

Verteilte Linda-artige Koordination mit XML-Dokumenten

Diplomarbeit
Dirk Glaubitz
Dezember 2000

Technische Universität Berlin
Fachbereich Informatik
Institut für Kommunikations- und Softwaretechnik
Fachgebiet Formale Modelle, Logik und Programmierung (FLP)
Franklinstraße 28-29
D-10587 Berlin
Aufgabensteller: Prof. Dr. Bernd Mahr
Matrikel-Nummer: 147795

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.
Berlin, den 13. Dezember 2000

Unterschrift (Dirk Glaubitz)

Inhaltsverzeichnis:

<u>1. Einleitung</u>	5
<u>2. Koordination</u>	7
<u>2.1. Motivation</u>	7
<u>2.2. Der Koordinationsbegriff</u>	8
<u>2.3. Getrennte Betrachtung von Koordination und Berechnung bei der Systementwicklung</u> ...	9
<u>3. Linda</u>	11
<u>3.1. Das Koordinationskonzept von Linda</u>	11
<u>3.2. Eigenschaften von Linda</u>	13
<u>3.2.1. Entkopplung in Zeit und Raum</u>	13
<u>3.2.2. Verteilte Datenstrukturen</u>	14
<u>3.3. Anwendungen von Linda</u>	16
<u>4. Workspaces</u>	17
<u>4.1. Integration von Workflow- und Internettechnologien</u>	17
<u>4.2. Dokumentgetriebener Workflow</u>	17
<u>4.3. Verwendung von XML-Dokumenten</u>	18
<u>4.4. Workspace-Engines</u>	19
<u>4.5. Workspace Coordination Language (WSCL)</u>	19
<u>4.6. XMLSpaces</u>	20
<u>4.7. Stand von Workspaces</u>	22
<u>5. XMLSpaces</u>	23
<u>5.1. Eigenschaften von XMLSpaces</u>	23
<u>5.2. Verwendete Software/Technologie</u>	23
<u>5.2.1. Java</u>	24
<u>5.2.2. TSpace</u>	24
<u>5.2.3. XML4J</u>	24
<u>5.2.4. DOM</u>	24
<u>5.2.5. Matching-Engines</u>	25
<u>6. Erweiterung von Linda um XML-Dokumente</u>	26
<u>6.1. Konzeptionelle Erweiterung von Linda um XML-Dokumente</u>	26
<u>6.2. Matchingrelationen</u>	27
<u>6.2.1. Anfragesprachen</u>	27
<u>6.2.1.1. XPath</u>	28
<u>6.2.1.2. XPointer</u>	32
<u>6.2.1.3. XQL</u>	33
<u>6.2.1.4. XML-QL</u>	34
<u>6.2.2. DTD-Matching</u>	36
<u>6.2.3. Matching auf Gleichheit</u>	38
<u>6.2.4. Matching mit Umbenennen von Tags und Attributen</u>	38
<u>6.2.5. Matching mit einem Identifier im Wurzelement</u>	40
<u>6.2.6. Boolesche Verknüpfungen von Matchingrelationen</u>	40
<u>6.2.7. Namespaces und Matching</u>	40
<u>6.2.8. Vergleich/Zusammenfassung der Matchingrelationen</u>	43
<u>6.3. Realisierung in TSpace</u>	46
<u>6.3.1. Einordnung in ähnliche Projekte</u>	46
<u>6.3.2. TupleSpaces, Tuples und Fields in TSpaces</u>	46
<u>6.3.3. Überblick über die Erweiterung von TSpaces um XML-Dokumente</u>	48
<u>6.3.4. Die Klasse XMLDocField</u>	49
<u>6.3.5. Das Interface XMLMatchable</u>	51
<u>6.3.6. Beispiel</u>	52
<u>6.3.7. Die Klasse XMLAndMatchable</u>	52

<u>6.3.8. Anleitung zum Einfügen neuer Matchingrelationen in XMLSpace</u>	53
<u>6.3.9. Überblick über realisierte Matchingrelationen</u>	54
<u>7. Verteiltheit</u>	56
<u>7.1. Verteilungskonzept</u>	56
<u>7.2.1. Teilweise Replikation</u>	61
<u>7.2.1.1. Gitterstruktur</u>	61
<u>7.2.1.2. In-Protokoll</u>	62
<u>7.2.1.3. Rd-Protokoll</u>	63
<u>7.2.1.4. Out-Protokoll</u>	64
<u>7.2.1.5. Anmelden</u>	64
<u>7.2.1.5.1. Simulierte Nodes</u>	64
<u>7.2.1.5.2. Anmeldeverfahren</u>	66
<u>7.2.1.6. Abmelden</u>	69
<u>7.2.2. Verteiltheit ohne Replikation</u>	70
<u>7.3. Realisierung in TSpace</u>	71
<u>7.3.1. Integration des Distributors in TSpace</u>	71
<u>7.3.2. Das Distributor-Interface</u>	74
<u>7.3.3. Realisierung der teilweisen Replikation</u>	75
<u>7.3.4. Gesamtüberblick der Klassenstruktur</u>	77
<u>7.3.5. Starten des XMLSpace-Servers</u>	77
<u>7.3.6. Beispielablauf für die Client-seitige Verwendung des XMLSpaces bei Verteiltheit</u>	78
<u>7.3.7. Unterstützung sonstiger Methoden und Eigenschaften von TSpace</u>	78
<u>8. Verteilte Events</u>	81
<u>8.1. Behandlung der Events bei teilweiser Replikation</u>	81
<u>8.1.1. Anmelden neuer Events</u>	81
<u>8.1.2. Auslösen von Events</u>	81
<u>8.1.3. Abmelden von Events</u>	83
<u>8.1.4. Events und das Anmelden neuer Server</u>	83
<u>8.1.5. Events und das Abmelden von Servern</u>	85
<u>8.2. Behandlung der Events bei Verteilung ohne Replikation</u>	86
<u>8.3. Realisierung in TSpace</u>	86
<u>9. Integration von XMLSpaces in Workspaces</u>	90
<u>10. Ausblick und offene Punkte</u>	92
<u>10.1. Fehlertoleranz</u>	92
<u>10.2. Disconnected Operation</u>	93
<u>10.3. Umstieg von DOM1 auf DOM2</u>	94
<u>10.4. Unterstützung von Namespaces</u>	95
<u>10.5. Security</u>	95
<u>Anhang A: Die Konfigurationsdatei von TSpace</u>	97
<u>Anhang B: Verwendete Software</u>	100
<u>Anhang C: Interfaces</u>	101
<u>Literaturverzeichnis</u>	104

1. Einleitung

Durch die verteilte und parallele Zusammenarbeit von Rechnern entstehen Abhängigkeiten zwischen den Prozessen der einzelnen Rechner, die koordiniert werden müssen. In den Computerwissenschaften wurden eine Reihe von Koordinationsmodelle und –sprachen entwickelt, die sich ausschließlich um diese Koordinationsproblematik kümmern.

Linda, als eine der ersten Koordinationssprachen, hat ein sehr einfaches, aber auch sehr flexibles und mächtiges Koordinationskonzept in Form eines gemeinsamen Datenraumes, auf den die Prozesse der verschiedenen Rechner zugreifen können. Dieser gemeinsame Datenraum wird Tuplespace genannt und die einzelnen Prozesse können über die Befehle out, in und rd Tuples darin ablegen und daraus wiedergewinnen. Ein Tuple ist hierbei eine Sequenz von Feldern mit Werten von einfachen Datentypen, wie z.B. String und Integer. Zum Beispiel legt eine Operation out(<15,12>) ein Tuple <15,12> in dem Tuplespace ab. Über ein in(<15,?int>) kann das Tuple <15,12> wieder ausgelesen werden. <15,?int> ist hierbei ein Tuple-Muster, das das auszulesende Tuple beschreibt.

Diese Diplomarbeit untersucht die Fragestellung, ob und wie das Linda-Konzept um XML-Dokumente erweitert werden kann, d.h. ob statt der Tuples XML-Dokumente im Tuplespace abgelegt und daraus wiedergewonnen werden können. Als Motivation für diese Fragestellung dient die in [Tolksdorf00a, Tolksdorf 00b] beschriebene Workspaces-Architektur. Workspaces ist ein internetbasiertes Workflowmanagement-System, das ein lindaartiges Koordinationsmedium namens XMLSpace verwendet, in dem XML-Dokumente abgelegt und daraus wiedergewonnen werden können. Die Entwicklung dieses XMLSpace, das eine verteilte lindaartige Koordination mit XML-Dokumenten darstellt, ist das Thema dieser Diplomarbeit. Das XMLSpace wurde im Rahmen dieser Diplomarbeit auf der Basis von TSpace [TSpace], einer Linda-Implementierung in Java, realisiert.

Die Diplomarbeit hat folgenden Aufbau:

- Das Kapitel 2 gibt einen Überblick über Koordination: Warum ist Koordination notwendig und was ist Koordination eigentlich?
- Das Kapitel 3 beschreibt die Koordinationssprache Linda. Dazu gehört die Beschreibung des gemeinsamen Datenraumes Tuplespace, in dem die verschiedenen Prozesse über eine einfache Sprache Tuples ablegen und daraus wiedergewinnen können, und die Beschreibung der daraus entstehenden Eigenschaften.
- Das Kapitel 4 beschreibt die Workspaces-Architektur, die als Motivation für die Erweiterung von Linda um XML-Dokumente dient. Workspaces führt den Begriff des XMLSpaces ein, das eine verteilte lindaartige-Koordination mit XML-Dokumenten realisiert. Die Entwicklung von XMLSpaces ist Gegenstand dieser Diplomarbeit und wird in den folgenden Kapiteln 5 bis 8 beschrieben.
- Das Kapitel 5 gibt einen Überblick über die zu realisierenden Eigenschaften von XMLSpaces und die dafür verwendete Software.
- Das Kapitel 6 stellt die konzeptionelle Erweiterung von Linda um XML-Dokumente dar. Ein Schwerpunkt ist hierbei die Beschreibung XML-spezifischer Matchingrelationen, über die man angeben kann, welches XML-Dokument aus dem Tuplespace wiedergewonnen werden soll. Das Kapitel endet mit der Beschreibung der Umsetzung der konzeptionellen Erweiterung auf der Basis von TSpace.
- Das Kapitel 7 beschreibt Möglichkeiten für das Zusammenarbeiten mehrerer räumlich verteilter Tuplespaces, die zusammen ein logisches Tuplespace bilden sollen. Auf diese Art ist es möglich, daß ein Prozess nicht nur auf die Daten eines lokalen Tuplespace, sondern auch auf die Daten eines entfernten Tuplespace zugreifen kann. Dazu gehört die Beschreibung möglicher Verteilungsstrategien und die entsprechende Realisierung in Tspace.

- Kapitel 8 beschreibt die Erweiterung von Linda um verteilte Events, durch die es für einen Prozess möglich ist automatisch informiert zu werden, wenn ein bestimmtes Ereignis im verteilten Tuplespace eintritt, wie z.B. das Hinzufügen oder Löschen eines Tuples.
- Die vorherigen Kapitel haben die Realisierung von XMLSpace beschrieben, das einen Bestandteil der Workspaces-Architektur bildet. Kapitel 9 beschreibt die Integration von XMLSpaces in Workspace. Da die endgültige Realisierung der Integration in Workspaces nicht mehr Bestandteil dieser Diplomarbeit ist, wird die Integration nur vom Prinzip her andiskutiert.
- Kapitel 10 gibt einen Überblick über noch offene Punkte und Erweiterungsmöglichkeiten in der Zukunft.

2. Koordination

Dieses Kapitel gibt eine kurze Einleitung in die Koordinationsthematik.

Das Kapitel 2.1. beschreibt, warum Koordination notwendig ist. Was Koordination eigentlich bedeutet, erklärt das Kapitel 2.2. Das Kapitel 2.3. schließlich erläutert, daß die Entwicklung eines parallelen oder verteilten Systems als die Kombination von zwei verschiedenen zueinander orthogonalen Aktivitäten betrachtet werden kann: zum einen die Behandlung der Aspekte, die die Berechnungen der Prozesse betreffen, und zum anderen die Behandlung der Aspekte, die die Koordination der Abhängigkeiten zwischen den einzelnen Prozessen betreffen.

2.1. Motivation

Schon seit geraumer Zeit findet eine Entwicklung weg von zentralisierten oder isolierten Maschinen hin zu parallelen und verteilten Systemen statt, in denen die zuvor isolierten oder zentralisierten Maschinen über Netzwerke miteinander verbunden sind und komplexe Systeme bilden (Abbildung 2.1.).

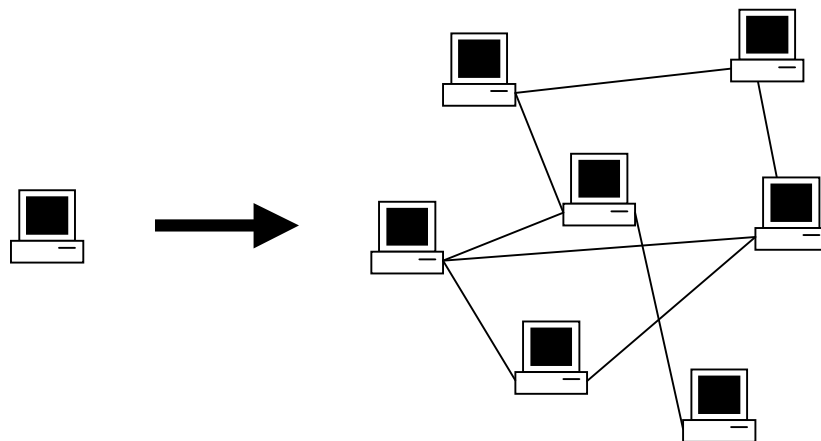


Abbildung 2.1.: Der Unterschied zwischen Früher und Heute.

Diese verteilten und parallelen Systeme bieten neue Anwendungsbereiche und Möglichkeiten. So ermöglichen z.B. parallele Systeme die schnellere Bearbeitung einer Aufgabe, indem mehrere Prozessoren an derselben Aufgabe arbeiten, und verteilte Systeme unterstützen die Benutzung von verteilten Ressourcen.

Gleichzeitig ergeben sich aus diesen parallelen und verteilten Systemen neue Herausforderungen für die Software- und Systementwicklung. Statt eines Prozesses, der sich nur um eine einzelne Aufgabe kümmert, hat man nun viele Prozesse, die gemeinsam an einer Aufgabe arbeiten. Daraus ergeben sich Abhängigkeiten zwischen den Prozessen und Maschinen, die koordiniert werden müssen. Die zu klärenden Fragen lauten also: Wie erfolgt die Koordination der Kooperation einer großen Anzahl von gleichzeitig aktiven Prozessoren bzw. wie werden die Abhängigkeiten zwischen den einzelnen Prozessen, die eine Anwendung bilden, gemanagt?

Die von den konventionellen Programmiersprachen angebotenen Koordinationsmechanismen sind nicht ausreichend, um vernünftig mit dieser Problematik umzugehen.

Es müssen Koordinationsmodelle- und sprachen entwickelt werden, die sich explizit mit der Koordination der Aktivitäten von großen Mengen von Prozessen, die eine einzelne Anwendung bilden, auseinandersetzen und diese Koordination erleichtern.

2.2. Der Koordinationsbegriff

Das Konzept der Koordination ist nicht nur auf Computerwissenschaften beschränkt (und innerhalb von Computerwissenschaften nicht nur auf parallele und verteilte Systeme).

Koordination kommt in vielen Bereichen vor: In Wirtschaft (z.B. in der Koordination von betrieblichen Abläufen), Biologie (z.B. beim Untersuchen des Verhaltens von Ameisenkolonien), Psychologie (z.B. beim Analysieren von Gruppenverhalten), Groupware (wie die Koordination von Menschen mit Hilfe von Computern erfolgt) und im täglichen Leben.

Malone und Crowston beschreiben in ihrem Papier [Malone94] Koordination unter einem interdisziplinären Gesichtspunkt und fordern die Entwicklung einer Koordinationstheorie. Diese Koordinationstheorie verwendet die Ideen von Koordination aus verschiedenen Disziplinen wie z.B. Computerwissenschaften, Wirtschaft, Biologie und erweitert sie. Ziel ist es, spezifische Koordinationsprozesse und –strukturen, die in allen Bereichen vorkommen, zu identifizieren und zu analysieren und Analogien zu finden. Fragestellungen lauten dabei z.B.: „Können Koordinationsstrukturen analog zu denen von Bienenwaben oder Ameisenkolonien für bestimmte Aspekte von Koordination in menschlichen Organisationen oder bei verteilten Systemen sinnvoll sein?“

Da Koordination ein breites Spektrum umfasst, gibt es viele Definitionen, was Koordination eigentlich ist.

Malone geben als einfachste und allgemeinste Definition an: „Coordination is managing dependencies between activities.“ (Koordination ist das Managen von Abhängigkeiten zwischen Aktivitäten).

In dem Gebiet der Programmiersprachen, kann man die von Carriero und Gelernter in dem Papier [Gelernter92] gegebene Definition verwenden: „Coordination is the process of building programs by gluing together active pieces“. Ein Koordinationsmodell ist dementsprechend: „A Coordination model is the glue that binds separate activities into an ensemble“.

Ein einfaches Beispiel für Koordination ist der Zugriff auf dieselbe begrenzte Ressource. Dieses Beispiel kommt sowohl im alltäglichen Leben vor, wenn z.B. zwei Personen dasselbe Buch in der Bibliothek ausleihen wollen, von dem aber nur noch ein Exemplar verfügbar ist, und auch in den Computerwissenschaften, wenn z.B. zwei Prozesse auf dieselbe Datei ohne Konflikte schreibend zugreifen wollen.

Entsprechend Malones Definition ist das Ausleihen eines Buches von einer einzelnen Person in der Bibliothek eine Aktivität und die Abhängigkeit zwischen den Aktivitäten entsteht dadurch, daß die Personen auf dieselbe Ressource, in diesem Fall das eine Buch, zugreifen wollen. Die Koordination ist der Umgang mit dieser Abhängigkeit. Mögliche Lösungen für das Managen dieser Abhängigkeiten wären z.B., daß derjenige, der zuerst am Ausleihtresen war, oder derjenige, der weniger längst überfällige Bücher hat, das Buch erhält.

In den Computerwissenschaften wurden (angefangen mit Linda Anfang der 80er Jahre[Gelernter85]) eine Reihe von Koordinationsmodellen und -sprachen entwickelt, die sich explizit um die Koordinationsproblematik kümmern. Das Papier [Papadopoulos98] enthält eine umfassende Übersicht über diese Koordinationsmodelle und –sprachen.

Der Zweck eines Koordinationsmodells und der entsprechend dazugehörigen Koordinationssprache ist es demnach, ein Mittel zum Integrieren einer Anzahl von möglicher Weise heterogenen Komponenten so zu bieten, daß die kollektive Menge eine einzelne Anwendung bildet, die auf einem parallelen oder verteilten System ausgeführt werden kann und daraus ihre Vorteile zieht.

Eine Koordinationssprache ist die entsprechend sprachliche Verkörperung eines Koordinationsmodells. Die Koordinationssprache bietet Unterstützung für das Kontrollieren von Synchronisation, Kommunikation, Erzeugen und Zerstören von Prozessen.

Der Begriff Koordination wird im Rahmen der Diplomarbeit verwendet, um sich auf das Management von Abhängigkeiten zwischen Aktivitäten durch Aktionen für Synchronisation, Kommunikation und dem Erzeugen und Zerstören von Prozessen zu beziehen.

2.3. Getrennte Betrachtung von Koordination und Berechnung bei der Systementwicklung

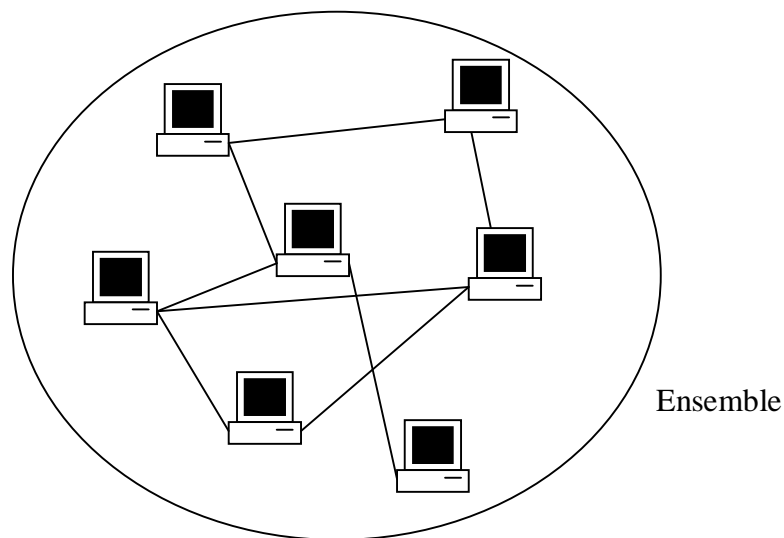


Abbildung 2.2.: Ein verteiltes System als Ensemble

Ein verteiltes oder ein paralleles System kann nach [Gelernter92] als ein asynchrones Ensemble (kurz genannt Ensemble) betrachtet werden. Ein Ensemble ist eine Sammlung von asynchronen Aktivitäten, die miteinander kommunizieren. Eine Aktivität ist z.B. ein Programm, Prozess oder ein Thread.

Ein Ensemble kann heterogen sein, weil verschiedene Arten von Komponenten miteinander agieren können, und Ensembles können auch verteilt sein, weil eine Komponente die Dienste einer anderen entfernten Komponente aufrufen kann.

Die Entwicklung eines verteilten oder parallelen Ensembles kann als die Kombination von zwei verschiedenen, zueinander orthogonalen Aktivitäten betrachtet werden:

- Der *Berechnungs-Teil* beschreibt die Aktivitäten eines Prozesses durch die Manipulierung von Daten. Die Berechnung wird durch eine einfache konventionelle Programmiersprache, wie z.B. C, Modula oder Java verkörpert.
- Der *Koordination-Teil* beschreibt die Koordination zwischen den Prozessen, d.h. die Kommunikation und Kooperation zwischen den Prozessen, die als Black-Boxes betrachtet werden. Der Koordinations-Teil abstrahiert von der Berechnung und damit von den Internas der Aktivitäten. Der Koordinations-Teil wird durch ein Koordinationsmodell repräsentiert, das sich um Synchronisation, Kommunikation und die Erzeugung und Zerstörung von Prozessen kümmert. Das in Kapitel 3 beschriebene Linda ist ein Beispiel für ein Koordinationsmodell.

Berechnung und Koordination werden also getrennt und orthogonal zueinander betrachtet. Man erhält eine getrennte Sichtweise auf dasselbe System: Wenn man sich um Koordination kümmert, muß man sich nur um Aspekte und Probleme kümmern, die die Koordination betreffen, und kann dabei alles, was die internen Details und Probleme der Berechnung betrifft, ignorieren. Entsprechendes gilt, wenn man die Berechnung betrachtet. Dies führt zur

Reduzierung der Komplexität bei der Entwicklung eines verteilten oder parallelen Systems und damit zu einer Vereinfachung.

Der Berechnungs- und Koordinations-Teil bieten damit zwei unterschiedliche, sich jedoch ergänzende Sichten auf dasselbe System, die erst zusammen betrachtet ein komplettes Bild ergeben.

Daß die beiden Sichten nur zusammen ein komplettes Bild ergeben, kann man sich dadurch klar machen, daß eine Berechnung keinen Sinn ergibt, wenn das Ergebnis niemanden mitgeteilt werden kann (was Aufgabe der Koordination ist) und Koordination keinen Sinn ergibt, wenn es keine zu koordinierenden Aktivitäten gibt.

