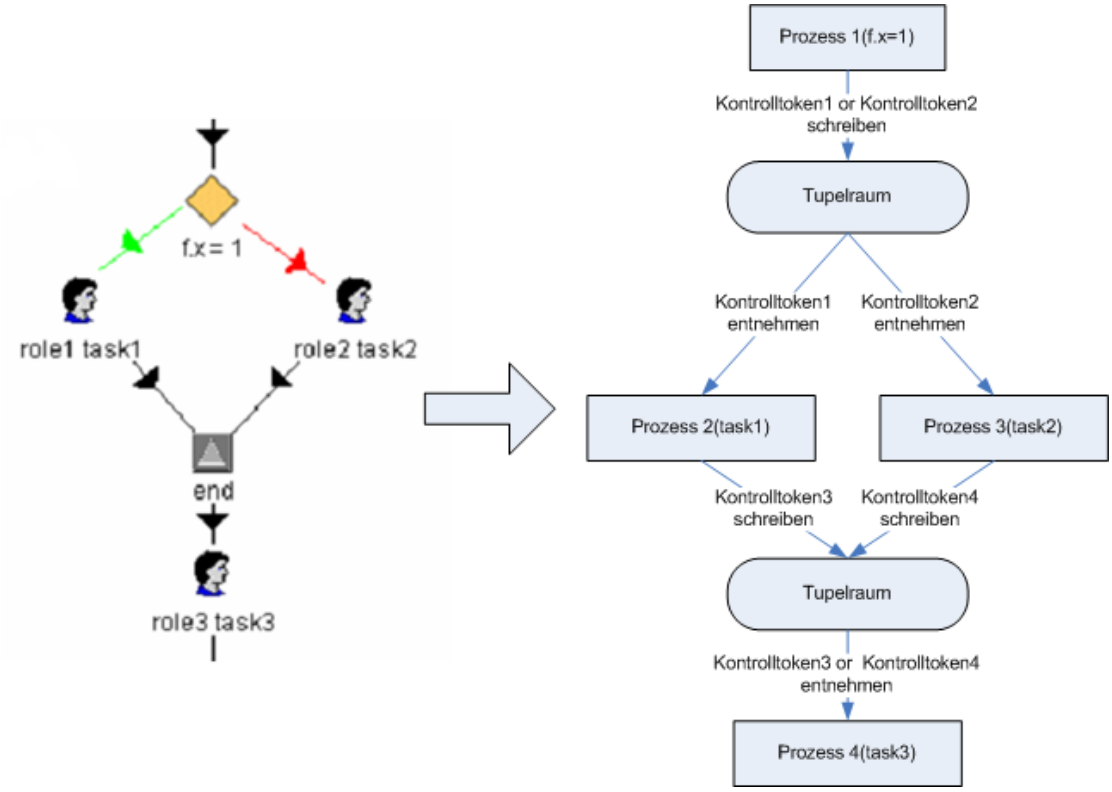

Abbildung 4-3: Implementierung eines konditionellen Routings

4.2.5 Iteration

Zur wiederholten Ausführung einer oder mehrerer Aktivitäten wird das Workflow Pattern *Iteration* benötigt. Im Hinblick auf die Überprüfungsbedingung der Schleife kann man zwei Iterationskonstrukte unterscheiden. Das erste Konstrukt startet damit, die Erfüllung der sogenannten Eingangsbedingung zu überprüfen. Ist die Bedingung erfüllt, wird die Schleife einmal durchgeführt und dann die Überprüfungsprozedur noch mal ausgeführt. So läuft der Vorgang rekursiv solange, bis die Bedingung nicht mehr erfüllt ist. Das Zweite verlegt die Überprüfungsprozedur in das Ende der Schleife. Sobald die sogenannte Ausgangsbedingung erfüllt ist, kommt die Iteration zum Ende.