3. Linda

Linda wurde Anfang der 80er auf der Yale-Universität entwickelt [Gelernter85] und ist historisch gesehen eine der ersten echten Koordinationssprachen, d.h. Linda konzentriert sich auf den Koordinationsaspekt und abstrahiert von Berechnungsaspekten.

Die Koordination wird realisiert durch die Verwendung eines gemeinsamen Datenraumes namens Tuplespace und durch die Bereitstellung einer einfachen Sprache, die im wesentlichen aus den 3 Operationen out, in und rd besteht und zum Platzieren und wiedergewinnen von Daten (sogenannten Tuples) im Tuplespace dient.

Diese Linda-Operationen werden zu einer Programmiersprache hinzugefügt und können zur Koordination verschiedener Prozesse verwendet werden. Die Prozesse in einem Linda-System verwenden also die Programmiersprache, um Berechnungen auszudrücken, und das Tuplespace und die dafür definierten Operationen, um die Koordination zwischen den Prozessen zu realisieren.

Da die durch die Linda-Operationen gebildete Koordinationssprache komplett unabhängig von der Programmiersprache (der Berechnungssprache) ist, kann vom Prinzip her jede Programmiersprache (egal ob imperativ, logisch, funktional oder objektorientiert) problemlos um Linda erweitert werden. Beispiele für Erweiterungen von Programmiersprachen um Linda sind: C-Linda, Fortran-Linda, Prolog-Linda, JavaSpaces, TSpace.

Wie in [Gelernter92] beschrieben, ermöglicht die Trennung von Berechnung und Koordination in einer jeweils eigenen Sprache die Unterstützung von Portabilität (weil es einfacher ist zwischen den Programmiersprachen zu wechseln, weil alle dasselbe Koordinationskonzept verwenden) und Heterogenität (weil das Koordinationskonzept unabhängig von Programmiersprache und Maschinenarchitektur ist und vom Prinzip her mit allen zusammenarbeiten kann).

3.1. Das Koordinationskonzept von Linda

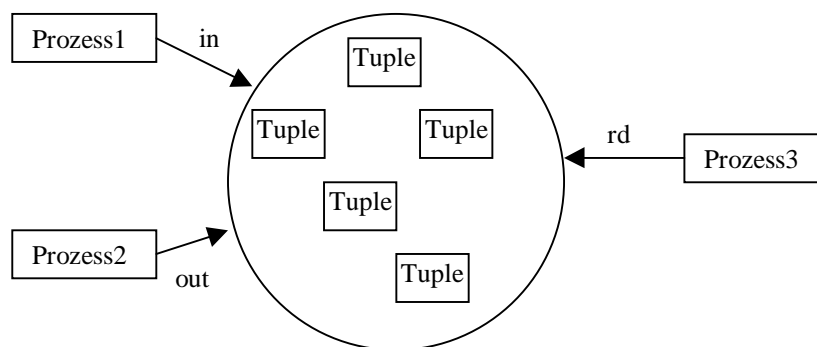


Abbildung 3.1. Das Tuplespace

Das Medium der Koordination ist in Linda der sogenannte Tuplespace. Ein Tuplespace ist ein gemeinsamer Datenraum bzw. Speicher, auf den alle Prozesse zugreifen können. In diesem Tuplespace können Tuples platziert und daraus wiedergewonnen werden. Alle Synchronisation und Kommunikation zwischen den Prozessen wird durch das Platzieren und Wiedergewinnen von Tuples in und aus dem Tuplespace verrichtet.

Ein Tuple ist eine Sequenz von Feldern mit Werten von einfachen Datentypen, wie z.B. String oder Integer. Ein Beispiel für ein Tuple ist $\langle \text{„Hallo Welt“}, 3 \rangle$. Dieses Tuple besteht aus zwei Feldern. Das erste Feld hat als Wert den String „Hallo Welt“ und das zweite Feld den Zahlenwert 3.

Für den Zugriff auf das Tuplespace sind 4 Operationen definiert: out, in, read und eval.

Über out(t) kann ein Prozess ein Tuple t zum Tuplespace hinzufügen. Out(t) ist nicht blockierend, d.h. ein Prozess kann sofort nach dem Aufruf von Out(t) mit seiner Verarbeitung fortfahren.

Über die in(t)-Operation kann ein Tuple aus dem Tuplespace wiedergewonnen werden. Das Tuple wird aus dem Tuplespace entfernt und ist nicht mehr für andere Prozesse verfügbar.

Die Angabe, welches Tuple aus dem Tuplespace wiedergewonnen werden soll erfolgt über das sogenannte *associative pattern matching*, bei dem das wiederzugewinnende Tuple nicht anhand eines Adresswertes, sondern anhand des Inhalts des Tuples herausgesucht wird. Zu diesem Zweck wird ein Template verwendet, das den Inhalt des wiederzugewinnenden Tuples beschreibt.

Ein Template ist wie ein Tuple eine geordnete Menge von Feldern, das jedoch auch formale Felder (genannt *Formals*) enthalten kann. Ein formales Feld wird durch einen Typ beschrieben und ist ein Platzhalter für einen Wert dieses spezifischen Typs. Felder, die als Inhalt einen Wert und nicht nur einen Typ haben werden dagegen *Actuals* genannt.

Ein Beispiel für ein Template ist $\langle \text{„Hallo Welt“}, \text{int} \rangle$. Das erste Feld ist ein Actual mit dem Wert „Hallo Welt“ und das zweite ein Formal (ein Platzhalter für irgendeinen Wert des Typs Integers). Der Aufruf von $\text{in}(\langle \text{„Hallo Welt“}, \text{int} \rangle)$ soll ein Tuple, das als ersten Feldwert den String „Hallo Welt“ und als zweiten Feldwert irgendeinen Integerwert enthält, aus dem Tuplespace holen.

Ob ein Template mit einem Tuple im Tuplespace matcht wird von einer Matchingrelation gesteuert. Ein Tuple t und ein Template t' matchen, wenn beide die gleiche Anzahl von Feldern haben und wenn jedes Feld von t mit dem entsprechenden Feld von t' matcht. Ein Feld von t matcht mit dem entsprechenden Feld von t', wenn beide einen Feldwert mit dem gleichen Wert enthalten oder wenn das Feld im Template t' ein formales Feld ist und das entsprechende Feld im Tuple t einen Wert des gleichen Typs enthält.

Zum Beispiel matcht das Template $\langle \text{„Hallo Welt“}, ?\text{int} \rangle$ mit dem Tuple $\langle \text{„Hallo Welt“}, 2 \rangle$ aber nicht mit dem Tuple $\langle \text{„Hallo Welt“}, 2, \text{„Hallo“} \rangle$ und auch nicht mit dem Tuple $\langle 2, \text{„Hallo Welt“} \rangle$.

Der Matching-Algorithmus sucht entsprechend dieser Matchingrelation ein passendes Tuple aus dem Tuplespace heraus. Wenn kein passendes Tuple gefunden wird, wird der aufrufende Prozess von in(t) solange blockiert, bis ein passendes Tuple t' im Tuplespace verfügbar ist. Wenn mehrere passende Tuples vorhanden sind, wird einer nicht-deterministisch ausgewählt.

Nachdem ein passendes Tuple gefunden wurde, werden die Werte von den Actuals im gefundenen Tuple den Formals im Template zugewiesen und der Prozess, der die in-Operation aufgerufen hat, kann in seiner Verarbeitung fortfahren.

Rd(t) hat dieselbe Funktionalität wie in(t), nur daß das matchende Tuple im Tuplespace erhalten bleibt. Der aufrufende Prozess erhält also eine Kopie des Tuples als Ergebnis zurück.

Über die Operationen out(t), in(t) und rd(t) können keine Tuples im Tuplespace geändert werden. Um ein Tuple zu ändern, muß zunächst das zu ändernde Tuple über ein in(t) aus dem Tuplespace herausgeholt werden. Dieses Tuple kann dann geändert und über out(t) wieder zum Tuplespace hinzugefügt werden. Dadurch können mehrere Prozesse gleichzeitig auf das Tuplespace zugreifen, ohne daß read/write-Konflikte auftreten können, d.h. es ist nicht möglich daß zwei Prozesse gleichzeitig dasselbe Tuple verändern und sich dabei negativ beeinflussen können, weil ein Prozess das Tuple zunächst aus dem Tuplespace entfernen muß, ehe er es verändern kann.

Linda bietet neben in, out und rd auch eine eval-Operation, über die ein sogenanntes *aktives Tuple* zum Tuplespace hinzugefügt werden kann. Ein aktives Tuple kann als Felder Referenzen auf Funktionen enthalten. Nachdem das aktive Tuple über eval zum Tuplespace

hinzugefügt wurde, werden die im aktiven Tuple referenzierten Funktionen parallel ausgeführt und die Ergebniswerte der Funktionen ersetzen die Funktionsreferenzen im aktiven Tuple als Feldwerte. Nachdem alle Funktionsreferenzen durch die entsprechenden Ergebniswerte ersetzt wurden, ist das Tuple als normales *passives* Tuple verfügbar. Zum Beispiel resultiert der Aufruf von `eval(square(3), square(4))`, wobei `square` als eine Funktion zum Quadrieren der Parameterwerte definiert ist, in dem Platzieren des Tuples $\langle 9, 16 \rangle$ im Tuplespace.

Die Operation `eval` spielt für den Rest der Diplomarbeit keine Rolle und wird daher nicht weiter betrachtet.

3.2. Eigenschaften von Linda

Linda realisiert das sogenannte *generative communication* Paradigma: Wenn zwei Prozesse Daten miteinander austauschen wollen, dann erzeugt (*generates!*) der Sender ein neues Tuple und platziert es in dem Tuple Space, auf den alle Prozesse zugreifen können. Der Empfänger kann dann dieses Tuple aus dem Tuplespace herausholen.

Das in dem Tuplespace platzierte Tuple existiert unabhängig von dem erzeugenden Prozess, bis ein anderer Prozess das Tuple aus dem Tuplespace entfernt. Man hat es also mit persistenten Objekten und nicht mit transienten Messages zu tun.

Aus diesem generative communication Paradigma ergeben sich einige vorteilhafte Eigenschaften, die im folgenden beschrieben werden.

3.2.1. Entkopplung in Zeit und Raum

Da die Kommunikation zwischen den Prozessen nur über den gemeinsamen Datenraum Tuplespace erfolgt und die darin platzierten Tuples unabhängig von den verschiedenen Prozessen existieren, wird eine Entkopplung der Prozesse in Zeit und Raum erreicht.

Entkopplung im Raum bedeutet, dass kein Prozess die Identität (bzw. Adresse) der anderen Prozesse kennen muss. Ein Prozess, der ein Tuple im Tuplespace erzeugt, muss nicht im Vorhinein wissen, welcher andere Prozess das Tuple ausliest. Ebenso muss ein Prozess, der ein Tuple aus dem Tuplespace ausliest nicht wissen, wer dieses Tuple erzeugt hat. Sender und Empfänger müssen also nicht im Vorhinein die Identität voneinander kennen und auch nicht explizit angeben. Die Kommunikationspartner bleiben anonym füreinander. Linda führt daher auch kein Konzept zur Identifizierung der Prozesse ein. Statt dessen kann ein Prozess das Tuple mit einem logischen Namen versehen, indem er z.B. als ersten Feldwert "Ergebnis für Teilaufgabe X" verwendet. Dieses Tuple wird über ein `out(<„Ergebnis der Teilaufgabe X“,...>)` im Tuplespace platziert, ohne dass sich der erzeugende Prozess irgendwelche weiteren Gedanken machen muss, wer dieses Tuple ausliest. Der Prozess, der das "Ergebnis für Teilaufgabe X" benötigt, kann das Tuple auslesen ohne wissen zu müssen, wer es erzeugt hat, indem er ein `in(<„Ergebnis der Teilaufgabe X“,...>)` verwendet.

Dadurch sind die Prozesse unabhängig im Raum voneinander, weil sich ein Prozess nicht direkt auf den anderen beziehen muß. Dies bedeutet auch, daß räumliche Verteiltheit unterstützt wird, da die verschiedenen Prozesse, die auf das Tuplespace zugreifen, in unterschiedlichen Addressräumen bzw. auf unterschiedlichen Maschinen existieren können.

Durch die Entkopplung im Raum entstehen Sicherheitsprobleme, weil jeder Prozess auf jedes Tuple zugreifen kann und die Prozesse nichts davon merken. Linda geht jedoch zunächst davon aus, dass die verschiedenen Prozesse, die auf das Tuplespace zugreifen, sich gegenseitig vertrauen. Entsprechende Sicherheitsmaßnahmen, wie z.B. die Kontrolle des Zugriffs auf ein Tuplespace über Authentifizierung ist nicht Bestandteil vom Linda-Konzept, aber Thema zahlreicher Erweiterungen von Linda.

Die Entkopplung in der Zeit bedeutet, daß die miteinander über das Tuplespace kommunizierenden Prozesse nicht zur gleichen Zeit existieren/aktiv sein müssen. Das wird

dadurch erreicht, daß die Tuples im Tuplespace unabhängig von den Prozessen existieren. Ein Prozess A, der ein Tuple t über $out(t)$ im Tuplespace platziert, kann anschließend sofort mit seinen anderen Aufgaben fortfahren oder auch terminieren. Das Tuple bleibt solange im Tuplespace erhalten, bis es über ein entsprechendes $in(t')$ von einem anderen Prozess B entfernt wird. Prozess A (Sender) und Prozess B (Empfänger) sind damit zeitlich unabhängig voneinander und können zu unterschiedlichen Zeiten aktiv sein.

Linda ist bzgl. der Entkopplung in Zeit und Raum allgemeiner als RPC. RPC fordert im Gegensatz zu Linda, dass der Sender im Vorhinein weiß, welcher Prozess der Empfänger ist, und daß er an diesen Prozess explizit seine Message sendet. Außerdem verlangt RPC, daß der sendende und empfangende Prozess zur gleichen Zeit aktiv sein müssen.

Das RPC-Schema läßt sich in Linda nachbilden [Gelernter85]:

Remote procedure call:

Out (P, me, out-actuals);

In(me, in-formals)

Remote procedure:

In(P, who:name, in-formals)

[body of procedure P;

out(who, out-actuals);

]

Hierbei führt der Aufrufer der Prozedur ein $out()$ aus, um die Parameter zu senden und führt ein anschließendes $in()$ aus, um das Ergebnis zu erhalten. Der Prozess, auf dem die Prozedur aufgerufen wird, holt sich über ein $in()$ die Parameter aus dem Tuplespace, führt die Prozedur aus und steckt über ein $out()$ das Ergebnis ins Tuplespace zurück. Sender und Empfänger identifizieren sich durch die Verwendung logischer Tuplenamen.

Durch die von Linda realisierte Entkopplung in Raum und Zeit wird der Programmierer beim Entwickeln eines Programms so weit wie möglich von der Behandlung von räumlichen und zeitlichen Beziehungen zwischen Prozessen entbunden. Dies erleichtert die parallele Programmierung, weil man nicht gleichzeitig über mehrere Prozesse (nämlich die datengenerierenden und datenkonsumierenden Prozesse) nachdenken muß, sondern die Prozesse weitestgehend unabhängig voneinander entwickeln kann.

Ein weiterer Vorteil ist die Unterstützung von dynamischer Anpassung im System, weil Prozesse problemlos hinzukommen und wegfallen können, weil die Prozesse unabhängig voneinander in Raum und Zeit sind.

3.2.2. Verteilte Datenstrukturen

Der Tuplespace in Linda realisiert verteilte Datenstrukturen. Verteilte Datenstrukturen sind Datenstrukturen, auf denen mehrere Prozesse gleichzeitig zugreifen können. Auf diese Art ist es auch möglich, daß sich Prozesse, die sich in verschiedenen Adressräumen befinden, Daten teilen können.

Da durch die Operationen in , out und rd kein Tuple direkt *im* Tuplespace geändert werden kann, wird sichergestellt, daß nie zwei Prozesse gleichzeitig schreibend auf die gleichen Tuples zugreifen können und dadurch werden Read/Write-Konflikte verhindert. Wenn ein Prozess ein Tuple im Tuplespace verändern will, muß er zunächst den Tuple über ein $in()$ aus dem Tuplespace herausholen. Die Operation $in()$ stellt sicher, daß für den Fall, daß mehrere Prozesse auf dasselbe Tuple zugreifen wollen, nur ein Prozess erfolgreich ist.

Andere Konzepte, wie z.B. RPC, verbieten verteilte Datenstrukturen und fordern, daß gemeinsame Datenstrukturen in Manager-Prozessen gekapselt werden. Bei Manager-Prozessen müssen alle Prozesse den Zugriff auf die gemeinsame Datenstruktur über den Manager abwickeln. Operationen, die sicher von vielen Prozessen parallel ausgeführt werden

können, müssen von dem einzelnen Manager-Prozess einem nach dem anderen ausgeführt werden.

Manager-Prozesse können in Linda nachgebildet werden, aber es gibt Fälle, in denen dies nicht gefordert und auch nicht gewünscht ist und eine verteilte Datenstruktur die natürlichere und effizientere Lösung ist.

Ein einfaches Beispiel für die Verwendung einer verteilten Datenstruktur ist die in [Ahuja86] beschriebene Realisierung einer Matrixmultiplikation.

Angenommen A und B sind jeweils eine Matrix und es soll $A * B$ berechnet werden.

Der in [Ahuja86] beschriebene Algorithmus besteht aus einem Initialisierungs-, einem Aufräum- und beliebig vielen Worker-Prozessen. Der Initialisierungsprozeß führt Vorbereitungen für die Berechnung durch. Die einzelnen Workerprozesse realisieren die eigentliche Matrixmultiplikation und der Aufräumprozess nimmt die von den einzelnen Workerprozessen erzeugten Teilergebnisse und erzeugt daraus ein Gesamtergebnis.

Der Initialisierungsprozeß fügt über out-Aufrufe die einzelnen Zeilen von A und die einzelnen Spalten von B in das Tuplespace ein. Das Tuplespace enthält anschließend Tuples der Form:

```
<„A“, 1, A's erste Zeile>
<„A“, 2, A's zweite Zeile>
...
<„B“, 1, B's erste Spalte>
<„B“, 2, B's zweite Spalte>
...
```

Das zweite Tupelfeld entspricht einem Indize, so daß ein Prozess die i-te Spalte oder j-te Zeile auswählen kann.

Der Initialisierungsprozeß fügt ein Tuple `<„Next“, 1>` hinzu, das angibt, welches Element als nächstes berechnet werden soll.

Jeder Worker-Prozess entscheidet anhand des `<„Next“, ...>`-Tuples, welches Element es berechnen soll. Die Worker-Prozesse versuchen über `in(<„Next“, int nextElem>)` auf das `<„Next“,...>`-Tuple zuzugreifen, wobei nur ein Worker-Prozess erfolgreich ist. Der erfolgreiche Worker-Prozess kann anhand des Integerwertes bestimmen, welches Element er berechnen soll. Er erhöht den Integerwert um 1 und steckt das Tuple über `out(<„Next“, ...>)` ins Tuplespace zurück, so daß ein anderer Work-Prozess es über ein `in(...)` herausholen kann und das nächste Element berechnen kann:

```
in(„Next“, int nextElem)
out(„Next“, nextElem + 1)
```

Der Worker-Prozess kann anhand des Wertes `nextElem` und der Größe der Matrix bestimmen, welches Teilergebnis er berechnen soll:

```
i = (nextElem - 1)/dim + 1;
j = (nextElem - 1)%dim + 1;
```

Der Workerprozess berechnet das Matrixelement (i,j) der Ergebnismatrix, indem er die i-te Spalte von A und die j-te Zeile von B aus dem Tuplespace liest und die entsprechenden Werte von Zeile und Spalte miteinander multipliziert und aufaddiert. Das Teilergebnis wird über ein out zum Tuplespace hinzugefügt:

```
rd(„A“, i, array zeile)
```

```
rd(„B“, j, array spalte)
out(„result“, I, j, DotProduct(zeile, spalte))
```

Der Workerprozess kann nun über ein erneutes `in(<“Next”, int nextElem>)` das nächste zu berechnende Element bestimmen usw.

Der Aufräumprozess sammelt die einzelnen Teilergebnisse über ein `in()` ein und fügt sie zu einem Gesamtergebnis zusammen:

```
for (row = 1; row <= NumRows; row++)
    for (col = 1; col <= NumCols; col++)
        in(„result“, row, col, array prod[row][col])
```

In diesem Beispiel stellen die Tuples, die die einzelnen Zeilen der Matrix A und die einzelnen Spalten der Matrix B repräsentieren, die verteilte Datenstruktur dar. Alle Worker-Prozesse können gleichzeitig lesend darauf zugreifen.

Das Beispiel der Matrixmultiplikation demonstriert das sogenannte Replicated-Worker-Programmierparadigma. Man hat eine bestimmte Anzahl replizierter Arbeiter, die alle vom Prinzip her das gleiche machen und gleichzeitig auf die verteilte Datenstruktur zugreifen, um nach Arbeit zu suchen (ein ähnliches Konzept verfolgt auch das im nächsten Kapitel beschriebene Workspaces).

Der Vorteil eines solchen Programmes sind die guten Skalierungsmöglichkeiten (das Programm läuft genauso gut jedoch schneller mit 100 als mit 10 Worker-Prozessen) und das Unterstützen von dynamischen Load-Balancing (die Aufgaben werden zur Laufzeit zwischen den verfügbaren Workern aufgeteilt und jeder Worker ist optimal ausgelastet).

3.3. Anwendungen von Linda

Linda ist eine Koordinationssprache, die nicht nur auf ein bestimmtes Anwendungsgebiet beschränkt ist, sondern sich für ein breites Spektrum von Anwendungsgebieten eignet.

Anfänglich wurde Linda schwerpunktmäßig für parallele Anwendungen verwendet, aber es hat sich gezeigt, daß Linda auch für verteilte und nebenläufige Anwendungen, sowie Groupware geeignet ist [Hupfer91]. So beschreiben z.B. das Papier [Banville] und das nächste Kapitel über Workspaces die Verwendung von Linda als Groupware und im Kontext von Workflow-Managementsystemen.

4. Workspaces

Die Workspaces-Architektur [Tolksdorf00a, Tolksdorf00b] dient zum Koordinieren von Arbeit über das Web und kombiniert Internet- und Workflowtechnologien. Gleichzeitig zeigt Workspaces, wie sinnvoll die Koordinationstechnologie von Linda im Kontext von Workflowmanagementsystemen sein kann.

Die in diesem Kapitel gemachten Beschreibungen zur Workspace-Architektur geben sinngemäß die in [Tolksdorf00a, Tolksdorf00b] gemachten Ausführungen zur Workspace-Architektur wieder. Bestimmte Abläufe werden hierbei an dem eigenen Beispiel eines Workflows zur Bewertung eines Exposé veranschaulicht.

Das Kapitel beginnt mit einer kurzen Beschreibung der Vorteile, die sich aus der Kombination von Workflow- und Internettechnologien ergeben, erläutert dann die Workspace-Architektur, die auf einem dokumentgetriebenen Workflow basiert, und beschreibt schließlich den lindaartigen XMLSpace, der zur Koordination zwischen den Workflow-Engines dient, und im Rahmen dieser Diplomarbeit entwickelt werden soll.

4.1. Integration von Workflow- und Internettechnologien

Das WfMC (Workflow Management Coalition) hat in dem Papier [WfMC98b] über die Notwendigkeit der Integration von Workflow- und Internettechnologien diskutiert und die beiden Technologien als sich gegenseitig ergänzend erkannt.

Internetanwendungen sind in der Regel gut geeignet für kurzlebige Interaktionen zwischen dem Benutzer und dem Computer-System, aber unterstützen keine lang laufenden Interaktionen. Workflowsysteme dagegen managen langlebige, prozessorientierte Anwendungen.

Durch die Integration von Internet- und Workflowtechnologien werden Internetanwendungen um Workflowfunktionen ergänzt, wie z.B. die Repräsentation und Interpretation von Arbeitsabläufen, das Führen von Statistiken, Überwachungsfunktionen für Abläufe und das Alarmieren bei fehlerhaften Abläufen.

Daraus resultieren Vorteile für die Internetanwendungen bzgl. Produktivität und Qualitäts- und Kostenkontrolle.

Auf der anderen Seite bringt das Internet dem Workflow die Vorteile seiner weltweiten Infrastruktur, *Zero-Application-Deployment-Costs* und die Möglichkeit der Realisierung von Virtual Enterprises.

Die in dem nächsten Kapitel beschriebene Workspace-Architektur kombiniert Workflowtechnologie mit Internetstandards wie XML und XSL um ein internetbasiertes Workflowmanagementsystem zu realisieren, das die vom WfMC erwarteten Vorteile/Nutzen zur Verfügung stellt.

4.2. Dokumentgetriebener Workflow

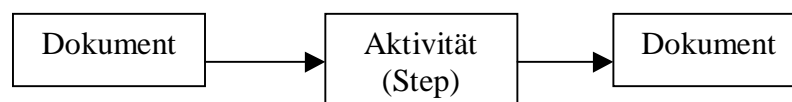


Abbildung 4.1.: Dokumentgetriebener Workflow (aus [Tolksdorf00a])

Workspaces basiert auf einem dokumentgetriebenen Workflow, d.h. der Workflow wird als eine Serie von Dokument-Transformationen verstanden.

Jede Aktivität in einem Workflow wird durch das Vorhandensein bestimmter Dokumente gestartet, führt Transformationen auf diese Dokumente aus und erzeugt als Ergebnis Dokumente, die wiederum Eingabe-Dokumente für andere Aktivitäten sind (Abbildung 4.1.).

Eine einzelne Aktivität eines Workflows wird in Workspaces Schritt genannt. Es wird zwischen Basic- und Koordinationsschritten unterschieden. Basic-Schritte repräsentieren die tatsächlich zu verrichtende Arbeit und Koordinationsschritte managen die Abhängigkeiten zwischen den Basic-Schritten und koordinieren damit den Arbeitsfluß.

Basic-Schritte lassen sich unterteilen in:

- Automatische Schritte, die automatisch vom System ausgeführt werden (z.B. das Umwandeln eines Textes in ein anderes Format).
- Externe Schritte, die eine Anwendung starten und den Benutzer darüber auf die Eingabe-Dokumente eine Aktivität ausführen lassen (z.B. Änderungen an einem Text über einen Editor).
- Benutzerschritte, die vom Benutzer allein ohne Unterstützung des Systems verrichtet werden (z.B. eine Idee entwickeln). Der Benutzer teilt die Beendigung eines solchen Schrittes über ein grafische Benutzerschnittstelle dem System mit.

Die Basic-Schritte werden durch die Koordinationsschritte koordiniert, die die Reihenfolge der Ausführung im Workflow beeinflussen. Abbildung 4.2. zeigt zwei Arten von Koordinationsschritten als Beispiel: JOIN und SPLIT. Ein JOIN synchronisiert mehrere Schritte, indem es mehrere Dokumente von verschiedenen Schritten als Eingabe nimmt und ein Ausgabe-Dokument erzeugt. Der SPLIT dagegen führt das genaue Gegenteil aus, indem er mehrere Schritte durch das Erzeugen mehrerer identischer Ausgabe-Dokumente aus einem Eingabe-Dokument startet. Die Koordinationsschritte bilden zusammen die Koordinationssprache zum Koordinieren der Basic-Schritte im Workflow.

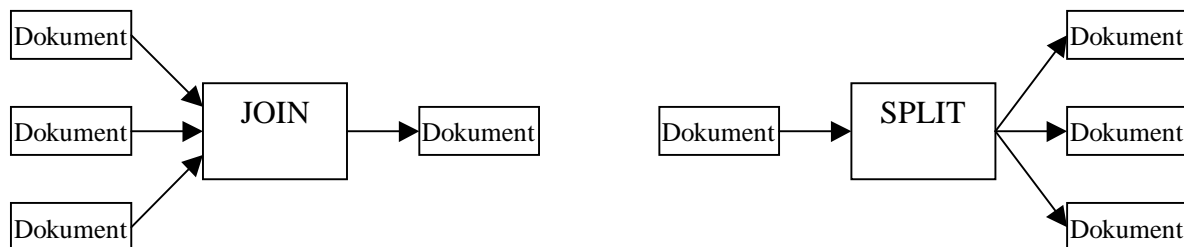


Abbildung 4.2.: Koordinationsschritte

4.3. Verwendung von XML-Dokumenten

Innerhalb von Workspaces sind alle Dokumente auf denen Arbeit verrichtet wird und damit alle Eingabe- und Ausgabe-Dokumente eines Schrittes XML-Dokumente. Abbildung 4.3. zeigt einen Schritt namens „Schreibe Kommentar“, der ein XML-Dokument als Eingabe erhält. Dieses XML-Dokument enthält einen Text, zu dem der Benutzer, der diesen Schritt ausführt, einen Kommentar schreiben soll. Der Schritt erzeugt als Ausgabe ein XML-Dokument, das den vom Benutzer geschriebenen Kommentar enthält.

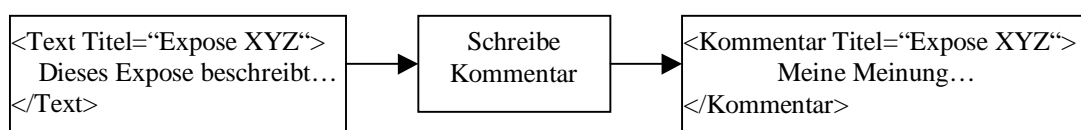


Abbildung 4.3.: Verwendung von XML-Dokumenten

4.4. Workspace-Engines

Für die Ausführung eines Schritts ist eine Engine verantwortlich. Die Engine wartet auf das Vorhandensein der notwendigen XML-Dokumente für das Ausführen des Schritts, führt den Schritt darauf aus und gibt die durch den Schritt erzeugten XML-Dokumente als Ergebnis zurück.

Die Engine ist mit Hilfe von XSL-Technologie realisiert: Ein Schritt entspricht der Transformation von XML-Dokumente in andere XML-Dokumente und dies ist die Aufgabe von XSL. In XSL nimmt ein XSL-Prozessor eine XSL-Regelmenge, die Patterns und Transformationen enthält. Der XSL-Prozessor wendet diese XSL-Regelmenge auf ein XML-Dokument an, indem er die Teile des XML-Dokumentes herausucht, die mit den Patterns übereinstimmen, und Transformationen darauf ausführt, durch die das transformierte XML-Dokument entsteht.

Die Workspace-Engine ist daher durch einen XSL-Prozessor realisiert und die einzelnen Schritte werden durch XSL-Regelmengen dargestellt (Abbildung 4.4.).

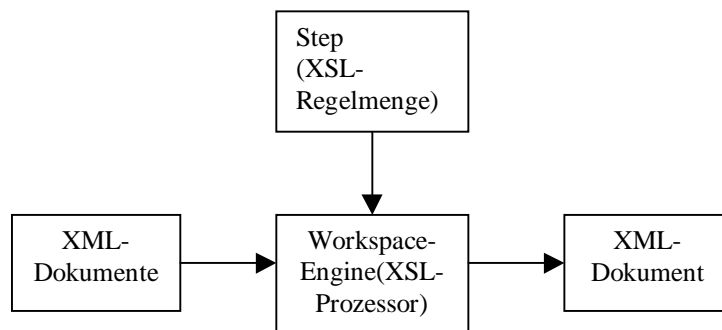


Abbildung 4.4.: Realisierung über XSL (aus [Tolksdorf00a])

4.5. Workspace Coordination Language (WSCL)

Basic-Schritte und Koordinationsschritte bilden zusammen einen Workflow. Die Beschreibung eines kompletten Workflows erfolgt in Workspace über ein XML-Dokument, das der durch die Workspace Coordination Language(WSCL) festgelegten DTD folgt. Die WSCL baut auf der vom WfMC im Papier [WfMC98a] beschriebenen Workflow Process Definition Language (WPDL) auf. Abbildung 4.5. zeigt einen Ausschnitt aus der Beschreibung eines Workflows als WSCL-Dokument und der entsprechenden Darstellung als Workflow-Graph aus Basic- und Koordinationsschritte.

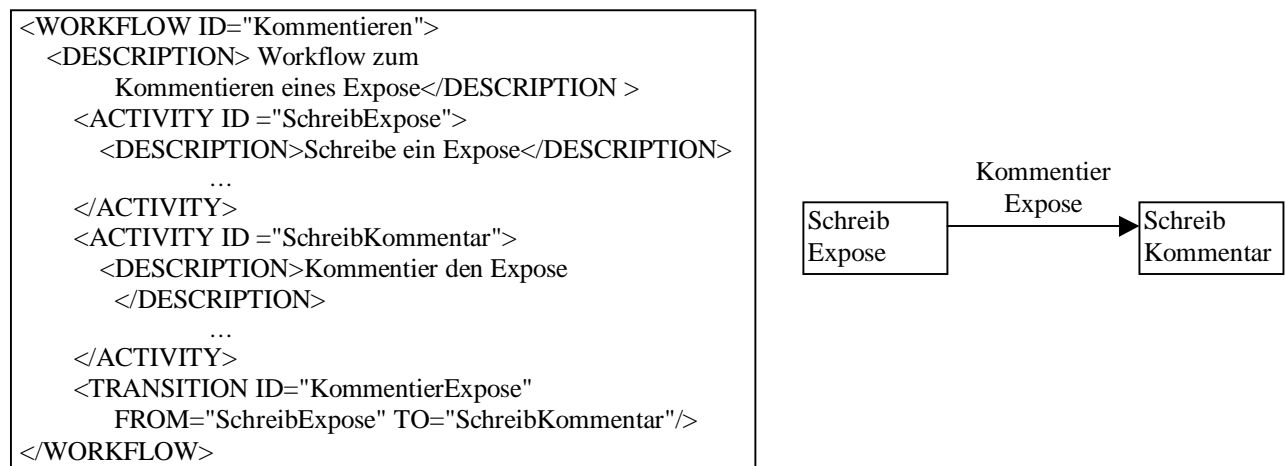


Abbildung 4.5.: Ein WSCL-Dokument und der entsprechende Workflowgraph

Der über das WSCL-Dokument beschriebenen Workflow wird durch sogenannte Meta-Schritte in eine Menge von XSL-Dokumenten zerlegt, die die einzelnen Basic- und Koordinationsschritte des Workflows darstellen, und von den Workspace-Engines ausgeführt werden können.

Die Meta-Schritte selbst sind ebenfalls durch XSL-Regeln beschrieben, weil die Umwandlung des über das WSCL-Dokument beschriebenen Workflows in viele einzelne Basic- und Koordinationsschritte dem Transformieren eines XML-Dokumentes in andere XML-Dokument entspricht und dies die Aufgabe von XSL ist.

Zusammengefasst ergibt sich folgender Ablauf in Workspace: Man beschreibt einen Workflow als XML-Dokument, das der WSCL-DTD folgt. Dieser Workflow wird durch Meta-Schritte in einzelne Basic- und Koordinationsschritte zerlegt, die als XSL-Dokumente dargestellt sind. Diese Basic- und Koordinationsschritte werden von den Workspace-Engines auf die anwendungsspezifischen XML-Dokumente angewendet, die als Eingabe für die verschiedenen Schritte dienen, und damit der Workflow abgearbeitet. Abbildung 4.6. veranschaulicht dies.

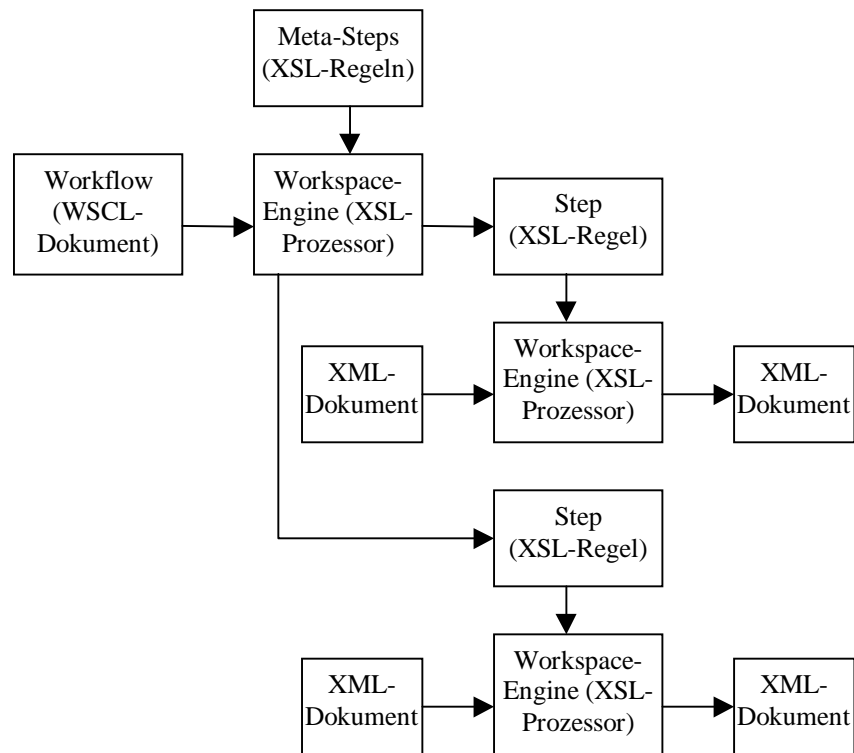


Abbildung 4.6. Gesamtablauf in Workspace (aus [Tolksdorf00a])

4.6. XMLSpaces

Zur Koordination der verschiedenen Workspace-Engines, die einen Workflow ausführen, wird eine Lindaartige Koordinationstechnologie in Form eines gemeinsamen Datenraumes verwendet. In diesem gemeinsamen Datenraum können die von den einzelnen Schritten erzeugten XML-Dokumente abgelegt und daraus wiedergewonnen werden. Da in dem Datenraum nur XML-Dokumente abgelegt werden, wird der Datenraum XMLSpace genannt (Abbildung 4.7.).

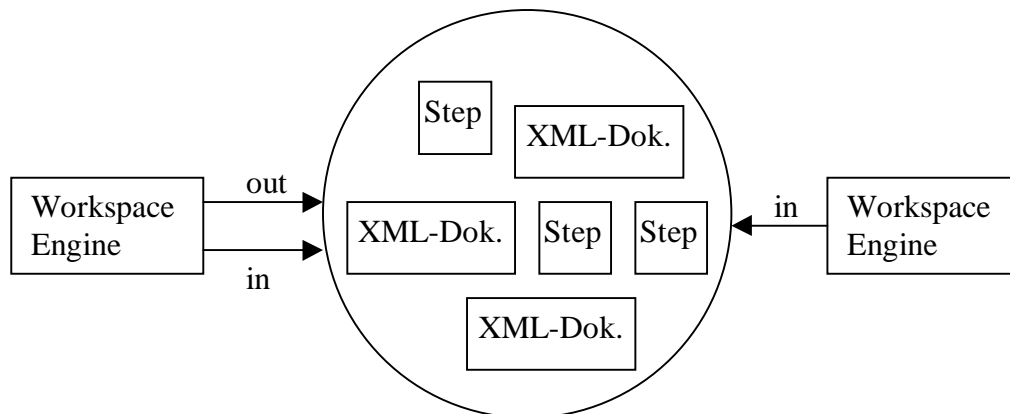


Abbildung 4.7.: Der XMLSpace

Die Verwendung einer Linda-artigen Koordination passt sehr gut in das Workspace-Modell: Ein Schritt muß solange warten, bis die notwendigen Eingabe-Dokumente vorhanden sind, und erzeugt als Ergebnis Dokumente, die von irgendeinem anderen Schritt als Eingabe-Dokumente verwendet werden. Dies ist durch die Verwendung der Lindaoperationen `in()` und `out()` realisierbar.

Damit ergibt sich folgender Ablauf: Eine Workspace-Engine, holt sich über einen `in()`-Aufruf einen Schritt aus dem XMLSpace. Über weitere `in()`-Aufrufe wartet die Engine auf die XML-Dokumente, die zur Ausführung des Schritts erforderlich sind. Nachdem die Engine die notwendigen XML-Dokumente über den `in()`-Aufruf erhalten hat, führt sie den Schritt darauf aus und steckt die durch den Schritt erzeugten Ergebnis-XML-Dokumente über `out()`-Aufrufe ins XML-Space, wo sie von anderen bereits wartenden Schritten über `in()`-Aufrufe wiedergewonnen werden können usw.

Wie in den vorherigen Kapiteln gezeigt, verwendet Workspaces ausschließlich XML-Dokumente:

- Der Workflow wird als ein XML-Dokument beschrieben, das der WSCL-DTD folgt.
- Die einzelnen Schritte (Basic-, Koordinations- und Meta-Schritte) werden als XSL-Regeln dargestellt. XSL-Regeln sind XML-Dokumente, die der XSL-DTD folgen.
- Die Dokumente, die von den einzelnen Schritten als Eingabe genommen und als Ausgabe erzeugt werden, sind XML-Dokumente, die einer anwendungsspezifischen DTD folgen.

Das Konzept von Linda, das auf Tuples mit einfachen Feldwerten aufbaut, muß daher um XML-Dokumente erweitert werden, damit es in Workspace verwendet werden kann. Diese Erweiterung des Tuplespaces um XML-Dokumente wird XMLSpace genannt.

Eine wesentliche Änderung bei der Erweiterung des Linda-Konzeptes um XML-Dokumente ist die Anpassung des Matchingverhaltens an die XML-Dokumente. XML-Dokumente haben eine komplexere Struktur als einfache Datentypen wie Integer und String. Es reicht daher nicht aus einfach das Typ-Wert-Matching von Linda zu verwenden, sondern es sollten Matchingrelationen verwendet werden können, die die Struktur des XML-Dokumentes berücksichtigen.

XMLSpace führt mehrere solcher Matchingrelationen ein, mit denen man die XML-Dokumente auf unterschiedliche Arten wiedergewinnen kann. Zum Beispiel kann eine Matchingrelation als DTD angegeben werden, die das zu matchende XML-Dokument beschreibt. Oder es werden Anfragesprachen wie z.B. XQL verwendet, mit denen man eine Matchingrelation beschreiben kann, die mit einem XML-Dokument matcht, das einen bestimmten Element- oder Attributwert enthält.

Im Rahmen von Workspaces wird für das Wiedergewinnen von XML-Dokumenten aus dem XMLSpace folgende XML-Matchingrelationen benötigt:

- Für das Wiedergewinnen eines auszuführenden Schritt, wird irgendein Schritt-Dokument über seine Step-DTD gematcht.
- Für das Wiedergewinnen eines speziellen Dokumentes innerhalb eines Workflows wird über einen Attributwert im Wurzelement des XML-Dokumentes gematcht.

Durch die Verwendung der Lindaartigen Koordinationstechnologie in Form eines XMLSpace ergeben sich eine Reihe von Vorteile für die Workspace-Architektur. Der XMLSpace entkoppelt die verschiedenen Workspace-Engines in Zeit und Raum. Dadurch können die Workspace-Engines räumlich verteilt sein und es wird Mobilität unterstützt. Die Anzahl der Engines muß zu keiner Zeit festgesetzt sein, sondern kann sich zur Laufzeit ändern. Die Workspace-Engines müssen nicht alle zur gleichen Zeit laufen.

4.7. Stand von Workspaces

Die Realisierung von Workspaces findet im Rahmen von verschiedenen Diplomarbeiten im Fachbereich FLP an der TU Berlin statt. Marc Stauch hat in seiner Diplomarbeit [Stauch] die Workspace-Engine basierend auf XML/XSL-Technologie entwickelt und Amarilis Macedo-Aranya hat in ihrer Diplomarbeit [Aranya] einen grafischen Editor zum Modellieren von Workflows und automatischen Generieren entsprechender WSCL-Dokumente implementiert.

Diese Diplomarbeit beschreibt die Realisierung von XMLSpace, das eine verteilte Linda-artige Koordination über XML-Dokumente ermöglicht. Das nächste Kapitel beschreibt die Anforderungen an XMLSpace und die darauffolgenden Kapitel beschreiben die Realisierung von XMLSpace auf der Basis von TSpace.

Weitere Informationen zu Workspaces gibt es unter www.cs.tu-berlin.de/~tolk/workspaces.

5. XMLSpaces

Der Rest dieser Diplomarbeit beschreibt die Realisierung von XMLSpaces, das in Workspaces zur lindaartigen Koordination mit XML-Dokumenten zwischen den Workspace-Engines verwendet wird.

Dieses Kapitel gibt einen Überblick über die zu realisierenden Eigenschaften von XMLSpaces und die dafür verwendete Software.

5.1. Eigenschaften von XMLSpaces

Zur Zeit unterstützt XMLSpace folgende Eigenschaften:

- **Unterstützung für XML-Dokumente:** XMLSpaces erweitert das Linda-Konzept um die Möglichkeit neben normalen Tuples auch XML-Dokumente im Tuplespace ablegen und daraus wiedergewinnen zu können und unterstützt damit Workspaces, das eine lindaartige Koordination verwendet, jedoch komplett auf XML-Dokumente basiert. XML-Dokumente haben eine wesentlich komplexere Struktur als die normalen Linda-Tuples. Daher benötigt man Matchingrelationen, die diese Struktur berücksichtigen. XMLSpace führt mehrere solcher Matchingrelationen ein, mit denen man die XML-Dokumente auf unterschiedliche Arten wiedergewinnen kann. Zum Beispiel kann eine Matchingrelation als DTD angegeben werden oder durch die Angabe einer XQL-Anfrage. Neben den bereits von XMLSpace bereitgestellten Matchingrelationen bietet XMLSpace die Möglichkeit problemlos neue Matchingrelationen einzubinden. Die Realisierung der XML-Erweiterung von Linda ist im Kapitel 6 beschrieben.
- **Verteiltheit :** XMLSpace unterstützt das Zusammenarbeiten mehrerer räumlich verteilter Tuplespaces, die zusammen ein logisches Tuplespace bilden sollen. Dazu sind die einzelnen XMLSpaces miteinander verbunden und können nach einer vorher festgelegten Verteilungsstrategie auf die Daten des jeweils anderen zugreifen. Die Realisierung der Verteiltheit ist im Kapitel 7 beschrieben.
- **Events:** XMLSpace unterstützt das registrieren von verteilten Events. Dadurch ist es für einen Client möglich automatisch informiert zu werden, wenn ein bestimmtes Ereignis im verteilten Tuplespace eintritt, wie z.B. das Hinzufügen oder Löschen eines Tuples. Auf diese Art können zum Beispiel die Engines in der Workspace-Architektur automatisch informiert werden, wenn neue zu bearbeitende Schritte vorliegen. Die Realisierung der verteilten Events ist im Kapitel 8 beschrieben.
- **Persistenz:** XMLSpace ermöglicht es die in dem Tuplespace gehaltenen Daten persistent auf Dateien zwischenspeichern, so daß die Daten auch über Server-Neustarts hinweg erhalten bleiben. Wie im nächsten Abschnitt 5.2. beschrieben baut die XMLSpace-Implementierung auf TSpace auf. XMLSpace verwendet unverändert die von TSpace bereitgestellt Persistenzunterstützung in Form von Checkpointing. Anhang A beschreibt die notwendigen Einstellungen in der Konfigurationsdatei von TSpace, um Persistenz zu aktivieren.