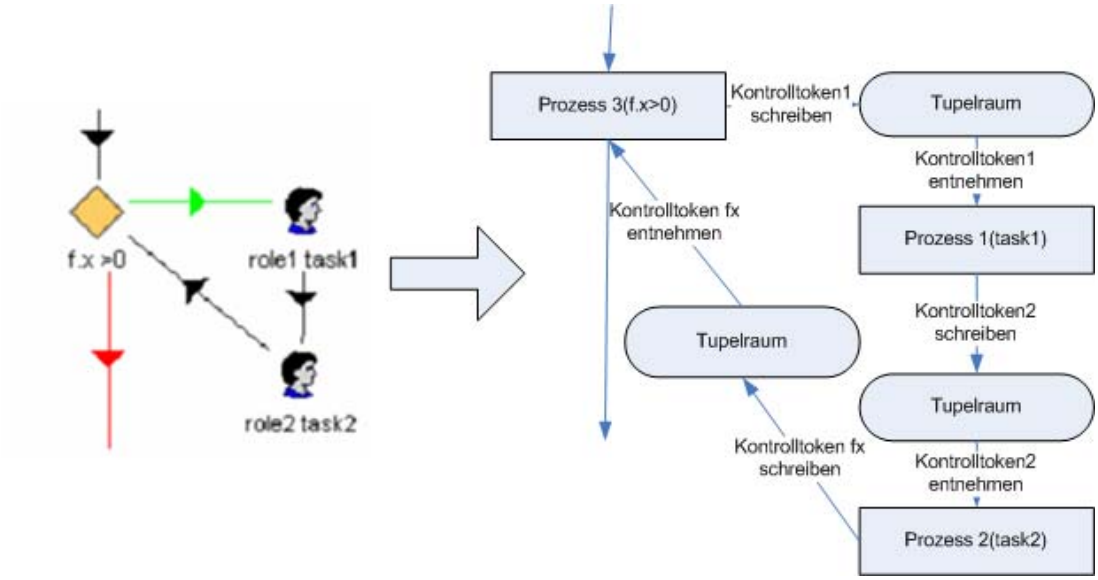

Abbildung 4-4: Implementierung einer Iteration mit der Eingangsbedingung

Die Abbildung 4-4 zeigt die Implementierung einer Iteration mit der Eingangsbedingung. In dieser Iteration ist der Prozess 3 immer aktiv, wobei der Vorgang rekursiv ausgeführt wird, dass wenn die Bedingung f.x>0 erfüllt ist, sendet der Prozess 3 das Kontrolltoken („process1“, „instance1“, „task0“, „task1“) durch einen Tupelraum an den Prozess 1 und danach wartet er auf das aus dem Prozess 2 kommende Kontrolltoken („process1“, „instance1“, „task2“, „task0“). Innerhalb der Schleife läuft eine einfache Sequenz. Falls f.x>0 nicht mehr erfüllt ist, ist der Prozess 3 und auch die Iteration erfolgreich erledigt.

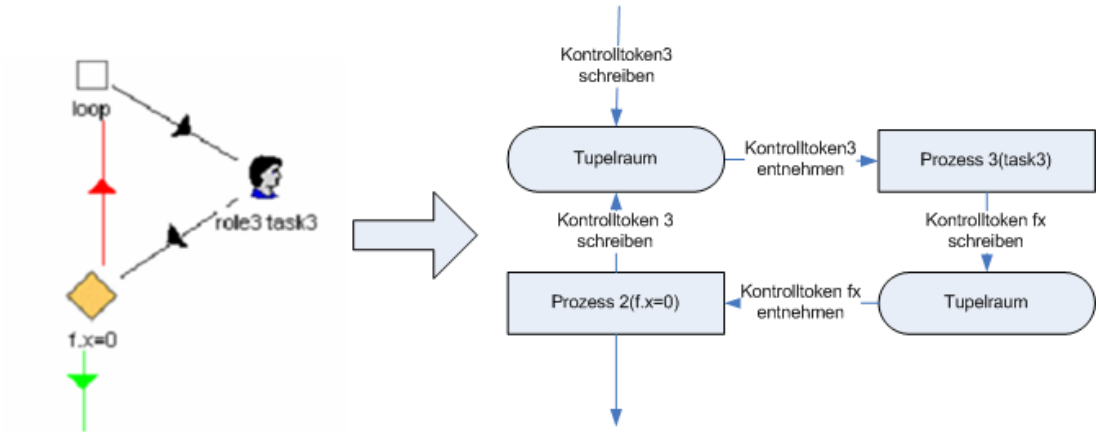

Abbildung 4-5: Implementierung einer Iteration mit der Ausgangsbedingung

Die Implementierung des anderen Iterationskonstrukts ist ähnlich wie die eben besprochene. Wie in Abbildung 4-5 gezeigt, startet die Iteration mit einem Kontrolltoken3 („process1“, „instance1“, null, „task3“), das für den Prozess3 im Tupelraum abgelegt wird. Der Prozess2 spiegelt einen Ausgangspunkt der Iteration wieder. Ist die Bedingung f.x=0 erfüllt, bricht er die Iteration ab. Ansonst schreibt er einem Kontrolltoken3 („process1“, „instance1“, „task0“, „task1“) in den Tupelraum, damit die Iteration fortgesetzt wird.