Neben diesen Eigenschaften gibt es noch weitere wünschenswerte und wichtige Eigenschaften, die jedoch nicht Bestandteil dieser Diplomarbeit sind und daher zur Zeit von XMLSpace noch nicht unterstützt werden. Dazu gehören Security und Fehlertoleranz. Kapitel 10 (Ausblick und offene Punkte) diskutiert diese Eigenschaften kurz an.

5.2. Verwendete Software/Technologie

Dieses Kapitel gibt einen Überblick über die Software und Technologien, die zur Realisierung von XMLSpace verwendet wurden. Anhang B enthält einen tabellarischen Überblick über die verwendete Software zusammen mit den Angaben zu den Herstellern.

5.2.1. Java

Ebenso wie Workspaces, baut XMLSpace auf der Programmiersprache Java auf. Java bietet mit RMI (Remote Method Invocation) ein einfaches Konzept zur Unterstützung von Verteiltheit, indem es ermöglicht auf entfernten Objekten Methoden aufzurufen. Die Realisierung der Verteiltheit in XMLSpaces baut auf RMI auf.

5.2.2. TSpace

TSpace [TSpace] ist eine in Java geschriebene Linda-Implementierung von IBM und wird im Rahmen dieser Diplomarbeit als Implementierungsplattform für die XML-Erweiterung von Linda und für XMLSpace verwendet. TSpace ist einfach zu erweitern (sowohl um neue Typen von Objekte, die im Space abgelegt werden können (wie z.B. XML-Dokumente) als auch um neue Befehle (wie z.B. verteilte Methoden)) und bietet viele Eigenschaften, auf denen die hier geforderten Eigenschaften aufbauen können. So bietet TSpace Unterstützung für Events, Persistenz, Security, Transaktionen. Neben den lindaspezifische Operationen write, waitToRead und waitToTake (entsprechen out, rd und in in Linda) bietet TSpace auch nicht-linda-spezifische Operationen an, wie z.B. readTupleById (das ein Tuple nicht über ein Template, sondern über eine eindeutige Tuple-Id aus dem Tuplespace ausliest), consumingScan (das einem in() entspricht, jedoch im Unterschied zu in() nicht nur ein matchendes, sondern alle matchenden Tuple aus dem Tuplespace löscht und zurückgibt) und ähnliches. Die Ausführungen in der Diplomarbeit konzentrieren sich auf die linda-spezifischen Operationen und diskutieren mögliche Realisierungen für die nicht-lindaspezifischen Operationen höchstens nebenbei an.

5.2.3. XML4J

XML4J ist ein in Java geschrieben validierender XML-Parser von IBM. Mit Hilfe von XML4J werden in XMLSpaces die XML-Dokumente eingelesen und in DOM-Objekte umgewandelt. Zur Zeit verwendet XMLSpaces XML4J Version 2, das nur Unterstützung für DOM1 bietet. Ein wesentlicher Nachteil von DOM1 gegenüber DOM2 ist, daß es keine Unterstützung für Namespaces bietet. Mittlerweile gibt es XML4J Version 3, das auch DOM2 unterstützt. Ein Umstieg von XML4J Version 2 auf Version 3 ist ein noch offener Punkt (siehe auch Kapitel 10).

5.2.4. DOM

Die XML-Dokumente werden in XMLSpace als DOM-Document-Objekte abgelegt. DOM (Document Object Model) [DOM1, DOM2] ist ein W3C-Standard, der standardisierte Interfaces für den Zugriff und das Bearbeiten der Bestandteile eines XML-Dokumentes spezifiziert. In DOM werden XML-Dokumente logisch als Objekt-Bäume repräsentiert. DOM spezifiziert die einzelnen Objekte, aus denen der Dokument-Baum besteht und die Interfaces für den Zugriff auf die Objekte. Der Vorteil von der Verwendung von DOM als Repräsentation für XML-Dokumente ist, daß DOM ein W3C-Standard ist und damit die Wahrscheinlichkeit hoch ist, daß DOM von Matching-Engines wie z.B. Engines für XQL, XPath, XMLQL usw. unterstützt wird und diese sich daher einfach in XMLSpaces einbinden lassen. Ein weiterer Vorteil ist, daß DOM ein sehr einfaches, umfangreiches API für den Zugriff auf die einzelnen Bestandteile eines XML-Dokumentes anbietet und damit die Entwicklung eigener Matching-Engines gut unterstützt. Zur Zeit verwendet XMLSpaces DOM1. Der wesentliche Unterschied und Nachteil von DOM1 zu DOM2 ist, daß DOM1 keine Namespaces unterstützt. Die Unterstützung von Namespaces erfordert einen Umstieg auf DOM2 und ist ein noch offener Punkte (siehe auch Kapitel 10).

5.2.5. Matching-Engines

Für die Realisierung der im Kapitel 6 beschriebenen Matchingrelationen wurde versucht nach Möglichkeit bereits vorhandene Engines einzubinden, die dieses Matchingverhalten realisieren. Dies wurde durch die Verwendung von DOM unterstützt und dadurch, daß versucht wurde, nach Möglichkeit die Matchingrelationen auf Standards basieren zu lassen. Kapitel 6.3.9. gibt einen Überblick über die verschiedenen Matching-Engines, die zur Realisierung der Matchingrelationen verwendet wurden.

6. Erweiterung von Linda um XML-Dokumente

Dieses Kapitel beschreibt die Erweiterung von Linda um XML-Dokumente.

Kapitel 6.1. beginnt mit der Beschreibung, wie Linda konzeptionell um XML-Dokumente erweitert werden kann und zeigt, daß es notwendig ist für XML-Dokumente neue Arten von Matchingrelationen einzuführen.

Einen Überblick über mögliche Matchingrelationen für XML-Dokumente wird in Kapitel 6.2. gegeben.

Das Kapitel 6.3. schließlich beschreibt die Realisierung der Linda-XML-Erweiterung in XMLSpaces auf der Basis von TSpaces.

6.1. Konzeptionelle Erweiterung von Linda um XML-Dokumente

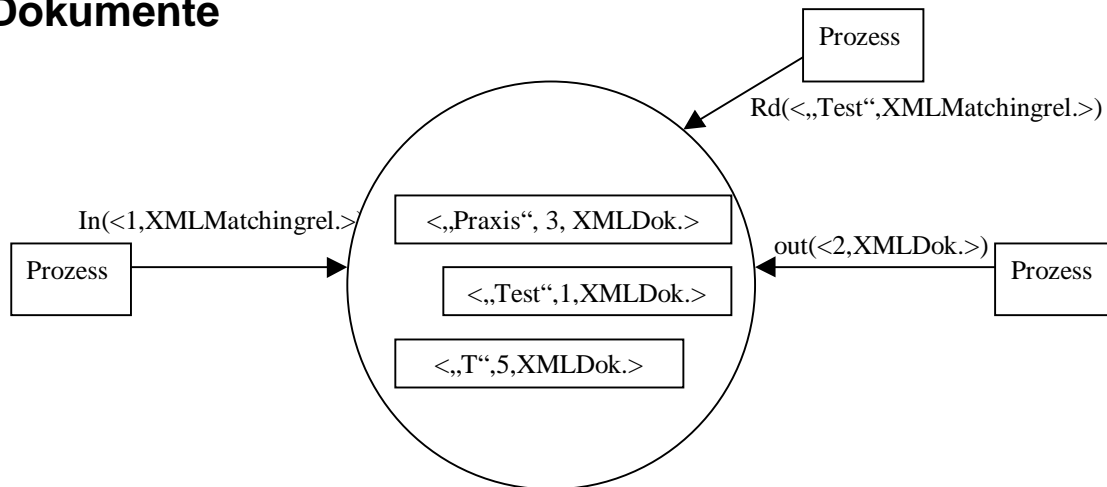


Abbildung 6.1.: Erweiterung eines Tuplespace um XML-Dokumente

Das im Kapitel 4 beschriebene Workspace baut auf XML-Dokumente auf und verwendet Lindatechnologie zur Koordination zwischen den einzelnen Workspace-Engines. Der Tuplespace von Linda verwendet standardmäßig Tuples und Templates mit einfachen Feldwerten vom Typ Integer, String usw. Für die Verwendung in Workspace muß der Tuplespace um die Möglichkeit erweitert werden, auch XML-Dokumente darin ablegen und daraus wiedergewinnen zu können. Der um XML-Dokumente erweiterte Tuplespace wird XMLSpaces genannt.

In der hier beschriebenen Erweiterung werden weiterhin Tuples verwendet, es besteht jedoch die Möglichkeit neben Feldern mit einfachen Datentypen wie Integer und String auch Felder mit XML-Dokumenten zu haben, z.B. `<„Test“, 2, XML-Dokument>`. Es findet hier also eine Erweiterung der einfachen Feld-Datentypen um den komplexeren XML-Dokument-Datentyp statt.

Damit bleibt die out-Operation unverändert: Man erzeugt ein Tuple mit einem XML-Dokument als Feldwert und fügt es über out zum XMLSpace hinzu.

Das Wiedergewinnen von Tuples mit XML-Dokumenten aus dem XMLSpace erfolgt weiterhin über in und rd und die Semantik der in- und rd-Operatoren bleibt durch die XML-Erweiterung unverändert. Jedoch reicht es nicht mehr aus die normale Typ-Wert-Matchingrelation von Linda zu verwenden.

XML-Dokumente haben eine wesentlich komplexere Struktur als die normalen Feldwerte vom Typ Integer, String usw. Man benötigt Matchingrelationen, die diese Struktur berücksichtigen. Es reicht nicht aus in einem Template nur angeben zu können, daß man ein

Tuple mit genau dem gleichen XML-Dokument (Wert-Wert-Matching) oder ein Tuple mit irgendeinem XML-Dokument haben will (Typ-Wert-Matching).

So fordert Workspaces zum Beispiel die Möglichkeiten XML-Dokumente entweder anhand der DTD oder anhand eines bestimmten Elementwertes im XML-Dokument wiedergewinnen zu können. Ersteres entspräche vom Prinzip her dem Typ-Wert-Matching, bei dem die DTD den Typ des Widerzugewinnenden XML-Dokumentes beschreibt. Letzteres (das Wiedergewinnen eines XML-Dokumentes mit einem bestimmten Element-Wert) ist aber weder durch Wert-Wert- noch durch Typ-Wert-Matching realisierbar.

Für die normalen Feldwerttypen wird weiterhin das Wert-Typ-Matching von Linda verwendet. Für XML-Dokumente jedoch wird das Matching um spezielle XML-Matchingrelationen erweitert, die in einem Template angegeben werden können, um das widerzugewinnende XML-Dokument zu beschreiben. Zum Beispiel wählt das Template `<„Hallo“, Integer, XMLMatchingrelation>` ein Tuple mit dem String Hallo, irgendeinem Integerwert und den durch die XMLMatchingrelation beschriebenen XML-Dokument aus.

Das Linda-Matchingverhalten verändert sich damit folgendermaßen: Ein Template matcht mit einem Tuple, wenn

- Das Template und das Tuple die gleiche Anzahl von Feldern haben.
- Für ein Feld mit einem normalen Feldwert im Template ein entsprechendes Feld mit dem gleichen normalen Feldwert im Tuple existiert (= normales Wert-Wert-Matching von Linda).
- Für ein Feld mit einem normalen Feldtyp im Template ein entsprechendes Feld mit einem Wert vom gleichen Typ im Tuple existiert (= normales Typ-Wert-Matching von Linda).
- Für ein Feld mit einer XML-Matchingrelation im Template ein entsprechendes Feld mit einem XML-Dokument im Tuple existiert, das die XML-Matchingrelation erfüllt.

Das nächste Kapitel beschreibt mögliche Matchingrelationen für XML-Dokumente.

6.2. Matchingrelationen

Dieses Kapitel gibt einen Überblick über mögliche Matchingrelationen für XML-Dokumente: Kapitel 6.2.1. beschreibt Anfragesprachen (XPath, XPointer, XQL, XMLQL), Kapitel 6.2.2. beschreibt das DTD-Matching (das dem Linda-Typ-Wert-Matching bezogen auf XML-Dokumente entspricht) und Kapitel 6.2.3. beschreibt das Matching auf Gleichheit (das dem Linda-Wert-Wert-Matching entspricht).

Die einzelnen Matchingrelationen können über Boolesche Operatoren verknüpft werden und dadurch beliebig komplexe Matchingrelationen bilden. Dies wird in Kapitel 6.2.6. beschrieben.

Da Namespaces die Wiederverwendung von DTDs unterstützen und dazu beitragen Matchingrelationen eindeutiger zu machen, wird in Kapitel 6.2.7. diskutiert, in wie weit die hier beschriebenen Matchingrelationen die Verwendung von Namespaces unterstützen.

Kapitel 6.2.8. enthält schließlich eine Zusammenfassung und einen Vergleich der Matchingrelationen.

6.2.1. Anfragesprachen

Eine Anfragesprache dient im Bereich von relationalen und objektorientierten Datenbanken zum Suchen und manipulieren (Erzeugen, Ändern, Löschen) von Daten. Die bekannteste Anfragesprache aus dem Bereich relationaler Datenbanken ist SQL[SQL]. SQL bietet unter anderem Konstrukte zum Erzeugen von Daten (`create Table`), zum Einfügen von Daten (`insert into`) und zum Suchen von Daten (`select-from-where`).

Da es sich beim Matching in XMLSpace um das Auffinden von Daten handelt ist nur der aufs Suchen bezogene Teil von Anfragesprachen für Matchingrelationen relevant.

SQL verwendet das SELECT-FROM-WHERE-Konstrukt zum Suchen von Daten. SELECT beschreibt in welcher Form das Ergebnis zurückgegeben werden soll, FROM beschreibt aus welcher Tabelle die Daten gesucht werden sollen und WHERE beschreibt, welche Daten wiedergefunden werden sollen. Im Prinzip ist eigentlich nur der WHERE-Abschnitt für das Matching relevant. FROM bezieht sich immer auf das zu matchende XML-Dokumente, in dem überprüft werden soll, ob es die über WHERE beschriebenen Daten enthält, und SELECT sollte immer sämtliche Ergebnisse zurückgeben, die gefunden wurden.

SQL ist für die Verwendung in relationalen Datenbanken zugeschnitten und lässt sich nicht einfach als Anfragesprache für XML-Dokumente verwenden, weil XML-Dokumente eine andere Struktur als relationale Datenbanken haben. Relationale Datenbanken bestehen aus Tabellen und XML-Dokumente haben eine Baumstruktur bestehend aus Elementen, Attributen, Text, Kommentare und PIs. Man spricht im Zusammenhang von XML-Dokumenten auch von semistrukturierten Daten, weil sie nicht so starr und regelmäßig strukturiert sind wie relationalen Datenbanken.

Das Suchen in XML-Dokumenten kann zum Beispiel über den Namen der Elemente, den Attributwerten oder den Texten erfolgen. Wichtig ist es jedoch auch die strukturellen und hierarchischen Beziehungen zwischen den einzelnen Bestandteilen des XML-Dokumentes mit in die Suche einbeziehen zu können. Beispiele für die strukturellen Beziehungen zwischen den Bestandteilen eines XML-Dokumentes sind: Vater-Kind-Beziehungen, Vorfahr-, Nachfahr-, Geschwister-Beziehungen usw. (z.B. „Suche sämtliche Elemente, die Kind-Elemente dieses Elementes sind...“)

Das W3C ist zur Zeit dabei eine Standard-Anfragesprache für XML-Dokumente zu entwickeln [W3C00a, W3C00b]. Zur Zeit existieren nur Anforderungen und ein Datenmodell.

Die nächsten Kapitel beschreiben XPath, XPointer, XQL und XML-QL als Anfragesprachen. XPath und XPointer sind keine echten Anfragesprachen, können jedoch zum Suchen von Informationen in einem XML-Dokument verwendet werden und werden daher in diesem Zusammenhang beschrieben. XPath, XPointer und XQL dienen nur zum Suchen von Daten in einem XML-Dokument. XML-QL dagegen hat den Anspruch eine komplette Anfragesprache zu sein, die neben dem Suchen von Daten auch das Manipulieren und Erzeugen von XML-Dokumenten unterstützt.

XPath, XPointer und XQL basieren auf Path-Expressions (Pfadausdrücke). Path-Expressions stammen aus dem Bereich von semistrukturierten Datenbanken [Abiteboul97] und beschreiben die zu suchenden Daten über Pfade durch den Baum.

XML-QL verwendet zum Beschreiben der Suche in XML-Dokumenten ein SELECT-FROM-WHERE ähnliches WHERE-IN-CONSTRUCT-Konstrukt.

Die hier beschriebenen Anfragesprachen lassen sich als Matchingrelation verwenden, indem man mit ihrer Hilfe eine Anfrage formuliert, die beschreibt, welche Daten das zu matchende XML-Dokument enthalten soll. XML-Dokumente, die diese Anfrage erfüllen, erfüllen auch die Matchingrelation.

6.2.1.1. XPath

XPath [XPath] ist eine Sprache, um auf Stellen innerhalb eines XML-Dokumentes zu verweisen.

XPath operiert auf einer DOM-ähnlichen Baum-Struktur aus Knoten (siehe auch die Beschreibung zu DOM in Kapitel 6.3.4.), in denen die Knoten die Elemente, Attribute, Textabschnitte, Processing-Instructions und Kommentare eines XML-Dokumentes repräsentieren.

Das wichtigste Konstrukt in XPath ist der Location-Path (Lokalisierungs-Pfad), der angibt, welche Knoten aus einem XML-Dokument ausgewählt werden sollen, indem er einen Pfad durch die Baumstruktur zu den auszuwählenden Knoten beschreibt.

Ein Beispiel für ein Location-Path ist:

```
/child::chapter[attribut::caption="XPath"]/child::section[attribut::caption="LocationPath"]
```

Dieser Location Path sucht innerhalb der Chapter-Elemente mit dem Caption-Attribut „XPath“ nach Section-Elemente mit dem Caption-Attribut „LocationPath“, die in einer Vater-Kind-Beziehung zu den Chapter-Elementen stehen.

Ein Location-Path besteht aus mehreren Location-Steps, die über ein „/“ miteinander verbunden werden, z.B.: /LocationStep1/LocationStep2/LocationStep3. Das führende „/“ kennzeichnet die Wurzel des XML-Dokumentes.

Die einzelnen Location-Steps werden von links nach rechts ausgeführt. Jeder Location-Step wählt eine Menge von Knoten aus, die den Kontext für den nächsten Location-Step bilden. Zum Beispiel wählt der LocationStep1 mit dem führenden alleinstehenden „/“ Knoten relativ zur Wurzel des XML-Dokumentes aus. Der auf den Location-Step1 folgende Location-Step2 wählt Knoten relativ zu den im Location-Step1 bestimmten Knoten aus usw.

Jeder Location-Step wählt also eine Menge von Knoten relativ zu den durch den vorherigen Location-Step ausgewählten Knoten aus. Die durch den vorherigen Location-Step ausgewählten Knoten werden Kontext-Knoten genannt, da sie den Kontext für den aktuellen Location-Step bilden. Die Anzahl der durch einen vorherigen Location-Step bestimmten Kontext-Knoten wird Kontextgröße genannt und die Position eines Kontext-Knoten innerhalb der Menge der Kontext-Knoten wird Kontextposition genannt. Abbildung 6.2.

veranschaulicht dies. Darin wird durch jeden Locationstep die Kinder-Elemente des vorherigen Location-Steps ausgewählt. Der Locationstep1 wählt sämtliche Kinder-Knoten von dem Wurzel-Element aus. Die in dem Locationstep1 ausgewählten Kinder-Knoten sind in der Abbildung durch eine einrahmende gestrichelte Linie hervorgehoben. Diese Knoten bilden die Kontext-Knoten für den Locationstep 2, d.h. der Locationstep 2 wird für jeden einzelnen Kontext-Knoten in der Menge der Kontext-Knoten ausgeführt und die Vereinigung der Knoten, die sich aus der Anwendung von Locationstep 2 auf jeden einzelnen Kontext-Knoten ergeben, ergibt das Gesamtergebnis vom Locationstep 2. Dieses Gesamtergebnis wäre dann wiederum die Kontextmenge für den (hier nicht vorhandenen) Locationstep 3. Das dritte Element in der Menge der Kontext-Knoten hat die Kontextposition 3, weil es an dritter Stelle in der Menge der Kontext-Knoten steht, und die Menge der Kontext-Knoten hat eine Kontextgröße von 4, weil die Menge 4 Kontext-Knoten enthält.

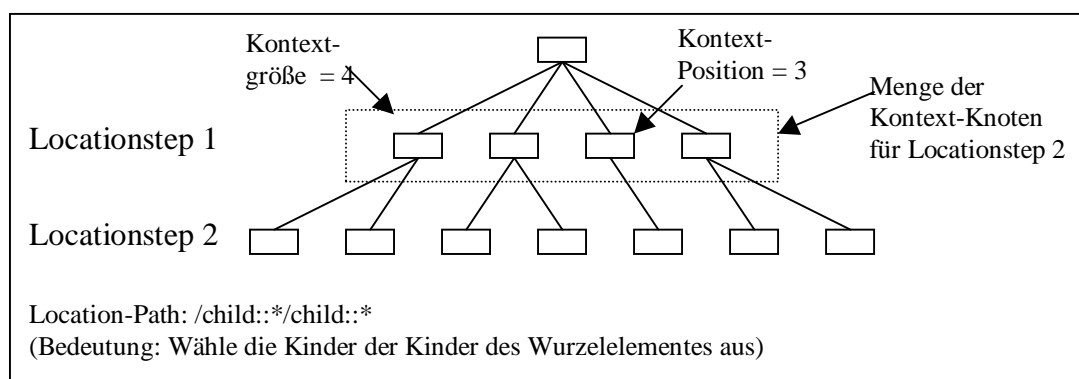


Abbildung 6.2.: Kontextknoten, -größe und -position

Ein einzelner Location-Step hat die Form: Axis::Knotentest[Prädikat1][Prädikat2]...

Die Axis beschreibt die Beziehung zwischen den vom Location-Step auszuwählenden Knoten und dem Kontext-Knoten. Tabelle 6.3. enthält eine Übersicht über die verschiedenen möglichen Arten von Axis. Der Location-Path „LocationStep1/child“ wählt z.B. alle Kinder-Knoten von den im LocationStep1 bestimmten Knoten aus.

<ul style="list-style-type: none"> • ancestor: Vorfahren des Kontext-Knoten • ancestor-or-self: Vorfahren und der Kontext-Knoten selbst • attribute: Attribute eines Kontext-Knoten • child: Kinder des Kontext-Knoten • descendant: Nachfahren des Kontext-Knoten • descendant-or-self: Nachfahren und der Kontext-Knoten selbst • following-sibling: Alle nachfolgenden Geschwister des Kontext-Knoten • parent: Der Vater des Kontext-Knoten • preceding-sibling: alle vorhergehenden Geschwister des Kontext-Knoten • self: der Kontext-Knoten selbst 	
---	--

Tab. 6.3.: Übersicht über mögliche Axis

Abbildung 6.4: Veranschaulichung von einigen Axis

Über den Knotentest werden die durch die Axis ausgewählten Knoten weiter eingeschränkt. Der Knotentest spezifiziert den Knoten-Typ und die Namen der Knoten, den die Knoten der Ergebnismenge des Location-Steps haben sollen.

Tabelle 6.5. enthält eine Übersicht über die verschiedenen Arten von Knotentests. Zum Beispiel wählt `child::para` von den Kinder-Knoten des Kontext-Knotens nur die `para`-Elemente aus. `child::*` wählt dagegen alle Kinder-Knoten vom Kontext-Knoten aus.