4.3 Testing

Zum Aufbau einer stabilen und zuverlässigen Software ist das Testing unentbehrlich. Deshalb entwerfen wir die Szenarien nach den oben besprochenen Umsetzungen des Workflow Patterns, um die Zuverlässigkeit und Wiederherstellung sowie Performanz der implementierten Middleware zu überprüfen. Im Folgenden werden die drei Aspekte jeweils näher erläutert.

4.3.1 Zuverlässigkeit

Bei der Zuverlässigkeit spricht man üblicherweise von Korrektheit, Sicherheit und Fehlertoleranz. Aus den Testergebnissen ergibt sich, dass die nach den Workflow Patterns konstruierten nebenläufigen Prozesse zusammenhängend ausgeführt und die gewünschten Abläufe nachgewiesen werden. Die Korrektheit des Middleware-Verhaltens ist dadurch auch festgestellt. Insbesondere bei den fehleranfälligen Situationen wie Lesen und Synchronisation mit Konflikten.

Mit der Sicherheit wurde wenig gerechnet. Man kann nur eine sogenannte Sicherungsdatei dazu benutzen, den Zugriff auf den Tupelraum zu kontrollieren. Jedoch ist es nicht möglich, eine Sicherheitsstrategie einem einzelnen Tupel zuzuordnen, was für spezifische Anwendungen sehr nützlich ist.

In dieser Middleware ist nur eine beschränkte Fehlertoleranz wegen Mangel an Redundanz implementiert. Auf der Kommunikationsebene könnte der Netzausfall maskiert werden, wenn das Netz wieder in Ordnung ist, bevor ein Timeout auftritt. Schlägt der Middleware-Server fehl, kann er nach dem manuellen erneuten Starten mithilfe der Wiederherstellung auf seinen letzten Zustand zurückgesetzt werden. Unter Verwendung der persistenten entfernten Referenz greift der Client auf die Middleware zu, ohne den Serverausfall zu bemerken. Hierfür wird aber vorausgesetzt, dass die Dauer des Ausfalls weniger als der vorgegebene Timeout auf der Clientseite beträgt.

4.3.2 Wiederherstellung

Die Wiederherstellung beruht auf dem Persistenzdienst (Siehe 3.3), der regelmäßig den Zustand der Middleware durch Snapshotting aufnimmt und auch alle Operationen an der Middleware aufzeichnet. Bei jedem Starten des Servers wird die Wiederherstellungsaufgabe durchgeführt, wobei alle persistenten Serverobjekte auf den letzten Zustand zurückgesetzt werden. Speziell im Prevayler System erfolgt die Ausführung derjenigen Operationen, die sich während der letzten Laufzeit angemeldet und aber noch nicht ausgeführt sind.

4.3.3 Leistung

Allein von einem Tupelraum ausgehend ist die Leistung wie im Abschnitt 3.6.1.3 diskutiert abhängig von der Anzahl der vorhanden Tupel im Tupelraum und der Anzahl der registrierten Vorlagen. Nach den Testergebnissen wie in Abbildung 4-6 gezeigt, wachsen die Antwortzeiten aller Operationen außer „out“ fast linear mit der Anzahl aller Tupel im Tupelraum. Und mit der Zunahme der Anzahl der registrierten Vorlagen schwanken die Antwortzeiten fast aller Operationen. Allerdings ist es noch zu merken, dass die Linien von „out“ und „in“ leicht steigt. Daraus folgt dass eine kleine Anzahl der registrierten Vorlagen nur kaum die Antwortzeit beeinflusst. Im Vergleich zur großen Menge der Tupel im Tupelraum kann dieser Einfluss ignoriert werden.