Knotentest	Funktionalität
Knoten-Name	wählt nur Knoten aus, die den angegebenen Namen haben.
*	wählt sämtliche Knoten aus unabhängig vom Namen
text()	wählt alle Text-Knoten aus
comment()	wählt alle Kommentar-Knoten aus
processing-instruction()	wählt alle processing-instruction-Knoten aus
node()	wählt sämtliche Knoten aus

Tabelle 6.5.: Übersicht über Knotentests

Mit Hilfe der Prädikate können die von einem Location-Step auszuwählenden Knoten noch weiter eingeschränkt werden. Das Prädikat filtert aus der durch `Axis::Knotentest` bestimmten Menge von Knoten die Knoten, die den im Prädikat angegebenen Ausdruck erfüllen. Zum Beispiel wählt `child::para[attribute::type='warning']` alle `para`-Kinder-Elemente von dem Kontext-Knoten aus, die ein `type`-Attribut mit dem Wert „warning“ haben.

Es können beliebig viele Prädikate hintereinander angegeben werden und dadurch die Ergebnismenge eines Location-Steps beliebig eingeschränkt werden. Die Prädikate werden alle miteinander ver-„und“-et, d.h. ein Knoten muß sämtliche angegebenen Prädikate erfüllen,

um in der Ergebnismenge aufgenommen zu werden. Zum Beispiel wählt `child::para[attribute::type='warning'][attribute::betriebssystem='Unix']` alle para-Kinder-Elemente von dem Kontext-Knoten aus, die ein type-Attribut mit dem Wert "warning" UND ein betriebssystem-Attribut mit dem Wert „Unix“ haben.

Die in einem Prädikat angegebenen Ausdrücke können Operatoren wie `and`, `or`, `not` für das Verknüpfen von Boolwerten (z.B. `/child::*[attribute::os="Unix" or attribute::os="Windows"]`) und Operatoren wie `+` und `-` für das Berechnen von Zahlenwerten verwenden.

XPath bietet außerdem eine Funktionsbibliothek mit vordefinierten Funktionen, die in den Prädikaten verwendet werden können. Die Tabelle 6.6. enthält einige Beispiele für vordefinierte Funktionen. Zum Beispiel kann über die Funktion `position()` die Kontext-Position eines Kontext-Knotens ermittelt werden: der Location-Step `child::chapter[position()=5]` liefert das fünfte Chapter-Kind-Element (child::chapter liefert alle chapter-Kind-Elemente und `[position()=5]` wählt das fünfte davon aus). Über die Funktion `id()` kann ein Element über seine einzigartige Id ausgewählt werden: `child::chapter[id("foo")]` wählt das Chapter-Kind-Element mit dem Id-Attributwert „foo“.

Funktion	Zweck	Verwendungsmöglichkeit der Funktion für Matchingrelation in XMLSpaces
<code>position()</code>	Bestimmt die Position eines Kontext-Knotens.	Zur Angabe von Bedingungen, die sich auf die Kontextposition beziehen. Zum Beispiel wählt <code>/descendant::chapter[position()=4 and attribute::name="Workspaces"]</code> das XML-Dokument aus, das als 4.Kapitel-Element den Namen "Workspaces" hat.
<code>last()</code>	Gibt die Kontextgröße an.	Zur Auswahl von XML-Dokumenten, die eine bestimmte Anzahl von Elementen haben sollen. Zum Beispiel: <code>/descendant::chapter[last()=5]</code> wählt XML-Dokumente aus, die 5 Chapter-Elemente beinhalten.
<code>id(object)</code>	Wählt aus einer Menge von Knoten diejenigen aus, die die über „object“ angegebene Identität haben.	Zur Auswahl von XML-Dokumenten mit Elementen einer bestimmten id. Zum Beispiel wählt <code>/descendant::chapter[id(„foo“)]</code> ein XML-Dokument mit einem Chapter-Element aus, das das Id-Attribut mit dem Wert „foo“ hat.

Tabelle 6.6.: In XPath definierte Funktionen

Zusammengefasst wählt ein Location-Step sämtliche Knoten aus, die in der über die Axis angegebenen Beziehung zu den Kontext-Knoten stehen, den über den Knotentest angegebenen Knoten-Typ und Namen haben und außerdem die angegebenen Prädikate erfüllen.

XPath kann als Matchingrelation für XMLSpaces verwendet werden, indem über einen Location-Path angegeben wird, welche Knoten ein XML-Dokument enthalten soll. Der Location-Path gibt als Ergebnis eine Menge von Knoten zurück, die den durch den Location-Path adressierten Knoten entspricht. Wenn diese Menge leer ist, ist das Matching fehlgeschlagen, ansonsten erfolgreich.

Das folgende ist ein etwas längeres, komplettes Beispiel für einen Location-Path und zeigt die Anwendungsmöglichkeit als Matchingrelation:

```
/descendant::chapter[position()=5]/descendant::section[position()=2][attribute::betriebssystem='Unix']
```

Über diesen Location-Path kann im XMLSpace ein XML-Dokument herausgesucht werden, das im zweiten Section-Element des fünften Chapter-Elementes ein Attribut `betriebssystem` mit dem Wert „Unix“ hat.

Ein LocationPath beschreibt immer nur *einen* Pfad durch den Baum. Mehrere LocationPaths können über den Vereinigungsoperator „|“ vereinigt werden, d.h. die Menge der Ergebnisknoten des einen Location-Paths werden mit der Menge der Ergebnisknoten des anderen Location-Paths zusammengefasst. Auf diese Art kann eine Ver-Oderung von Location-Paths erreicht werden. Zum Beispiel sucht folgende Matchingrelation nach XML-Dokumenten, die entweder als direktes Kind des Wurzelementes oder als Kind des Kindes des Wurzelementes ein Attribut version mit dem Wert „2.0“ hat:

```
(/child::*[attribute::version="2.0"]) | (/child::*/*[attribute::version="2.0"])
```

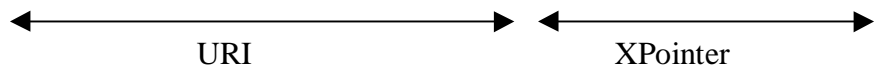
Dieser Ausdruck lässt sich nur durch die Verknüpfung mehrere Location-Paths beschreiben, weil es sich um unterschiedliche Pfade durch den Baum handelt.

Eine Ver-Undung von Location-Paths ist nicht möglich (jedoch wird dies durch die im Kapitel 6.2.6. beschriebenen Booleschen Verknüpfungen von Matchingrelationen möglich).

Zusammengefasst eignet sich XPath für Matchingrelationen, die sich als Zugriffe auf die Eigenschaften einzelner Bestandteile und damit als Pfade durch die Baumstruktur eines XML-Dokumentes beschreiben lassen.

6.2.1.2. XPointer

XPointer [XPointer] dient ebenso wie XPath zum Adressieren auf Stellen innerhalb eines XML-Dokumentes und wird im Zusammenhang mit URI-Referenzen als Fragment-Identifizier verwendet. Zum Beispiel verweist in

[www.cs.tu-berlin.de/~glaubitz/Test.xml#xpointer\(id\(„workspaces“\)\)](http://www.cs.tu-berlin.de/~glaubitz/Test.xml#xpointer(id(„workspaces“)))


die URI auf das XML-Dokument Test.xml und der XPointer verweist auf ein Element im XML-Dokument mit der ID „Workspaces“. Im Gegensatz zum HTML-href-Fragment-Identifizier erlaubt XPointer auf Teile eines XML-Dokumentes zu verweisen, die nicht explizit mit einem Anchor (ID-Attribut) markiert sind.

XPointer basiert auf XPath. Wie auch in XPath werden Location-Paths verwendet, um Teile des XML-Dokumentes auszuwählen. XPointer führt jedoch einige Erweiterungen ein, die weiter unten in diesem Kapitel kurz beschrieben werden, jedoch für das Matching an sich keine Vorteile bieten.

Ein XPointer hat die allgemeine Form „xpointer(XPtrExpr)“. Ein XPtrExpr entspricht den in XPath definierten Location-Path, jedoch mit XPointer-spezifischen Erweiterungen. Ein Beispiel für einen XPointer ist:

```
xpointer(/child::para[attribute::type='warning'])
```

Dieser XPointer wählt vom Wurzel-Element die para-Kinder-Elemente mit dem Attribut type mit dem Wert „warning“ aus.

XPointer bietet neben der allgemeinen Form zwei Abkürzungen in Form von bloßen Namen und Child-Sequenzen.

Ein bloßer Name ist die Abkürzung für die Verwendung des Namens als Argument in der id()-Funktion. Zum Beispiel ist „intro“ die Abkürzung für „xpointer(id(„intro“))“.

Eine Child-Sequenz lokalisiert ein Element im XML-Dokument, indem es eine Sequenz von durch „/“ getrennte Integers erhält. Jeder Integer lokalisiert das n-te Kind-Element des zuvor lokalisierten Elementes. Zum Beispiel wählt „/1/3/5“ das fünfte Kind-Element des dritten Kind-Elementes des ersten Kind-Elementes der Wurzel.

Die wesentliche Erweiterung von XPointer gegenüber XPath besteht in der Erweiterung um Punkte und Bereiche.

XPath kann nur auf *ganze* Knoten innerhalb eines XML-Dokumentes verweisen, wobei ein Knoten ein Element, Attribut, Text, Kommentar oder ProcessingInstruction(PI) ist. Ein Punkt kann dagegen auf einen Knoten oder auf irgendeine beliebige Stelle *innerhalb* eines Textes, Kommentars oder PIs verweisen. Ein Bereich ist definiert über einen Start- und einen End-Punkt und umfasst den zwischen den Start- und Endpunkt liegenden Bereich des XML-Dokumentes.

XPointer führt dementsprechend neue Punkt- und Bereichs-Funktionen ein. Zum Beispiel kann über die Funktion range-to(expression) ein Bereich festgelegt werden. Der Startpunkt von dem Bereich ist der Kontext-Punkt, auf den diese Funktion angewendet wird und der Endpunkt ist der Punkt, der sich durch die Auswertung der expression auf den Kontext-Punkt ergibt. Zum Beispiel lokalisiert der folgende XPointer den Bereich vom Element mit der Id "chap2" bis zum Element mit der Id „chap4“: xpointer(id("chap2")/range-to(id("chap4")))

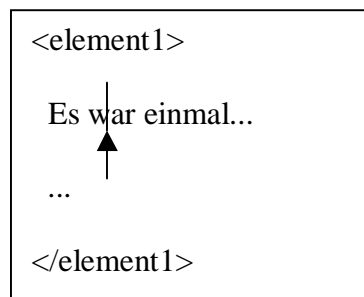


Abbildung 6.7.: Beispiel für einen Punkt, der hinter ‚w‘ und vor ‚a‘ verweist.

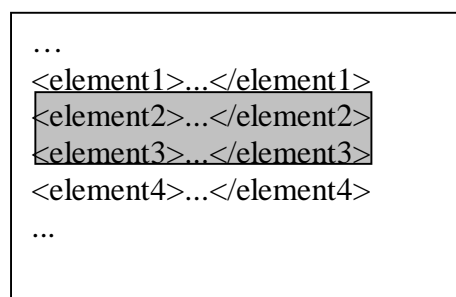


Abbildung 6.8.: Beispiel für einen Bereich, der sich von <element2> bis <element3> erstreckt.

Bezogen auf die Verwendung von XPointer als Matchingrelation für XMLSpaces, bietet XPointer gegenüber XPath keine wesentlichen Unterschiede. Der wesentliche Teil zum Adressieren auf Bestandteile eines XML-Dokumentes sind weiterhin die von XPath definierten Location-Paths. Punkte und Bereiche sind für das Adressieren innerhalb eines XML-Dokumentes sicherlich sinnvoll (da z.B. nicht nur auf den Beginn eines Text-Knotens, sondern auf einen Punkt im Text-Knoten gesprungen werden kann oder ganze Bereiche ausgewählt werden können), aber bieten fürs Matching keine Vorteile.

6.2.1.3. XQL

XQL [XQL] ist eine einfache Anfragesprache für XML-Dokumente und dient zum Wiedergewinnen von Informationen aus XML-Dokumenten. XQL baut ebenso wie XPath und XPointer auf Path-Expressions auf und ist eigentlich eine Teilmenge von XPath.

XPath bietet eine Reihe von Abkürzungsmöglichkeiten für das Ausdrücken von Location-Paths und auf diesen Abkürzungen baut XQL auf. Die Tabelle 6.9. gibt einen Überblick über die Abkürzungsmöglichkeiten in XPath.

XPath-Ausdruck	Abkürzung
/child::elementname	/elementname
/descendant::elementname	//elementname
attribute::attributname	@attributname
[position()=1]	[1]

Tabelle 6.9.: Abkürzungen in XPath

Ein XPath-Location-Path der Form

```
/child::chapter[position()=2]/descendant::section[attribute::caption="informatik"]
```

wird in XQL dargestellt als

```
/chapter[2]//section[@caption="informatik"]
```

und entspricht damit der abgekürzten Schreibweise von XPath.

XQL bietet keine Axis an, sondern beschreibt die hierarchischen Beziehungen zwischen den Elementen nur über „/“ und „//“. Es gibt in XQL also zum Beispiel keine Möglichkeit für den Zugriff auf den following-sibling(nachfolgende Geschwister) oder den ancestor(Vorfahr). Die XQL-Path-Expressions gehen immer von oben nach unten (top-down).

XQL entspricht vom Prinzip her also XPath, verwendet aber ausschließlich die einfachere Schreibweise von XPath und ist durch das Wegfallen der Axis etwas eingeschränkter. XQL bietet daher genauso wie XPointer gegenüber XPath keine wesentlichen Unterschiede als Verwendung als Matchingrelation für XMLSpace.

6.2.1.4. XML-QL

XML-QL [XMLQL] ist wie XQL eine Anfragesprache, die jedoch nicht nur das Extrahieren von Daten ermöglicht, sondern auch das Konstruieren von XML-Dokumenten, das Transformieren von XML-Dokumente in andere XML-Dokumente und das Integrieren von XML-Daten aus verschiedenen XML-Dokumenten unterstützt.

XMLQL hat ein SELECT-FROM-WHERE ähnliches Konstrukt wie SQL: „WHERE... IN... CONSTRUCT“.

IN verweist auf das XML-Dokument, aus dem die Daten extrahiert werden sollen, WHERE gibt an, welche Daten extrahiert werden sollen und CONSTRUCT beschreibt das zu konstruierende Ergebnis.

Zum Beschreiben der zu extrahierenden Daten verwendet XML-QL in dem WHERE-Abschnitt Element-Patterns. Zum Beispiel wählt die folgende Anfrage sämtliche Autoren aus dem bib.xml-Dokument aus, die nach 1998 ein Buch im Verlag Addison-Wesley veröffentlicht haben. Jahr ist hierbei ein Attribut vom Element Buch. Mit \$ werden Variablen gekennzeichnet, die beim Auswerten der Anfrage an die entsprechenden gefundenen Werte gebunden werden:

```
WHERE
  <buch jahr =$j>
    <verlag><name>Addison-Wesley</name></verlag>
    <autor> $a</autor>
  </buch> IN "www.a.b.c/bib.xml",
  $j > 1998
CONSTRUCT $a
```

XML-QL unterstützt Joins durch die Verwendung gemeinsamer Variablen. Die folgende Anfrage verwendet einen Join zur Auswahl der Artikel, deren Autor auch ein Buch geschrieben hat (der Join erfolgt über die gemeinsame Variable \$a):

```
WHERE
  <artikel>
    <autor>$a</autor>
    <titel> $t </titel>
```

```

</artikel > IN "www.a.b.c/bib.xml"

<buch>
  <autor> $a </autor >
</buch> IN "www.a.b.c/bib.xml",
CONSTRUCT
<artikel>
  <autor> $a </autor>
  </titel> $t </titel>
</artikel>

```

Für Anfragen, in denen der Elementtyp nicht genau bekannt ist oder es mehrere Möglichkeiten gibt, bietet XML-QL die Möglichkeit Tag-Variablen zu verwenden. Zum Beispiel sucht die folgende Anfrage nach allen Büchern, bei denen Smith Autor oder Editor ist (die Tagvariable ist hierbei \$e):

```

WHERE
  <buch >
    <titel> $t </titel>
    <$e> Smith </>
  </buch> IN "www.a.b.c/bib.xml",
  $e IN {autor, editor},
CONSTRUCT
  <buch>
    <titel> $t </titel>
    <$e> Smith </>
  </buch>

```

XML-QL bietet auch die Möglichkeit der Verwendung von Path-Expressions um einen Pfad durch das XML-Dokument zu beschreiben. Ein Path-Expression besteht in XML-QL aus einer Folge von Elementnamen und hat folgende Operatoren zum Verknüpfen der Elementnamen:

- element1|element2: ein Pfad, der entweder aus element1 oder element2 besteht (Alternation)
- element1.element2 : ein Pfad, in dem ein element1 ein element2 enthält (Concatenation)
- element1*: bedeutet null oder mehrere Elemente ineinander verschachtelt (eine Hierarchie von null oder mehr element1-Elementen)
- element1+ bedeutet ein oder mehr Elemente ineinander verschachtelt (eine Hierarchie von ein oder mehr element1-Elementen)

Zum Beispiel sucht die folgende Anfrage den Editor von dem Abschnitt mit dem Titel Workspace heraus, der beliebig tief in anderen Abschnitten des Kapitels geschachtelt sein kann:

```

WHERE
  <kapitel.abschnitt+>
    <titel>Workspaces</titel>
    <editor>$e</editor>
  </>
CONSTRUCT $e

```

Diese Anfrage könnte zum Beispiel auf folgendes XML-Dokument angewendet werden:

```
<kapitel>
  <abschnitt>
    < abschnitt >
      < abschnitt >
        <titel>Workspaces</titel>
        <editor> ... </editor>
      </abschnitt >
    </abschnitt>
  </abschnitt>
</abschnitt>
```

Der Wesentliche Unterschied von XML-QL zu XQL bzw. XPath ist zum einen, daß XML-QL nicht ausschließlich auf Path-Expressions basiert, sondern ein WHERE-IN-CONSTRUCT verwendet und daß XML-QL nicht nur das Extrahieren von Daten, sondern auch das Erzeugen von XML-Dokumenten über die CONSTRUCT-Anweisung unterstützt. Letzteres ist jedoch für XMLSpaces irrelevant, weil dies keinen Vorteil für das Matchingverhalten bringt.

Im Unterschied zu XQL oder XPath unterstützt XML-QL nicht den Zugriff auf Kommentare und ProcessingInstructions, sondern betrachtet nur Elemente und Attribute als relevante Bestandteile eines XML-Dokumentes. XML-QL scheidet damit für Matchingrelationen, die sich auf Kommentare oder ProcessingInstructions beziehen, aus.

XML-QL bietet jedoch im Gegensatz zu XQL durch die Verwendung von Tag-Variablen eine bessere Möglichkeit die zu matchenden Elementnamen variabel zu halten. XQL bietet für diesen Zweck nur die *-Wildcards im Knotentest.

6.2.2. DTD-Matching

Das DTD-Matching überprüft, ob ein XML-Dokument einer angegebenen DTD folgt. Dies entspricht dem Typmatching von Linda: Ein formales Feld in einem Linda-Template matcht mit einem Feldwert in einem Tuple, wenn der Feldwert im Tuple denselben Typ wie das formale Feld im Template hat, z.B. matcht das Template <?int> mit dem Tuple <1>. In XML wird der Typ eines XML-Dokumentes über die DTD beschrieben.

Für das Realisieren des DTD-Matchings gibt es zwei Möglichkeiten: Entweder übergibt man die komplette DTD und überprüft, ob ein XML-Dokument zu der DTD konform ist (im Folgenden DTD-Matching genannt), oder man verwendet die im XML-Dokument angegebene Doctype-Deklaration (im Folgenden Doctype-Matching genannt).

Ersteres lässt sich problemlos über einen validierenden Parser realisieren: Man übergibt eine DTD in Stringform und lässt den Parser überprüfen, ob ein XML-Dokument zu der DTD konform ist.

Das komplette Validieren durch einen Parser ist zeitaufwendig und es kann Situationen geben, in denen man über die DTD matchen will, ohne die DTD zur Verfügung zu haben. Eine Alternative für das DTD-Matching ist einfach die Doctype-Deklaration eines XML-Dokumentes zu überprüfen. Die Doctype-Deklaration ist der Verweis auf die DTD, zu dem ein XML-Dokument konform ist, und wird von einem Parser verwendet, um Herauszufinden, wo er die DTD zum Validieren finden kann.

Die Doctype-Deklaration hat die Form:

```
<!DOCTYPE Name (SYSTEM URI | PUBLIC PublicLiteral)?>
```

Die Doctype-Deklaration enthält also einen Namen und optional einen System- oder einen Public-Identifizier.

Der Name ist *kein* eindeutiger Bezeichner¹ für die DTD, sondern hat lediglich die Bedingung, daß er gleich dem Wurzelement des XML-Dokumentes ist. Da in einer DTD in XML nicht ausgezeichnet ist, welche Element-Deklaration dem Wurzelement entspricht, kann im Prinzip jede Element-Deklaration als Wurzelement dienen. So sind zum Beispiel die in Abbildung 6.10. und Abbildung 6.11. angegebenen XML-Dokumente zur in Abbildung 6.12. angegebenen DTD konform, obwohl sie unterschiedliche Wurzelemente haben. Der im Doctype angegebene Name eignet sich also nicht unbedingt, um eindeutig herauszufinden, ob ein XML-Dokument zu einer DTD konform ist. Trotzdem wird es viele Fälle geben, in denen der Doctype-Name ausreicht, weil sämtliche Dokumente einer gegebenen DTD dasselbe Wurzelement verwenden (weil es zum Beispiel von der Software, die das Dokument verarbeitet so verlangt wird).

```
<!DOCTYPE element1
SYSTEM "Test.dtd">

<element1>
  <element2>
    Test
  </element2>
</element1>
```

Abbildung 6.10.: XML-Dokument mit element1 als Wurzelement

```
<!DOCTYPE element2
SYSTEM "Test.dtd">

<element2>
  Test
</element2>
```

Abbildung 6.11.: XML-Dokument mit element 2 als Wurzelement

```
<!-- Test.dtd -->

<!ELEMENT element1
(element2)>

<!ELEMENT element2
(#PCDATA)>
```

Abbildung 6.12.: DTD zu den in Abb. 6.10 und 6.11. dargestellten XML-Dokumenten

Auch der System-Identifizier ist nicht eindeutig. Der System-Identifizier enthält eine URI, die auf die Datei verweist, die die DTD enthält. Über den System-Identifizier weiß ein validierender Parser, wo er zum Validieren nach der DTD suchen kann. Der System-Identifizier kann lokal zum Benutzer, der das XML-Dokument erzeugt hat, angegeben sein oder auch auf einen öffentlich zugänglichen Server verweisen. Zum Beispiel könnten „www.cs.tu-berlin.de/~glaubitz/test.dtd“ und „File:C:\test2.dtd“ auf dieselbe DTD verweisen. Zwei XML-Dokumente können also, obwohl sie unterschiedliche System-Identifizier in der Doctype-Deklaration haben, zur selben DTD konform sein.

Am besten eignet sich als eindeutiger Bezeichner für die DTD der Public Identifizier. Ein Public Identifizier hat eine vom ISO standardisierte Form und ist unabhängig von dem Standort der DTD (also unter welcher URI sich die DTD befindet) und dem Wurzelement. Ein Public Identifizier hat die Form "Registration//Owner//DTD Description//Language". Die einzelnen Bestandteile haben folgende Bedeutung:

- Registration kann "+" oder "-" sein und zeigt an, ob der Public Identifizier offiziell bei einem Registration-Service registriert ist oder nicht.
- Der Owner ist der Ersteller der DTD.
- DTD zeigt an, daß es sich um eine DTD handelt.
- Description ist eine eindeutige Beschreibung der DTD.
- Language kennzeichnet die Sprache, in der die DTD verfasst ist, z.B. Deutsch oder Englisch.