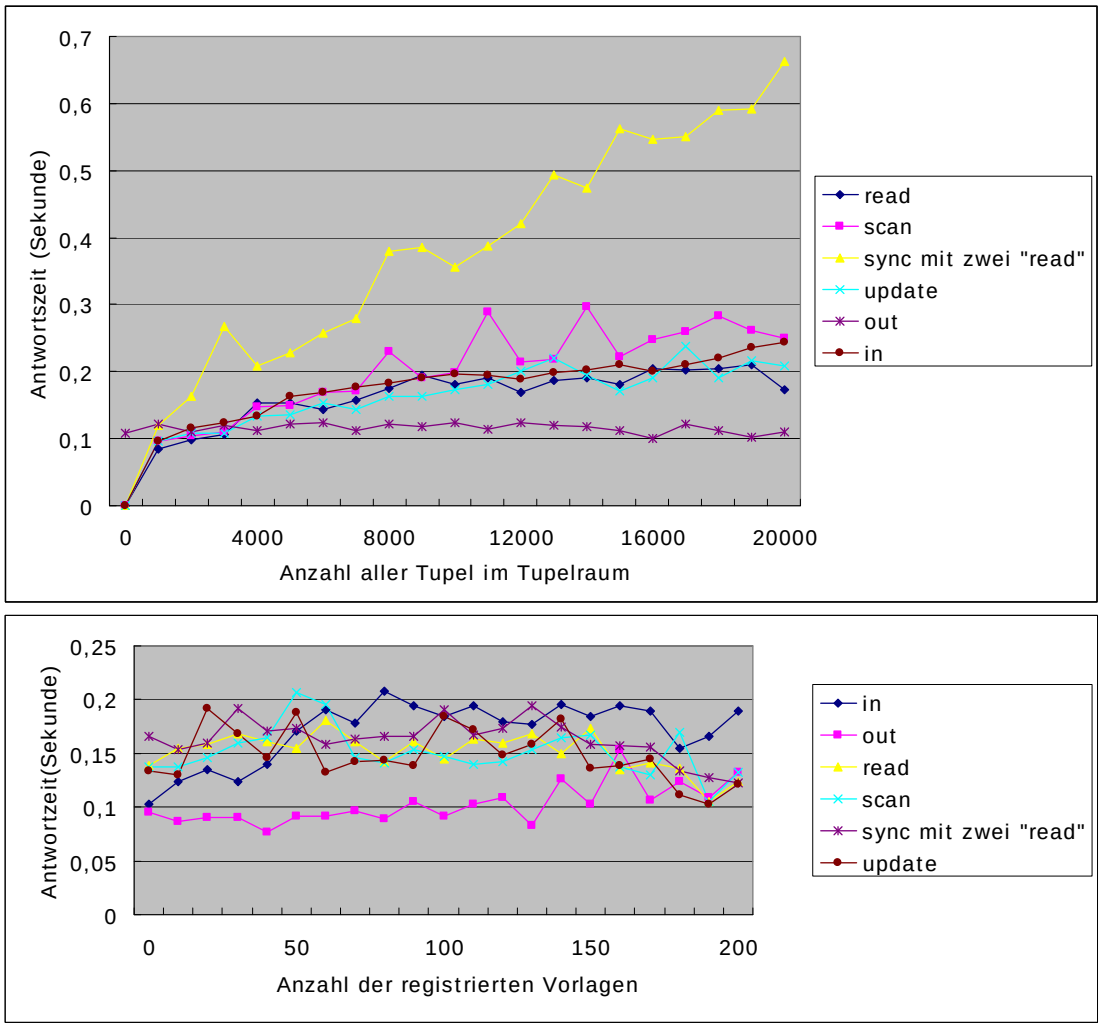

Abbildung 4-6: Antwortzeit aller Operationen

Im Hinblick auf mehrere Tupelräumen, die eine Middleware unterstürzen kann, können verteilte Prozesse verschiedene Tupelräume zu ihrem Kooperationsvermittler verwenden, sodass die zur Kooperation genutzten Nachrichten gut verteilt sind und der Durchsatz durch parallele Verarbeitung erhöht wird. Weiterhin wenn mehr als eine Middleware eingesetzt wird, können diese Middlewares auch auf ein lokales Netz verteilt werden, um Rechenauslastung gleichmäßig zu verteilen.

5 Fazit

In dieser Arbeit wurde die Middleware auf Basis vom erweiterten Tupelraum definiert, die es dem Anwendungsentwickler erleichtert, verteilte Anwendungen zu erstellen. Viele Schwierigkeiten der gemeinsamen Nutzung eines Speichers werden dem Entwickler abgenommen. Die Überschaubarkeit der Methoden vereinfacht die Einarbeitung in die Middleware. Der Entwickler kann sich auf das Protokoll seiner Anwendung konzentrieren, ohne sich um die Kommunikation und Synchronisation der beteiligten Clients zu kümmern.