Zum Beispiel kennzeichnet der Public Identifizier "-//IETF//DTD HTML 3.0//EN" die DTD für HTML. Ähnlich können auch eigene Public Identifizier erzeugt werden.

Das Doctype-Matching sollte die Möglichkeit bieten sowohl über den in der Doctype deklarierten Namen als auch über die SystemId oder die PublicId matchen zu können. Es

¹ Die XML-Spezifikation [XML] definiert keinen Namen für die DTD.

hängt dann von der jeweiligen Anwendung ab, wie eindeutig das Doctype-Matching sein muß.

Die Realisierung von XMLSpaces basiert im Moment auf DOM1. DOM1 bietet nur die Möglichkeit für den Zugriff auf den Namen des Doctypes, aber nicht für den Zugriff auf die System- oder den Public-Identifizier. Zur Zeit kann für das Doctype-Matching daher nur die Überprüfung des Doctype-Namens realisiert werden. Vollständiges Doctype-Matching ist erst ab DOM2 möglich und daher ein noch offener Punkt (siehe auch Kapitel 10).

6.2.3. Matching auf Gleichheit

Matching auf Gleichheit bedeutet, daß ein XML-Dokument matcht, wenn es gleich einem anderen XML-Dokument ist. Damit entspricht das Matching auf Gleichheit dem Linda-Matching für ein Actual im Template (dem Wert-Matching): das Template `<1,2>` matcht mit allen Tuples, die die gleichen Feldwerte haben.

Was bedeutet jedoch Gleichheit zwischen zwei XML-Dokumenten?

Ein XML-Dokument besteht aus: Elemente, Elementattribute, Text, PIs, Kommentare und CData-Abschnitte. Elemente, Elementattribute und Texte sind sicherlich fester Bestandteil eines XML-Dokumentes und sollten in den Vergleich einfließen. Aber wie sieht es mit PIs, CData-Abschnitte und Kommentare aus? Kommentare gehören eigentlich nicht zum Dokument und sollten mit keiner Semantik belegt sein, trotzdem kann es sinnvoll sein auch Kommentare beim Vergleich mit einzubeziehen.

Da jeder seine eigenen Vorstellungen von Gleichheit bei XML-Dokumenten hat, ist es am besten das Matching auf Gleichheit möglichst allgemein und flexibel zu halten, so daß jeder selbst bestimmen kann, welche Bestandteile eines XML-Dokumentes in den Vergleich mit einfließen. Daher sollte das Matching auf Gleichheit die Möglichkeit bieten bestimmte Bestandteile beim Vergleich zu ignorieren.

Das Matching auf Gleichheit berücksichtigt zunächst sämtliche Bestandteile eines XML-Dokumentes: Zwei XML-Dokumente sind gleich, wenn sie in genau derselben Reihenfolge die gleichen Elemente mit den gleichen Attributen, Texte, Kommentare, PIs und CData-Abschnitte enthalten.

Durch zusätzliche Argumente ist es möglich bestimmte Bestandteile vom Vergleich auszuschließen. Wenn man zum Beispiel als Argument `[PI, Comment]` mit angibt, werden Kommentare und PIs beim Vergleich ignoriert.

Abbildung 6.13. zeigt zwei XML-Dokumente, die nur gleich sind, wenn Kommentare und PIs beim Vergleich ignoriert werden.

<pre><element1> <element2> Test <element2> </element1></pre>	<pre><element1> <?unterschied1 anweisung?> <element2> <!-- Unterschied2 --> Test </element2> </element1></pre>
--	--

Abbildung 6.13.: Zwei *fast* gleiche XML-Dokumente

6.2.4. Matching mit Umbenennen von Tags und Attributen

Die hier beschriebene Matchingrelation baut auf dem XML-Matching mit Gleichheit (Kapitel 6.2.3.) und dem DTD-Matching (Kapitel 6.2.2.) auf und erweitert sie um die Möglichkeit Tags und Attribute umzubenennen. Ein möglicher Anwendungsfall wäre, daß man XML-Dokumente hat, die vom Prinzip her derselben DTD folgen, also vom Prinzip her denselben

Aufbau haben, jedoch länderspezifische Element- und Attributnamen. Das eine Dokument beschreibt zum Beispiel eine Adresse, wobei die Element- und Attributnamen englische Namen haben. Ein anderes Dokument beschreibt auch eine Adresse und hat denselben Aufbau. Jedoch werden statt englischer Element- und Attributnamen deutsche verwendet (Abbildung 6.14. und Abbildung 6.15.).

```

<adress>
  <name>
    <firstname>B</firstname>
    <lastname>Müller</lastname>
  </name>
  <street> Franklinstrasse </street>
  <city> Berlin </city>
</adress>

```

Abbildung 6.14.: Ein XML-Dokument mit englischen Bezeichnern

```

<adresse>
  <name>
    <vorname>B</vorname>
    <nachname>Müller</nachname>
  </name>
  <strasse> Franklinstrasse </strasse>
  <stadt> Berlin </stadt>
</adresse>

```

Abbildung 6.15.: Ein XML-Dokument mit deutschen Bezeichnern

Angenommen man hat ein XML-Dokument wie in Abbildung 6.14. und will mit Matching über Gleichheit ein XML-Dokument wie in Abbildung 6.15. aus dem XMLSpace herausholen. Wenn man das normale Matching über Gleichheit verwendet, werden die beiden XML-Dokumente nicht miteinander matchen, weil sie unterschiedliche Element- und Attributnamen haben. Wenn man jedoch zusätzlich zur Matchingrelation angibt, daß adress in adresse, firstname in vorname, lastname in nachname, street in strasse und city in stadt umbenannt werden soll, dann wären die beiden XML-Dokumente gleich. Ein entsprechender Anwendungsfall wäre, wenn man eine DTD mit deutschen Element- und Attributnamen hat und ein XML-Dokument heraussuchen will, daß vom Prinzip her der gleichen DTD folgt jedoch englische Element- und Attributnamen hat.

Durch die Verwendung von Wildcards, die bei einem Vergleich die Element- und Attributnamen ignorieren, kann dies zu einem rein strukturellen Vergleich ausgebaut werden. Angenommen man hat ein XML-Dokument wie in Abbildung 6.14. und sucht nach einem XML-Dokument, das die gleiche Struktur hat. Es sollen also beim Vergleich die Element- und Attributnamen ignoriert werden und nur die Struktur der XML-Dokumente miteinander verglichen werden. Abbildung 6.14. und Abbildung 6.15. sind unterschiedlich, weil sie unterschiedliche Elementnamen haben, wenn man jedoch nur die Baumstruktur betrachtet und dabei die Element- und Attributnamen ignoriert sind sie von der Struktur her gleich. Abbildung 6.16. bis 6.18. veranschaulicht dies.

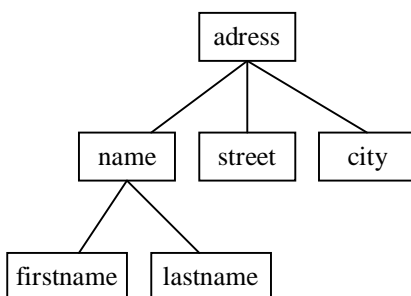


Abbildung 6.16.: Die Baumstruktur des XML-Dokumentes aus Abb. 6.14.

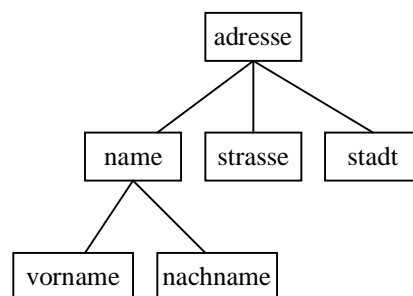


Abbildung 6.17.: Die Baumstruktur des XML-Dokumentes aus Abb. 6.15.

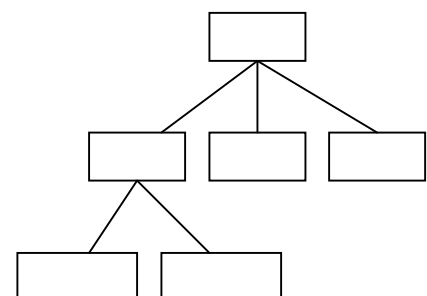


Abbildung 6.18.: Die Baumstruktur ohne Elementnamen

6.2.5. Matching mit einem Identifier im Wurzelement

Workspaces benötigt die Möglichkeit zum Matchen eines XML-Dokumentes, das einen eindeutigen Identifier als Attributwert im Wurzelement enthält.

Zu diesem Zweck kann natürlich XPath oder XQL verwendet werden (`./child::*[id(„Id-Wert“)]` oder `./*[id(„Id-Wert“)]`). Einer der Vorteile von XMLSpaces ist es jedoch, daß es problemlos um neue Matchingrelationen erweiterbar ist. Auf diese Art ist es möglich für bestimmte Anwendungsfälle spezifische Matchingrelationen, die schneller als andere sind, zu implementieren.

XPath ist sehr flexibel, schleppt aber gerade auch deshalb einen großen Balast mit sich herum (die XPath-Engine muß gestartet werden, der XPath-Ausdruck geparkt und ausgewertet werden...). Daher eignet es sich für Matchingrelationen, die häufig verwendet werden, spezifische und unflexible, aber dafür schnellere Matchingrelationen zu realisieren.

Das Matching mit einem Identifier im Wurzelement besteht bei der Verwendung von DOM als Repräsentation des XML-Dokumentes (wie im Kapitel 6.3.4. beschrieben, werden in XMLSpaces XML-Dokumente als DOM-Objekte dargestellt) aus folgendem einfachen Schritt:

```
Document.getDocumentElement().getAttribute(„Identifier“) == Wert
```

Hierdurch wird auf das Attribut „Identifier“ im Wurzelement des Dokumentes zugegriffen und überprüft, ob es gleich dem durch „Wert“ repräsentierten Identifier ist.

6.2.6. Boolsche Verknüpfungen von Matchingrelationen

Die Anwendung einer XML-spezifischen Matchingrelation auf ein XML-Dokument ergibt als Ergebnis einen Booleschen Wert: Wenn das XML-Dokument die Matchingrelation erfüllt, erhält man ein True, ansonsten ein False. Daher können Matchingrelationen über die Booleschen Operatoren AND, OR, NOT und XOR verknüpft werden. Auf diese Art ist es möglich aus mehreren einzelnen Matchingrelationen eine komplexe Matchingrelation zu bilden, z.B.:

- Suche ein Dokument, das der DTD 1 ODER der DTD 2 entspricht (OR-Verknüpfung).
- Suche ein Dokument, das einer bestimmten DTD entspricht UND diesen XPath erfüllt (AND-Verknüpfung).
- Suche ein Dokument, das NICHT dieser DTD entspricht (NOT-Verknüpfung).
- Suche ein Dokument, das entweder einen bestimmten XPath enthält oder einer bestimmten DTD entspricht, aber nicht beides (XOR-Verknüpfung).

6.2.7. Namespaces und Matching

Durch die Verwendung von Namespaces [W3C99] können Element- und Attributnamen global eindeutig gemacht werden, indem die Element- und Attributnamen durch global eindeutige URIs qualifiziert werden.

Das folgende Beispieldokument zeigt die Verwendung von Namespaces:

```
<akte xmlns:person = "www.cs.tu-berlin.de/~glaubitz/person"
      xmlns:auto = "www.cs.tu-berlin.de/~glaubitz/auto">
  <person:name person:alter="...">...</person:name>
  <auto:name> ...</auto:name>
</akte>
```


Als Element- und Attributnamen werden qualifizierte Namen verwendet. Ein qualifizierter Name hat die Form „Präfix:LokalerTeil“. Der Präfix ist über die Namespace-Deklaration in Form von „xmlns:Präfix = URI“ an die URI gebunden, die den Namespace identifiziert. Hierbei ist der Präfix nur der Platzhalter für den eigentlichen Namespace-Name in Form der URI. Durch die Kombination Präfix:LokalerTeil wird der zweideutige lokale Teil (in diesem Beispiel „name“) eindeutig gemacht.

Durch die Verwendung von qualifizierten Namen ist es möglich DTDs in anderen DTDs wiederzuverwenden, ohne sich darum Sorgen machen zu müssen, daß die beiden verknüpften DTDs evtl. gleiche Element- oder Attributnamen verwenden. Durch die Verwendung von Namespaces sind die *lokal* gleichen Element- und Attributnamen global eindeutig unterscheidbar.

Angenommen ein XML-Dokument enthält wie in dem obigen Beispiel Informationen über den Käufer eines Autos. Die DTD des XML-Dokumentes wurde aus zwei verschiedenen DTDs zusammengesetzt: Eine DTD für Dokumente zum Beschreiben von Personeninformationen und eine DTD für Dokumente zum Beschreiben von Informationen über Autos. Beide DTDs verwenden den Elementnamen „name“, einmal für den Bezeichner des Autos und einmal als Namen für den Käufer des Autos. Ohne Namespaces gäbe es keine Möglichkeit für die Software, die diese XML-Dokumente verarbeiten soll, diese beiden Elemente, die den gleichen Namen, aber eine unterschiedliche Bedeutung haben, voneinander zu unterscheiden.

Ähnlich kann man beim Matching in XMLSpaces durch die Verwendung von Namespaces die zu matchenden Element- oder Attributnamen eindeutiger beschreiben. Zum Beispiel könnte ein XMLSpace XML-Dokumente mit Autoinformationen und XML-Dokumente mit Personeninformationen enthalten. Nun will man nach einem XML-Dokument suchen, das die *Person* mit dem Namen „Mercedes“ enthält. XMLSpaces könnte ohne die Verwendung von Namespaces nicht zwischen Auto- und Personennamen unterscheiden, weil beide denselben Elementnamen „name“ verwenden.

Generell müssen die verschiedenen Matchingrelationen, wenn sie Namespaces unterstützen wollen, eine Möglichkeit bieten die durch Namespaces gegebenen Zusatzinformationen in Form von Namespace-URI, Präfix und lokaler Teil in die Matchingrelation mit einzubeziehen. Im Folgenden wird ein Überblick gegeben ob und wie die verschiedenen Matchingrelationen Namespaces unterstützen.

XPath unterstützt komplett Namespaces. Ein Knotentest der Form Präfix:LokalerTeil wählt nur die Knoten mit demselben Präfix und demselben lokalen Teil aus. Zum Beispiel wählt `child::auto:name` nur die Kind-Elemente mit dem qualifizierten Namen `auto:name` aus. Statt einem lokalen Teil kann auch der Wildcard „*“ verwendet werden. Zum Beispiel wählt `child::auto:*` sämtliche Kind-Elemente mit dem Präfix `auto` aus, egal welchen lokalen Teil sie haben. Das Problem bei den Knotentests ist, daß nur auf Präfixe hin überprüft werden kann und Präfixe eigentlich nur die Platzhalter für die Namespace-URIs sind. Ein Präfix, der sich auf den selben Namespace bezieht, kann in unterschiedlichen Dokumenten unterschiedlich definiert sein. Zum Beispiel haben die folgenden beiden Dokumente zwei unterschiedliche Präfixe (`auto` und `automobil`), die sich jedoch auf den gleichen Namespace beziehen. Es handelt sich also im Prinzip bei `auto:name` und `automobil:name` um dieselben Elemente:

```
<akte
  xmlns:auto="www.cs.tu-berlin.de/auto">
  <auto:name>...</auto:name>
  ...
</akte>
```

```
<akte
  xmlns:automobil="www.cs.tu-berlin.de/auto">
  <automobil:name>...</automobil:name>
  ...
</akte>
```

Wichtiger als die Präfixe sind also eigentlich die Namespace-URIs, die den verwendeten Namespace eindeutig identifizieren. Auf die Namespace-URI kann man in XPath mit Hilfe der Funktion namespace-uri() zugreifen. Zum Beispiel wählt child::*[namespace-uri()="www.cs.tu-berlin.de/auto" and local-name()="name"] alle Kind-Elemente mit dem lokalen Teil "name" und der Namespace-URI "www.cs.tu-berlin.de/auto" aus. Es wäre also egal, ob der Präfix "auto" oder "automobil" heißt, solange er sich auf den gleichen Namespace "www.cs.tu-berlin.de/auto" bezieht. Bei der Verwendung von XPath muß man also nicht die Präfixe kennen, die sich von Dokument zu Dokument unterscheiden können, sondern braucht lediglich die immer eindeutigen URIs zu kennen.

Da XPointer und XQL von XPath abgeleitet sind, ist die Namespace-Unterstützung in XPointer und XQL zur XPath-Unterstützung (bis auf einige Kleinigkeiten) identisch.

XML-QL unterstützt in [XMLQL] noch keine Namespaces und diskutiert Namespaces nur kurz als zukünftige Erweiterung an. Die Unterstützung von Namespaces in XML-QL wird wahrscheinlich ähnlich wie bei XPath erfolgen. Zum Beispiel soll folgende Anfrage nach den Namen der Personen suchen, die ein Auto mit dem Namen ‚Mercedes‘ gekauft haben:

```
WHERE
  <akte>
    <person:name> $p </person:name>
    <auto:name>'Mercedes'</auto:name>
  </akte>
CONSTRUCT $p
```

Beim DTD-Matching wird überprüft, ob ein XML-Dokument einer bestimmten DTD entspricht. Die Namespaces-Recommendation [W3C99] ändert nichts an der Verarbeitung einer DTD, d.h. die Überprüfung, ob ein XML-Dokument einer bestimmten DTD entspricht, ist unabhängig davon, ob Namespaces verwendet werden oder nicht.

Die Elemente und Attribute, die in einer DTD deklariert werden, müssen dieselben qualifizierten Namen "Präfix:Lokaler Teil" haben, wie die entsprechenden Elemente und Attribute im XML-Dokument, damit sie miteinander matchen z.B.:

<pre>... <akte xmlns:auto= "www.cs.tu-berlin.de/auto" xmlns:person="www.cs.tu.berlin.de/person" > <auto:name>...<auto:name> <person:name>...<person:name> </akte> ...</pre>	<pre>... <!ELEMENT akte (auto:name, person:name)> <!ELEMENT auto:name ...> <!ELEMENT person:name ...> ...</pre>
--	---

Die Namespace-Deklaration kann auch als Defaultwert in der DTD mit angegeben werden, z.B.:

```
<!ATTLIST akte
  xmlns:auto CDATA #FIXED "www.cs.tu-berlin.de/auto"
  xmlns:person CDATA #FIXED "www.cs.tu-berlin.de/person">
```

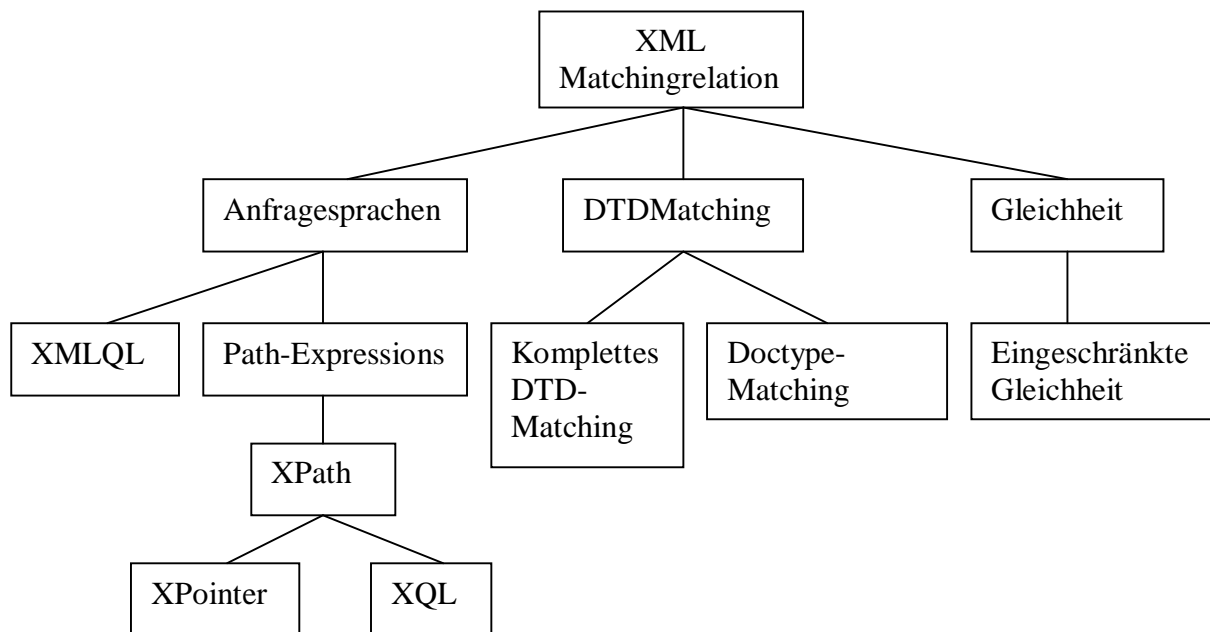
Da die Namespace-Recommendation die DTD-Verarbeitung unverändert lässt kann ein XML-Dokument, das vom Prinzip her die gleichen Namespaces wie eine DTD verwendet, aber andere Präfixe hat, nicht als der DTD folgend erkannt werden:

<pre> ... <akte xmlns:auto= "www.cs.tu-berlin.de/auto" xmlns:person="www.cs.tu.berlin.de/person" > <auto:name>...</auto:name> <person:name>...</person:name> </akte> ... </pre>	<pre> ... <!ELEMENT akte(automobil:name,person:name)> <!ATTLIST akte xmlns:automobil CDATA #FIXED "www.cs.tu-berlin.de/auto" xmlns:person CDATA #FIXED "www.cs.tu- berlin.de/person"> <!ELEMENT automobil:name ...> <!ELEMENT person:name ...> </pre>
--	--

Beim Matching auf Gleichheit müssen bei den Elementen- und Attributen neben dem lokalen Teil auch der Präfix und die Namespace-URI auf Gleichheit hin überprüft werden. In DOM2 ist dies durch den Zugriff auf die Node-Attribute `localName`, `prefix` und `namespaceURI` möglich. Wie auch beim Matching auf Gleichheit unterschieden werden kann, ob alle Bestandteile eines XML-Dokumentes in den Vergleich mit einfließen oder bestimmte Bestandteile wie z.B. Kommentare oder PIs ignoriert werden, kann auch bei der Berücksichtigung der Namespaces vorher festgelegt werden, ob bei Elementen und Attributen `localName`, `namespaceURI` und `prefix` übereinstimmen sollen oder ob es z.B. ausreicht, daß nur `localName` und `namespaceURI` übereinstimmen.

Wie gezeigt ist die Unterstützung von Namespaces innerhalb der Matchingrelationen vom Prinzip her generell möglich. Zur Zeit ist die Namespaces-Unterstützung in XMLSpaces noch nicht realisiert, weil die Implementierung DOM1 zur Repräsentation der XML-Dokumente verwendet und Namespaces erst ab DOM2 unterstützt werden (siehe auch Kapitel 10).

6.2.8. Vergleich/Zusammenfassung der Matchingrelationen



6.19.: Überblick über die Matchingrelationen

Die hier beschriebenen Matchingrelationen lassen sich grob in Matchingrelationen über Anfragesprachen, über die DTD eines XML-Dokumentes und über die Gleichheit mit einem anderen XML-Dokument unterscheiden. Diese drei Arten von Matchingrelationen beschreiben unterschiedliche, grundlegend verschiedene Matchingrelationen mit verschiedenen Anwendungsbereichen. So lässt sich zum Beispiel mit XPath kein DTD-Matching realisieren und DTD-Matching eignet sich nicht für das Matchen über einzelne Elemente im XML-Dokument.

Die Anfragesprachen beziehen sich nur auf einzelne Bestandteile eines XML-Dokumentes, wohingegen das DTD-Matching und das Matching auf Gleichheit sich auf das XML-Dokument als Ganzes beziehen.

Die Anfragesprachen lassen sich in zwei Gruppen unterteilen: Anfragesprachen, die ausschließlich Path-Expressions verwenden, und XML-QL, das ein CONSTRUCT-WHERE-IN-Konstrukt ähnlich zu SELECT-FROM-WHERE verwendet.

Die Basissprache für die Path-Expression-Anfragesprachen bildet XPath, von der XPointer und XQL abgeleitet sind. Alle drei bieten für das Matching die gleiche Funktionalität, indem sie über einen Pfadausdruck auf bestimmte Bestandteile innerhalb des XML-Dokumentes verweisen und auf diese Art die Matchingrelation beschreiben.

XML-QL stammt aus dem Umfeld der Datenbankentwicklung im Gegensatz zu XQL, das aus den Anforderungen des Bereiches der Dokumentenwissenschaften entstanden ist. Die primäre Aufgabe von XQL liegt im Wiederauffinden von Daten in einem Dokument, wohingegen XML-QL auch das Konstruieren von Daten unterstützt. Dies ist jedoch für das Matching irrelevant. XML-QL verwendet ein SELECT-FROM-WHERE ähnliches Gebilde der Form WHERE-CONSTRUCT-IN. In XML-QL werden nur Elemente, Attribute und Texte als relevante Bestandteile eines XML-Dokumentes betrachtet und es wird im Gegensatz zu XPath keine Unterstützung für den Zugriff auf Kommentare oder PIs geboten.

Das DTD-Matching entspricht dem Typ-Wert-Matching von Linda. Man kann entweder ein XML-Dokument von einem Parser komplett gegen eine DTD validieren lassen oder die in der Doctype-Deklaration (<!DOCTYPE...>) angegebenen Namen, SystemId oder PublicId verwenden, die aber (bis auf PublicId) nicht immer eindeutig sind.

Das Matching auf Gleichheit kann als das Wert-Wert-Matching von Linda betrachtet werden. Bei der Überprüfung der Gleichheit können entweder sämtliche Bestandteile eines XML-Dokumentes berücksichtigt werden oder bestimmte Bestandteile, wie z.B. Kommentare oder PIs, ignoriert werden.

Die folgende Tabelle fasst noch einmal die wesentlichen Eigenschaften der XML-Matchingrelationen zusammen.

Kategorie	Name	Matching über	Verwendungszweck	Besonderheiten	Namespaces
Anfragesprachen	XPath	Path-Expressions	Zum Suchen von XML-Dokumenten, die bestimmte Elemente, Attribute, Text, PIs oder Kommentare in der über die Path-Expression angegebenen Weise enthalten.	<ul style="list-style-type: none"> - Umfangreiche Axis. - Unterstützung von Funktionen wie z.B. id(). - Zugriff auf alle Bestandteile (Elemente, Attribute, Text, PIs, Comments) - Keine Tag-Variablen wie XML-QL, sondern nur *-Wildcards 	Ja
	XPointer	Path-Expressions	Wie Xpath	<ul style="list-style-type: none"> - Unterscheidet sich von XPath nur durch das Unterstützen von Punkten und Bereichen (fürs Matching unrelevant!) 	Ja
	XQL	Path-Expressions	Wie Xpath	<ul style="list-style-type: none"> - Baut auf den Abkürzungen von XPath auf. - Hat keine Axis. (Path-Expressions können nur Top-down angegeben werden über „/“ und „//“) 	Ja
	XML-QL	WHERE-IN-CONSTRUCT	Zum Suchen von XML-Dokumenten, die bestimmte Elemente, Attribute und Texte über die in der WHERE-IN-CONSTRUCT-Klausel beschriebenen Weise enthalten.	<ul style="list-style-type: none"> - Unterstützt auch das Konstruieren von Daten (fürs Matching unrelevant) - Tag-Variablen - Keine Axis in Path-Expressions (nur Top-down) - Zugriff nur auf Elemente und Attribute 	Noch nicht realisiert. (vom Prinzip her ja)
DTD Match	Komplettes DTD-Matching	DTD	Zum Suchen von XML-Dokumenten, die einer bestimmten DTD entsprechen.		Ja
	Doctype-Matching	<!Doctype ...>	Zum Suchen von XML-Dokumenten, die der in der Doctype-Deklaration über Name, SystemId oder PublicId angegebenen DTD entsprechen.		Ja
Matching auf Gleichheit	Gleichheit	Komplettes XML-Dokument	Zum Suchen von XML-Dokumenten, die gleich einem anderen XML-Dokument sind		Ja
	Eingeschränkte Gleichheit	Teilweises XML-Dokument	Zum Suchen von XML-Dokumenten, die gleich einem anderen XML-Dokument sind (wobei einige Bestandteile ignoriert werden.)		Ja

6.3. Realisierung in TSpace

6.3.1. Einordnung in ähnliche Projekte

Es gibt bereits viele andere Projekte [Cabri00, TSpace, Abraham, Moffat], in denen Linda um XML-Dokumente erweitert wurde. Der wesentliche Unterschied von XMLSpaces gegenüber den anderen Projekten ist, daß XMLSpaces mehrere Matchingrelationen anbietet und problemlos um neue Matchingrelationen erweitert werden kann. Dies ist einerseits erforderlich, um den Anforderungen von Workspaces zu genügen, da Workspaces wie im Kapitel 4 beschrieben unterschiedliche Matchingrelationen benötigt. Außerdem ist das Anbieten einer einzigen Matchingrelation auf Grund der komplexen Struktur von XML-Dokumenten oft unzureichend.

Die von TSpace angebotene Unterstützung von XML-Dokumente bietet zum Beispiel nur die Möglichkeit die Matchingrelation über einen XQL-Ausdruck anzugeben. Über ein XQL-Ausdruck ist es jedoch nicht möglich ein Dokument herauszusuchen, das einer bestimmten DTD entspricht und damit scheiden Anwendungsfälle, die DTD-Matching erfordern, für TSpaces aus.

6.3.2. TupleSpaces, Tuples und Fields in TSpaces

Die XMLSpace-Realisierung baut auf TSpaces [TSpace] auf, ignoriert dabei jedoch völlig die bereits von TSpace vorhandene Unterstützung von XML-Dokumenten, weil diese nicht in das von XMLSpace verfolgte Konzept passt.

Im folgenden wird die normale Linda-Realisierung von TSpace beschrieben, auf der die XML-Erweiterung von XMLSpaces aufsetzt.

In TSpace wird ein Tuplespace über die Klasse TupleSpace repräsentiert. Der Aufruf `new TupleSpace("test")` erzeugt entweder ein neues Tuplespace mit dem Namen „test“ oder greift auf ein bereits vorhandenen Tuplespace namens „test“ zu.

Die Tuples innerhalb eines Tuplespace werden durch die Klasse Tuple und die Felder innerhalb eines Tuples über die Klasse Field dargestellt. Zum Beispiel erzeugt `new Tuple(new Field("Hallo"), new Field(1))` ein Tuple mit zwei Feldern. Das erste Feld enthält den Stringwert „Hallo“ und das zweite Feld den Integerwert 1.

Tuples können über die TupleSpace-Methode `write` (entspricht dem `out` in Linda) zum Tuplespace hinzugefügt und über die Methoden `waitToRead` und `waitToTake` (entsprechen den Linda-Operatoren `rd` und `in`) wiedergewonnen werden. TSpace bietet neben `waitToRead` und `waitToTake` auch die nicht-blockierenden Methoden `read` und `take`, die in dem Fall, daß kein matchendes Tuple gefunden wurde, einen Null-Wert zurückliefern.

In TSpace wird nicht zwischen Template und Tuples unterschieden. Ein Template ist ein Objekt von der Klasse Tuple. Formale Felder werden durch Field-Objekte, die nur einen Typ, aber keinen Wert haben, repräsentiert. Zum Beispiel hat das Template `new Tuple(new Field("Hallo"), new Field(Integer.class))` als zweites Feld ein formales Feld vom Typ Integer.

Der folgende Programmcode fasst das eben beschriebene noch einmal zusammen. Der Programmcode erzeugt einen Tuplespace mit dem Namen „test“, schreibt ein Tuple hinein und liest es über `waitToRead` wieder aus:

```
TupleSpace ts = new TupleSpace("test");
Tuple t = new Tuple(new Field("Hallo"));
ts.write(t);
Tuple template = new Tuple(new Field(String.class));
Tuple Ergebnis = ts.waitToRead(template);
```

Die Methode `Tuple.matches(Tuple t)` realisiert das Matchingverhalten zwischen Tuples und gibt `true` zurück, wenn zwei Tuple miteinander matchen. `Tuple.matches()` wird in der Klasse `TupleSpace` automatisch von den Methoden `waitToRead`, `waitToTake`, `read`, `take` usw. aufgerufen, um zu überprüfen, ob die im `TupleSpace` abgelegten Tuple mit dem `Template-Tuple` matchen. Zum Beispiel durchläuft das in dem Beispiel-Programmcode angegebene `ts.waitToRead(template)` sämtliche Tuple im `TupleSpace ts` und ruft für jedes Tuple `x` in `ts` die Methode `template.matches(x)` auf. Das erste Tuple, für das `template.matches(x)` `true` ergibt wird als Ergebnis zurückgegeben.

Der Aufruf `t1.matches(t2)` – wobei `t1` und `t2` vom Typ `Tuple` sind und `t1` das `Template-Tuple` sein muß – hat folgenden Ablauf:

- Wenn `t1` keine Instanz der Klasse von `t2` ist, dann wird `false` zurückgegeben.
- Wenn die Anzahl der Felder von `t1` und `t2` nicht übereinstimmt, wird `false` zurückgegeben.
- Wenn die `Field.matches()`-Methode – die überprüft, ob zwei Felder miteinander matchen – für jedes Feld im Tuple `true` ergibt, wird `true` zurückgegeben.
- Ansonsten wird `false` zurückgegeben.

Das Matchingverhalten zwischen den einzelnen Feldern zweier Tuple wird von der Methode `Field.matches()` realisiert. Der Aufruf `f1.matches(f2)` – wobei `f1` und `f2` vom Typ `Field` sind und `f1` das Feld des Templates und `f2` das Feld des Tuples darstellt- hat folgenden Ablauf:

- Wenn `f1` ein formales Feld ist und der in `f1` angegebene Typ mit dem Typ des Wertes von `f2` übereinstimmt, wird ein `true` zurückgegeben (entspricht dem Typ-Wert-Matching von Linda).
- Wenn `f1` einen Wert enthält, der gleich dem Wert von `f2` ist, wird `true` zurückgegeben (entspricht dem Wert-Wert-Matching von Linda).
- Ansonsten wird `false` zurückgegeben.

Durch das Überschreiben der `Field.matches()`-Methode kann man das Matchingverhalten zwischen Feldern ändern. Hier kann z.B. ein XML-spezifisches Matchingverhalten realisiert werden. Auf diese Art ist es möglich `TSpace` um XML-Dokumente zu erweitern, indem von der Klasse `Field` eine Klasse `XMLDocField` abgeleitet wird (die die XML-Dokumente in einem Tuple repräsentiert) und die `Field.matches()`-Methode überschrieben wird, um ein XML-spezifisches Matching zu realisieren.

6.3.3. Überblick über die Erweiterung von TSpaces um XML-Dokumente

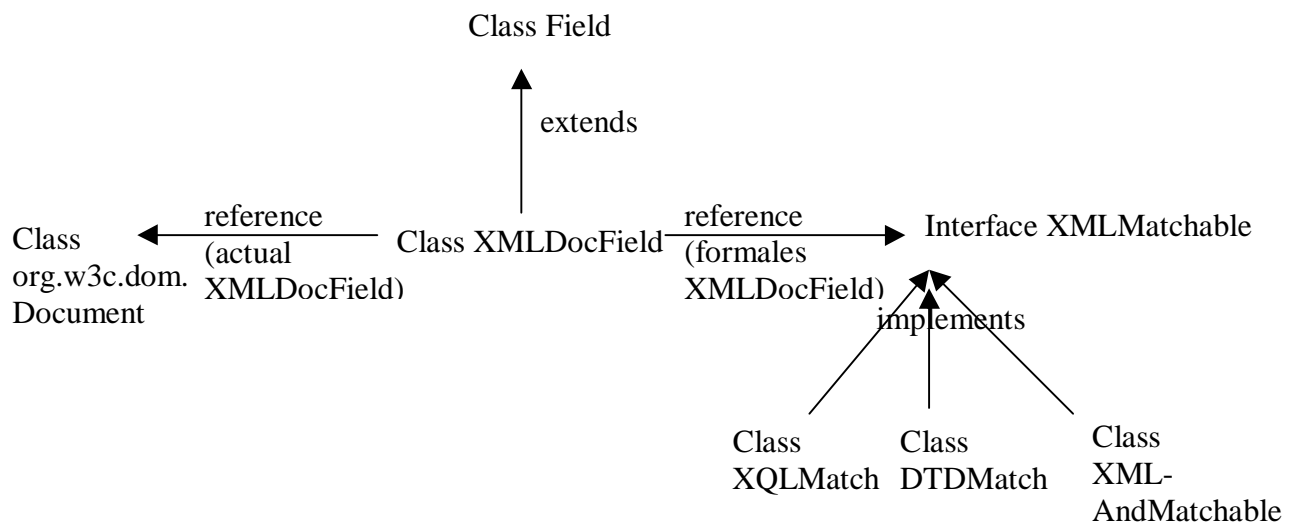


Abbildung 6.20.: Die Klassenstruktur zur Erweiterung um XML-Dokumente

Das Ziel besteht darin, TSpaces so zu erweitern, dass XML-Dokumente in dem Tuplespace abgespeichert und durch verschiedene XML-spezifische Matchingrelationen wiedergewonnen werden können.

Zu diesem Zweck wird die Klasse XMLDocField von der Klasse Field abgeleitet. XMLDocField entspricht einem Feld eines Tuples, das ein XML-Dokument (ein *actual Feld*) oder eine XML-Matchingrelation zum Matchen eines XML-Dokumentes (ein *formales Feld*) enthalten kann.

XMLDocField stellt entsprechend Konstruktoren zum Erzeugen von actual und formalen XMLDocFields zur Verfügung:

- Actual XMLDocFields enthalten als Feldwert ein XML-Dokument in Form eines DOM-Document-Objektes.
- Formale XMLDocFields enthalten als Felderwert ein XMLMatchable-Objekt, das das zu matchende XML-Dokument beschreibt.

Um ein XML-spezifisches Matching zu realisieren, überschreibt die XMLDocField.matches()-Methode die Field.matches()-Methode, die von TSpaces aufgerufen wird, um zu überprüfen, ob die einzelnen Felder zweier Tuple miteinander matchen.

Klassen, die ein XML-spezifisches Matching realisieren (z.B. XQL- oder DTD-Matching), müssen das Interface XMLMatchable implementieren. Das Interface XMLMatchable spezifiziert die Methode xmlMatch(Document doc), die von XMLDocField.matches() aufgerufen wird, um zu überprüfen, ob ein im Tuplespace abgelegtes XML-Dokument der angegebenen Matchingrelation (z.B. in Form eines XQL-Ausdruckes oder einer DTD) entspricht.

Die Klassen XQLMatch, DTDMatch und XMLAndMatchable sind Beispiele für XML-spezifische Matchingroutinen, die das Interface XMLMatchable implementieren:

- XQLMatch ermöglicht die Angabe einer Matchingrelation in Form eines XQL-Ausdruckes
- DTDMatch überprüft, ob ein XML-Dokument einer angegebenen DTD entspricht
- XMLAndMatchable ermöglicht es mehrere XML-Matchingrelationen über einen AND-Operator miteinander zu verknüpfen.

6.3.4. Die Klasse XMLDocField

Ein XMLDocField ist ein Feld eines Tuples, das entweder ein XML-Dokument oder eine XML-Matchingrelation in Form eines XMLMatchable-Objektes als Feldwert enthalten kann.

Die Klasse XMLDocField ist von Field abgeleitet und bietet XML-spezifische Konstruktoren und überschreibt die Field.matches()-Methode um ein XML-spezifisches Matchingverhalten zu realisieren.

Es gibt zwei Arten von Konstruktoren in XMLDocField:

- Mehrere Konstruktoren zum Erzeugen von actual XMLDocFields:
 - XMLDocField(String filename): Erhält den Verweis auf eine Datei, die das XML-Dokument enthält. XMLDocField parst mit Hilfe von XML4J die XML-Datei und legt das daraus resultierende DOM-Document-Objekt als Wert im XMLDocField ab.
 - XMLDocField(Document xmlDoc): Erhält das XML-Dokument als DOM-Document-Objekt und legt es als Wert im XMLDocField ab.
- Einen Konstruktor zum Erzeugen eines formalen XMLDocFields:
 - XMLDocField(XMLMatchable xmlMatch): Erzeugt ein formales XMLDocField mit dem angegebenen XMLMatchable-Objekt als Wert.

Ein XML-Dokument wird in XMLDocField intern als ein DOM-Document-Objekt abgelegt. DOM (Document Object Model) [DOM1, DOM2] ist ein W3C-Standard, in dem XML-Dokumente logisch als Bäume aus Objekten repräsentiert werden. DOM spezifiziert die einzelnen Objekte, aus denen der Dokument-Baum besteht und die Interfaces für den Zugriff auf die Objekte. Für jeden Bestandteil eines XML-Dokumentes ist ein entsprechendes Interface für eine Klasse spezifiziert, die diesen Bestandteil darstellt: Document, Element, Attr, Comment, Text, Processing Instruction, CDATASection usw. Das Interface Element bietet zum Beispiel Methoden für den Zugriff auf die Kind-Elemente (childNodes()) oder auf den Tagnamen des Elementes (tagName) an. Abbildung 6.2.1. zeigt die Darstellung eines XML-Dokumentes als einen DOM-Baum.

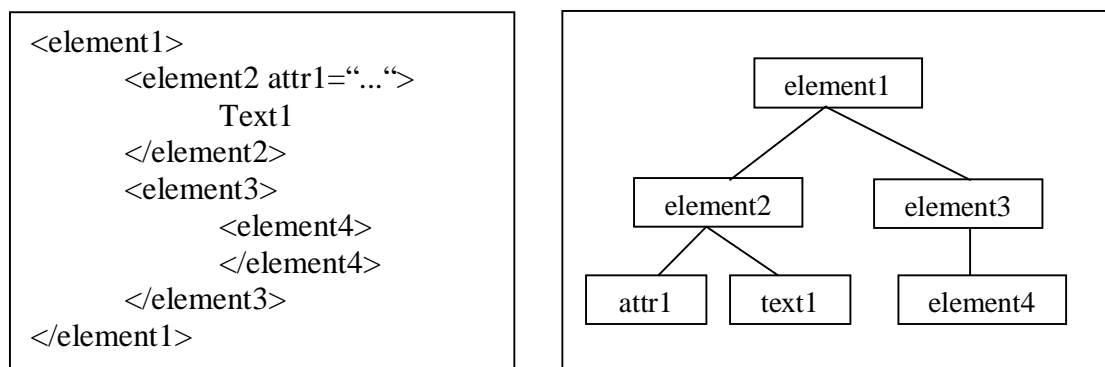


Abbildung 6.2.1.: Ein XML-Dokument und seine Darstellung als DOM-Baum

Der Vorteil von der Verwendung von DOM als Repräsentation für XML-Dokumente ist, daß DOM ein W3C-Standard ist und damit die Wahrscheinlichkeit hoch ist, daß DOM von Matching-Engines wie z.B. Engines für XQL, XPath, XMLQL usw. unterstützt wird und diese sich daher einfach in XMLSpaces einbinden lassen. Ein weiterer Vorteil ist, daß DOM ein sehr einfaches umfangreiches API für den Zugriff auf die einzelnen Bestandteile eines XML-Dokumentes anbietet und damit die Entwicklung eigener Matching-Engines gut unterstützt.

Zur Zeit verwendet XMLSpaces DOM1. Der wesentliche Unterschied und Nachteil von DOM1 zu DOM2 ist, daß DOM1 keine Namespaces unterstützt. Die Unterstützung von

Namespaces erfordert einen Umstieg auf DOM2 und ist ein noch offener Punkte (siehe auch Kapitel 10).

Formale Felder werden in einem XMLDocField als Objekte von Klassen, die das XMLMatchable-Interface implementiert haben, abgelegt. Das Interface XMLMatchable spezifiziert die Methode xmlMatch(Document xmlAsDoc), die als Argument ein XML-Dokument erhält und überprüft, ob das XML-Dokument mit der durch das XMLMatchable-Objekt repräsentierten Matchingrelation übereinstimmt.

Um ein XML-spezifisches Matchingverhalten zu realisieren, wurde in XMLDocField die von Field geerbte matches()-Methode überschrieben, die in TSpaces automatisch von Tuple.matches() aufgerufen wird, um zu überprüfen, ob die einzelnen Felder zweier Tuple miteinander matchen.

Für f1.matches(f2) ergibt sich folgender Ablauf (wobei f1 ein formales oder actual XMLDocField sein kann und f2 ein actual Feld sein muß; f1 entspricht dem Feld eines Template-Tuples und f2 entspricht dem Feld eines im Tuplespace abgelegten Tuples):

- f2 muß vom Typ XMLDocField sein, ansonsten wird false zurückgegeben.
- Wenn f1 und f2 ein XML-Dokument als Feldwert enthalten, dann wird auf exakte Gleichheit hin überprüft.
- Wenn f1 ein XMLMatchable-Objekt enthält und f2 ein XML-Dokument, dann wird das Ergebnis der XMLMatchable.xmlMatch()-Methode zurückgegeben, die das XML-spezifische Matching realisiert:

```
return ((XMLMatchable)this.getValue()).
    xmlMatch(((Document)f.getValue()));
```

- Ansonsten wird false zurückgegeben.

Die folgende Tabelle fasst das Matchingverhalten von XMLDocField.matches noch einmal zusammen:

Feldwert von f1	Feldwert von f2	Semantik von f1.matches(f2)
Document	Document	Überprüfung auf exakte Gleichheit
XMLMatchable	Document	Überprüft durch Aufruf der XMLMatchable.matches-Methode, ob das Document dem angegebenen XMLMatchable-Ausdruck entspricht
Document	XMLMatchable	False.
XMLMatchable	XMLMatchable	False.
Sonstiges	Sonstiges	False

Die XMLDocField.matches-Methode unterstützt *nicht* das Wiedergewinnen von formalen Tuples aus dem Tuplespace, die formale XMLDocFields enthalten: f1.matches(f2) ergibt immer false, wenn f2 ein XMLMatchable-Objekt ist. Der Grund dafür ist, daß ansonsten eine doppeldeutige Semantik entstehen würde. Wenn man ein waitToRead oder waitToTake mit einem Template-Tuple ausführt, das ein XMLDocField enthält, erwartet man als Ergebnis ein Tuple mit einem XML-Dokument zurückzubekommen und nicht ein Tuple mit einem XMLMatchable-Objekt.

Da nicht das Wiedergewinnen von formalen Tuples aus dem Tuplespace unterstützt wird, ist es auch nicht sinnvoll zu erlauben, daß formale Tuples im Tuplespace abgelegt werden können. Daher wurde von der Klasse Tuplespace die Klasse XMLSpace abgeleitet und die Methode write so überschrieben, daß bei dem Versuch ein formales Tuple im Tuplespace abzulegen eine Exception ausgelöst wird.

6.3.5. Das Interface XMLMatchable

Das Interface XMLMatchable muß von den Klassen, die eine XML-spezifische Matchingroutine (XQL-, DTD-Matching usw.) realisieren, implementiert werden.