Nach den Anforderungen der verteilten Workflow-Engine bietet die Middleware mehrere Unterstützungen wie Transaktion, Synchronisierung und XPath an, welche die Ausführung verteilter Prozessmodelle effizienter und auch zuverlässiger machen. Die Umsetzung der Kontrollkonstrukte – Sequence, Parallel Split, Synchronisation, Exclusive Choice, Simple Merge und Iteration – mit Tupelräumen wurde besprochen. Die Ergebnisse der den Pattern entsprechenden Testszenarien haben festgestellt, dass die Anforderungen erfüllt sind.

Jedoch reicht die Performance dieses Middleware-Servers mit erhöhter Belastung und großen Anzahlen von zu verwaltenden Tupeln für die reale Anwendung nicht aus. Und das Problem der Sicherheit zur Zugriffkontrolle einzelner Tupeln bleibt ungelöst.

Literaturverzeichnis

[1] Martin, D.; Wutke D.; Leymann F.: *A Novel Approach to Decentralized Workflow Enactment*. 12th IEEE International EDOC Conference (EDOC 2008). Munich, Germany, September 15 - 19, 2008.

[2] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems, 1985

[3] Christoph Bussler, *A Minimal Triple Space Computing Architecture*, DERI Technical Report 2005-04-22, 2005

[4] D. Khushraj, O. Lassila, T. Finin, sTuples: *Semantic Tuple Spaces*, IEEE, 2004

[5] Michael Benjamin Heidt, *Linda und JavaSpaces,* Seminararbeit, 2005

[6] Russell, N. et al. *Workflow Control-Flow Patterns: A Revised View*, BPM Center Report BPM-06-22, 2006

[7] D. Wutke, D. Martin, and F. Leymann. *Model and Infrastructure for Decentralized Workflow Enactment*. Proceedings of the 23rd ACM Symposium on Applied Computing (SAC'08), 2008

[8] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. *Decentralized orchestration of composite web services*. In WWW Alt. '04, pages 134-143, 2004

[9] Johannes Altaner, *Realisierung und Bewertung eines verteilten Ausführungsmodells für Web Services in XL*, Diplomarbeit, 2003

[10] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall PTR, 2000.

[11] G. Hohpe, B. Woolf, and K. Brown. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[12] Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/libray/ws-bpel/ .

[13] Van der Aalst, W. (1998) *The application of Petri nets to workflow management*. The Journal of Circuits, Systems and Computers 8 (1) 21-66.

[14] G. Coulouris, J. Dollimore, T. Kindberg. *Verteilte Systeme: Konzepte und Design*. Addison-Wesley Person Studium, 2002.

[15] RMI Architecture and Functional Specification http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

[16] Christian Ullenboom, *Java ist auch eine Insel,* Galileo Computing, 2007

[17] Wolfgang Emmerich : *Konstruktion von verteilten Objekten*. Dpunkt.verlag, 2003

[18] Prevayler Homepage, http://www.prevayler.org/

[19] J. Passing, *Object Prevalence und Prevayler,* Hasso Platterner Institut für Softwaresystemtecknik

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *„Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995

[21] Antony I. T. Rowstron und Alan Wood. *Solving the linda multiple rd problem*. In COORDINATION '96: Proceedings of the First

International Conference on Coordination Languages and Models, pages 357–367. Springer-Verlag, 1996

[22] D. Wutke, D. Martin, and F.Leymann. *Synchronizing Control Flow in a Tuplespace-Based, Distributed Workflow Management System*. Proceedings of the 10th international conference on Electronic commerce, 2008

[23] Paolo Ciancarini, *Distributed Programming with Logic Tuple Spaces*, Technical Report UBLCS-93-7, 1993

[24] W.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Workflow Patterns*. Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.

[25] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. *Business Process Execution Language for Web Service*s, version 1.1 (updated 01 feb 2005), 2005. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[26] RMI Dokumentation Webseite: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

[27] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation, 2000

[28] Wells, G. C.; Chalmers, A. G.; Clayton, P. G.: *Linda implementations in Java for concurrent systems*: Research Articles. In: Concurr. Comput. : Pract. Ex-per. 16 (2004), Nr. 10, S. 1005–1022

[29] Leymann, Frank: *Space-based Computing and Semantics: A Web Service Purist's Point-Of-View* / Universität Stuttgart, Germany. 2006.http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-05&engl=0

[30] Workflow Management Coalition: "*The Workflow Reference Model*", Document Number TC00-1003, Brüssel, Belgien, 1994.

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift: _____

Stuttgart, den 30.Oktober 2008