XMLMatchable spezifiziert die Methode xmlMatch, die das XML-Matchingverhalten realisiert und von der XMLDocField.matches()-Methode aus aufgerufen wird, um zu überprüfen, ob ein im Tuplespace abgelegtes XML-Dokument mit der im XMLMatchable-Objekt angegebenen Matchingrelation übereinstimmt:

```
interface XMLMatchable{
    public boolean xmlMatch(Document xmlAsDoc);
}
```

Die Methode xmlMatch erhält als Argument ein XML-Dokument in Form eines DOM-Document-Objektes. XmlMatch überprüft, ob das übergebene XML-Dokument dem durch das XMLMatchable-Objekt repräsentierten XML-Matchingausdruck entspricht und gibt true oder false zurück.

Die Integration von bereits vorhanden XML-Matching-Engines (wie z.B. XQL- oder XPath-Engines) als XML-Matchingrelation ist besonders einfach, wenn die Matching-Engine in Java geschrieben ist und DOM-Document-Objekte verwendet. Ansonsten müsste das übergebene DOM-Document-Objekt in der xmlMatch-Methode zunächst in ein entsprechend passendes Format der Matching-Engine umgewandelt werden.

So besteht zum Beispiel die Realisierung der Klasse XQLMatch, die das XQL-Matching umsetzt und mit Hilfe der GMD-IPSI XQL-Engine vom GMD Forschungszentrum realisiert wurde, einfach aus einem Weiterleiten des xmlMatch-Aufrufes an die entsprechende match-Methode der GMD-IPSI-XQL-Engine, die überprüft, ob ein XML-Dokument einen XQL-Ausdruck erfüllt:

```
public boolean xmlMatch(Document xmlAsDoc){
    return XQL.match(query, xmlAsDoc);
}
```

Die Variable query enthält den XQL-Ausdruck, gegen den das XML-Dokument gematcht werden soll, und wird über den XQLMatch-Konstruktor XQLMatch(String xqlQuery) initialisiert:

```
String query;

XQLMatch(String xqlQuery){
    query = xqlQuery;
}
```

Damit besteht die gesamte Realisierung von XQL-Match aus folgendem kurzen Programmcode:

```
package matchingrelation;
import xmlspaces.XMLMatchable;
import java.io.*;
import org.w3c.dom.Document;
import de.gmd.ipsi.xql.*;

public class XQLMatch implements XMLMatchable{
```

```

String query;

public XQLMatch(String xqlQuery){
    query = xqlQuery;
}

public boolean xmlMatch(Document xmlAsDoc){
    return XQL.match(query, xmlAsDoc);
}
}

```

Die Kürze des Programmcodes resultiert natürlich daraus, daß die GMD-IPSI-XQL-Engine so gut ins XMLSpace-Konzept passt, weil sie den Methodenaufwurf `XQL.match()` anbietet, der mit DOM-Document-Objekten arbeitet. Schwieriger und umfangreicher wird der Programmcode, wenn die Engine ein anderes Datenformat verlangt oder es überhaupt keine Engine für das gewünschte Matchingverhalten gibt und man das Matchingverhalten komplett selbst realisieren muß.

6.3.6. Beispiel

Der folgende Programmcode zeigt ein komplettes Beispiel für die Verwendung von `XMLDocField` und XQL-Matching. Hierbei wird zunächst ein Tuple mit einem XML-Dokument im XMLSpace abgelegt und anschließend über ein XQL-Matching wieder herausgeholt:

```

XMLSpaces ts = new XMLSpace("test");
XMLDocField f = new XMLDocField("test1.xml");
Tuple t = new Tuple(new Field("Key1"),f);
ts.write(t);
XMLDocField formalesXMLField = new XMLDocField(new
    XQLMatch("//test1"));
Tuple template = new Tuple("Key1",formalesXMLField);
Tuple Ergebnis = ts.waitToRead(template);
Document xmlDoc =
    (Document)Ergebnis.getField(1).getValue();

```

6.3.7. Die Klasse XMLAndMatchable

Im Folgenden wird die Realisierung der booleschen Verknüpfung (AND, OR, NOT, XOR) mehrerer Matchingrelationen am Beispiel der AND-Verknüpfung demonstriert.

Die Klasse `XMLAndMatchable` ermöglicht es mehrere XML-Matchingroutinen über einen AND-Operator miteinander zu verknüpfen. Auf diese Art ist es möglich z.B. nach einem XML-Dokument zu suchen, das einer bestimmten DTD entspricht UND außerdem einen bestimmten XQL-Ausdruck erfüllt.

Der Konstruktor-Aufruf `new XMLAndMatchable()` erzeugt ein leeres `XMLAndMatchable`-Objekt. Über die Methode `addXMLMatch(XMLMatchable xmlMatch)` können weitere mit AND zu verknüpfende Matchingrelationen zu einem `XMLAndMatchable`-Objekt hinzugefügt werden. Die mit AND zu verknüpfenden XML-Matchingrelationen werden intern in einem Vector abgelegt.

Die Methode `xmlMatch(Document xmlAsDoc)`, die von `XMLDocField.matches` aufgerufen wird, um zu überprüfen, ob ein XML-Dokument mit dem

XMLAndMatchable-Objekt übereinstimmt, durchläuft den Vector, der die mit AND zu verknüpfenden XML-Matchingrelationen (XMLMatchable-Objekte) enthält, und ruft für jedes XMLMatchable-Objekt die Methode xmlMatch mit dem XML-Dokument xmlAsDoc auf. Nur wenn alle xmlMatch-Aufrufe der einzelnen XMLMatchable-Objekte true ergeben, gibt die xmlMatch-Methode vom XMLAndMatchable-Objekt ein true zurück, ansonsten false. Auf diese Art wird eine AND-Verknüpfung der einzelnen XML-Matchingrelationen erreicht.

Der folgende Programmcode zeigt, wie mit Hilfe der Klasse XMLAndMatchable nach einem XML-Dokument gesucht werden kann, das einer bestimmten DTD und einem bestimmten XQL-Ausdruck entspricht:

```
XMLAndMatchable xmlAnd = new XMLAndMatchable();
xmlAnd.addXMLMatch(new DTDMatch(...));
xmlAnd.addXMLMatch(new XQLMatch(...));
XMLDocField f = new XMLDocField(xmlAnd);
Tuple template = new Tuple("Key1", f);
Tuple ergebnis = ts.waitForRead(template);
```

6.3.8. Anleitung zum Einfügen neuer Matchingrelationen in XMLSpace

Um in XMLSpace eine neue Matchingrelation einzubinden, muß man eine neue Klasse für die Matchingrelation schreiben, die das XMLMatchable-Interface implementiert, z.B.

```
class neueRelation implements XMLMatchable{
    ...
}
```

Das XMLMatchable-Interface spezifiziert die Methode xmlMatch, die von XMLSpaces aufgerufen wird, um zu überprüfen, ob ein XML-Dokument im Tuplespace mit der durch das XMLMatchable-Objekt realisierten Matchingrelation matcht:

```
public boolean xmlMatch(Document xmlAsDoc);
```

Die Methode xmlMatch muß von der Klasse neueRelation implementiert werden, um das neue Matchingverhalten zu realisieren. Die Methode xmlMatch erhält als Argument ein DOM-Document-Objekt, das das zu überprüfende XML-Dokument repräsentiert, auf das die Matchingrelation angewendet wird. Die Methode xmlMatch der Klasse neueRelation nimmt dieses XML-Dokument, wendet die von dieser Klasse realisierte Matchingrelation darauf an und gibt ein true zurück, wenn das XML-Dokument mit der Matchingrelation übereinstimmt und ansonsten ein false. Anhand dieses Rückgabewertes kann XMLSpace entscheiden, ob ein XML-Dokument eine Matchingrelation erfüllt oder nicht:

```
public boolean xmlMatch(Document xmlAsDoc);
    // Realisierung des Matchingverhaltens.
    // Wenn xmlAsDoc matcht: return true
    // Wenn xmlAsDoc nicht matcht: return false
    ...
}
```

Auf diese Art ist die neue Matchingrelation realisiert. Neben der vom XMLMatchable-Interface geforderten Methode xmlMatch kann die neue Matchingklasse beliebige

Konstruktoren und Set-Methoden enthalten, über die spezifische Werte für die neue Matchingrelation gesetzt werden können (z.B. beim DTD Matching die zu matchende DTD, bei XQL-Matching der XQL-Ausdruck, mit dem gematcht werden soll usw.)

Als Gesamte Klassenstruktur erhält man vom Prinzip her folgendes Bild:

```

class neueRelation implements XMLMatchable{
    ...

    // Konstruktor und Set-Methoden zum Setzen
    // der zum Matchen notwendigen Werte
    ...

    public boolean xmlMatch(Document xmlAsDoc);
        // Realisierung des Matchingverhaltens.
        // Wenn xmlAsDoc matcht: return true
        // Wenn xmlAsDoc nicht matcht: return false
        ...
    }
}

```

Die neue Matchingrelation wird verwendet, indem ein neues Objekt dieser Klasse erzeugt und an XMLDocField übergeben wird:

```

XMLSpaces ts = new XMLSpace("test");
XMLMatchable match = new neueRelation(...);
XMLDocField formalesXMLField = new XMLDocField(match);
Tuple template = new Tuple("Key1",formalesXMLField);
Tuple Ergebnis = ts.waitForRead(template);

```

6.3.9. Überblick über realisierte Matchingrelationen

Die folgende Tabelle gibt einen Überblick über die bisher realisierten Matchingrelationen in XMLSpaces:

Matching-relation	Funktion	Beispiel	Verwendete Software	Klassenname
XPath	Überprüft, ob ein XML-Dokument einen XPath-Ausdruck erfüllt.	/descendant::betriebssystem[attribute::version="4"] sucht sämtliche XML-Dokumente mit einem betriebssystem-Element, das ein Attribut version mit dem Wert 4 hat.	Xalan-Java von Apache Software Foundation	Xpath Match
XQL	Überprüft, ob ein XML-Dokument einen XQL-Ausdruck erfüllt.	Beispiel wie bei XPath als XQL-Ausdruck: //betriebssystem[@version="4"]	GMD-IPSI-XQL-Engine vom GMD – Forschungszentrum Informationstechnik GmbH	XQL Match
XMLQL	Überprüft, ob ein XML-Dokument einen XMLQL-Ausdruck erfüllt.	Beispiel wie bei XPath als XMLQL-Ausdruck: WHERE <*.betriebssystem version="4">\$x</betriebssystem > IN XMLSpace CONSTRUCT \$x	XML-QL-Engine von AT&T	XMLQL Match

DTD	Überprüft, ob ein XML-Dokument einer angegebenen DTD folgt.	Suche ein XML-Dokument, das der auto.dtd folgt.	XML4J von IBM	DTD Match
Doctype	Überprüft, ob der in einem XML-Dokument in <!DOCTYPE name...> angegebene Doctype-Name einen bestimmten Wert hat.	Suche ein Dokument, das als Doctype-Name "auto" hat.	DocumentType-Interface von DOM (realisiert durch XML4J von IBM)	Doctype Match
Gleichheit	Überprüft, ob ein XML-Dokument gleich einem anderen XML-Dokument ist, wobei sämtliche Bestandteile eines XML-Dokumentes im Vergleich mit einbezogen werden.	Suche ein XML-Dokument, das gleich dem XML-Dokument duden.xml ist.	DOM-Interfaces (realisiert durch XML4J von IBM)	Exact Match
Ein-geschränkte Gleichheit	Überprüft, ob ein XML-Dokument gleich einem anderen XML-Dokument ist, wobei bestimmte Bestandteile eines XML-Dokumentes wie z.B. Kommentare oder PIs ignoriert werden.	Suche ein XML-Dokument, das gleich duden.xml ist, ohne Berücksichtigung der Kommentare.	DOM-Interfaces (realisiert durch XML4J von IBM)	Restricted Equality Match
AND-Verknüpfung	Realisiert eine AND-Verknüpfung mehrerer Matchingrelationen	Suche ein XML-Dokument, das einen XPath-Ausdruck X erfüllt und einer DTD Y folgt.	---	XML And Matchable
NOT-Verknüpfung	Realisiert eine NOT-Verknüpfung für eine Matchingrelation	Suche ein XML-Dokument, das nicht einer DTD X folgt.	---	XML Not Matchable
OR-Verknüpfung	Realisiert eine OR-Verknüpfung mehrerer Matchingrelationen	Suche ein XML-Dokument, das einen XPath-Ausdruck X erfüllt oder einer DTD Y folgt.	---	XML Or Matchable
XOR-Verknüpfung	Realisiert eine XOR-Verknüpfung mehrerer Matchingrelationen	Suche ein XML-Dokument, das einen XPath-Ausdruck X erfüllt oder einer DTD Y folgt, aber nicht beides.	---	XML XOr Matchable

Die genauen Angaben zum Hersteller (WWW-Adresse usw.) der verwendeten Software befinden sich im Softwarezeichnis im Anhang B.

7. Verteiltheit

Dieses Kapitel beschreibt, wie das Zusammenarbeiten räumlich verteilter Tuplespaces erreicht werden kann.

Kapitel 7.1. gibt einen kurzen Überblick über das Verteilungskonzept in XMLSpace, das die für den Zugriff auf ein entferntes Tuplespace notwendigen Abläufe in einem Distributor-Objekt kapselt. Das Kapitel 7.2. erläutert mögliche Verteilungsstrategien, nach denen die verteilten Tuplespaces zusammenarbeiten können und die durch das Distributor-Objekt umgesetzt werden. Das Kapitel 7.3. beschreibt schließlich, wie die Verteilung auf der Basis von TSpace implementiert werden kann.

7.1. Verteilungskonzept

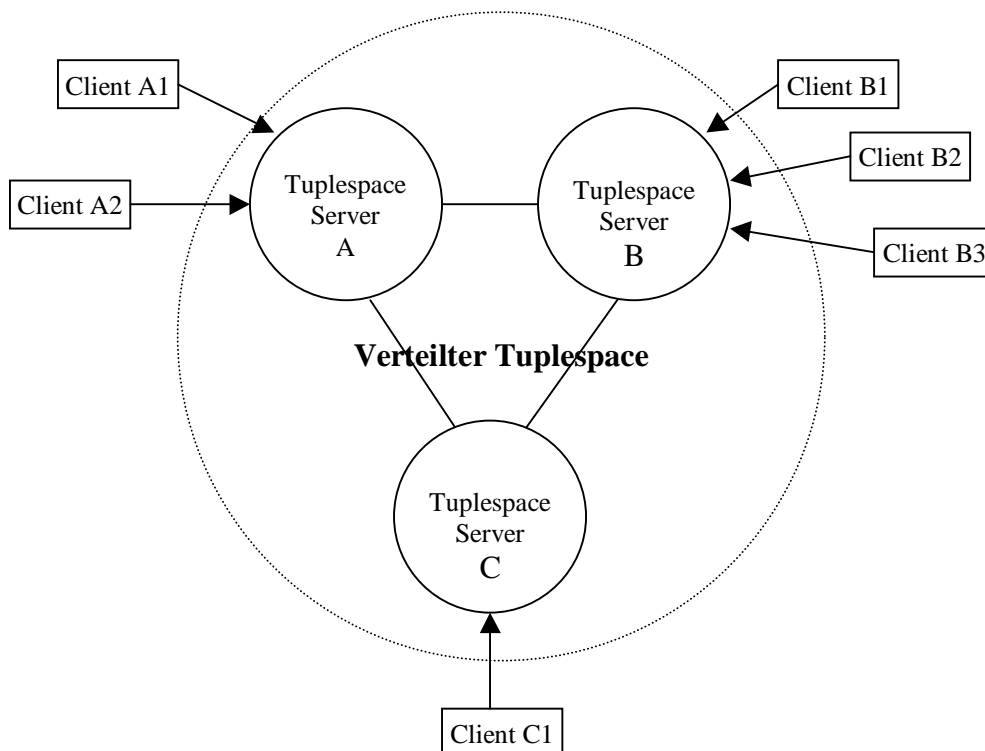


Abbildung 7.1.: Ein verteiltes Tuplespace

XMLSpace unterstützt das Zusammenarbeiten mehrerer räumlich verteilter Tuplespaces, die zusammen ein logisches Tuplespace bilden sollen. Das Prinzip ist in Abbildung 7.1. veranschaulicht. Hierbei ist ein Tuplespace-Server für das Verwalten eines lokalen Tuplespaces zuständig und die drei Tuplespace-Server A, B und C bilden zusammen ein verteiltes Tuplespace, das nach außen hin für die Clients wie ein einziges großes Tuplespace aussieht. Die drei Tuplespace-Server sind untereinander verbunden und können auf die Daten des anderen zugreifen. Auf diese Art kann zum Beispiel Client A1 auch auf Tuples zugreifen, die sich im Tuplespace auf dem Tuplespace-Server C befinden. Die Verteiltheit bleibt für den Client transparent. Client A1 führt einfach einen in()-Aufruf durch und weiß nicht, ob das Ergebnis vom lokalen Tuplespace-Server A oder von einem der entfernten Tuplespace-Server B oder C stammt.

Abbildung 7.2. zeigt wie in XMLSpace die Verbindung und Zusammenarbeit zwischen den einzelnen Tuplespace-Servern realisiert wird am Beispiel der Verbindung vom Tuplespace-Server A zum Tuplespace-Server B. Entsprechend sieht auch die umgekehrte

Verbindung vom TupleSpace-Server B zum TupleSpace-Server A und die restlichen Verbindungen zwischen den TupleSpace-Servern B und C und zwischen A und C aus.

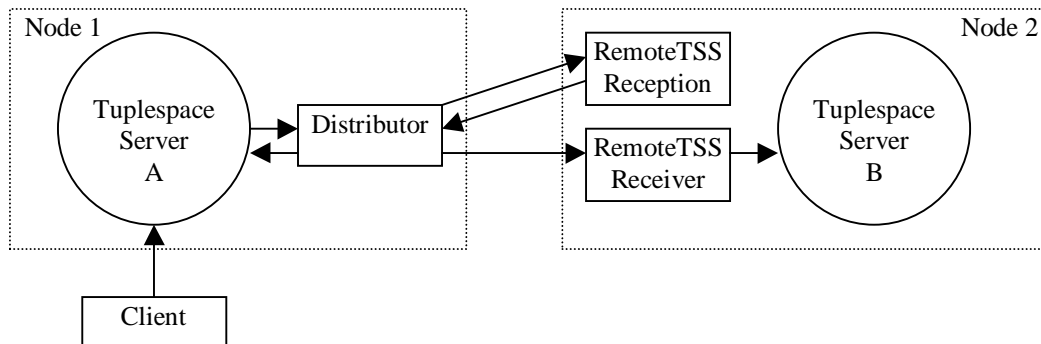


Abbildung 7.2.: Verbindung vom Server A zum Server B

Das Distributor-Objekt realisiert die Verteilung. Dazu gehört das An- und Abmelden der Server untereinander und das verteilte Ausführen der out()-, in()- und rd()-Aufrufe. Dabei folgt das Distributor-Objekt einer beim Serverstart festgelegten Verteilungsstrategie, die sicherstellt, daß die verteilten TupleSpaces zueinander konsistent bleiben (das nächste Kapitel 7.2. gibt einen Überblick über mögliche Verteilungsstrategien).

Alle TupleSpace-Server und damit das verteilte TupleSpace als ganzes verfolgen eine einheitliche, gemeinsame Verteilungsstrategie. Dies wird bei der Anmeldung der TupleSpace-Server untereinander dadurch sichergestellt, daß sich nur TupleSpace-Server untereinander anmelden und miteinander zusammenarbeiten können, die nach derselben Verteilungsstrategie arbeiten. Es ist also nicht möglich, daß z.B. TupleSpace-Server A mit einer Verteilungsstrategie ohne Replikation und TupleSpace-Server B mit einer Verteilungsstrategie mit voller Replikation arbeitet, weil dies zu Inkonsistenzen führen würde.

Beim Starten des TupleSpace-Servers A, meldet der Distributor des TupleSpace-Servers A den Server bei den anderen entfernten Servern an. Mit welchen Servern er Verbindung aufnehmen muß, ist je nach Verteilungsstrategie unterschiedlich. Der Aufbau der Verbindung zu einem entfernten TupleSpace-Server erfolgt über das RemoteTSSReception-Objekt des entfernten Servers. Das RemoteTSSReception-Objekt bietet eine Methode zum Anmelden und authentifizieren an und gibt bei erfolgreicher Authentifizierung eine Referenz auf das RemoteTSSReceiver-Objekt zurück, über das der Distributor auf das entfernte TupleSpace zugreifen kann.

Nachdem der Server A sich bei den anderen Servern angemeldet hat, ist der Server A in das verteilte TupleSpace integriert und bereit für das Ausführen der Client-Befehle. Die in-, out- und rd-Aufrufe des Clients werden vom TupleSpace-Server an den Distributor weitergeleitet, der je nach Verteilungsstrategie unterschiedlich damit umgeht. Wenn das verteilte TupleSpace zum Beispiel eine einfache Verteilungsstrategie ohne Replikation verfolgt, so fügt der Distributor bei einem out-Aufruf das Tuple einfach zum lokalen TupleSpace hinzu und sucht bei einem rd-Aufruf zunächst auf dem lokalen TupleSpace nach einem matchenden Tuple und wenn er lokal nichts findet, sucht er auf den entfernten TupleSpaces.

Der Zugriff auf ein entferntes TupleSpace erfolgt über das RemoteTSSReceiver-Objekt, auf das der Distributor beim Anmelden eine Referenz erhalten hat. Das RemoteTSSReceiver-Objekt bietet Methoden zum Hinzufügen, Suchen, Sperren und Löschen von Tuples an.

Das hier beschriebene Schema hat sich auf die Verbindung vom TupleSpace-Server A zum TupleSpace-Server B bezogen. Dem selben Schema entsprechend existiert auch eine

Verbindung vom TupleSpace-Server B nach A, und zwischen den TupleSpace-Servern A und C und zwischen B und C.

7.2. Verteilungsstrategien

Eine Verteilungsstrategie stellt sicher, daß die verschiedenen räumlichen verteilten TupleSpaces, die zusammen ein gemeinsames TupleSpace bilden, nach einem einheitlichen Verfahren arbeiten, das einen konsistenten Gesamtzustand des verteilten TupleSpaces garantiert. Die Verteilungsstrategie wird in XMLSpace von den Distributor-Objekten realisiert.

Zu den wichtigsten Verteilungsstrategien gehören: Verteilung ohne Replikation, Verteilung mit teilweiser Replikation, Verteilung mit voller Replikation und Verteilung über eine Hashingfunktion. Im folgenden wird ein kurzer Überblick über diese Verteilungsstrategien gegeben.

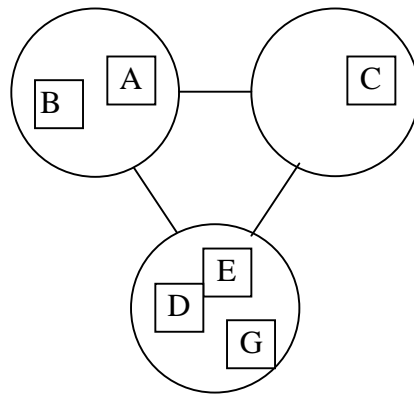


Abbildung 7.3.: Verteilung ohne Replikation

Bei einer Verteilung ohne Replikation [Ahuja86, Carriero86b] wird jedes Tuple genau auf einem Node gespeichert und ist netzwerkweit einmalig vorhanden. Abbildung 7.3. veranschaulicht dies. Hierbei entsprechen die Kreise den verteilten TupleSpaces und die Quadrate den verschiedenen Tuples. Jedes Tuple ist genau in einem Node gespeichert. Bei einem out-Aufruf wird das Tuple zum lokalen TupleSpace hinzugefügt. Bei einem in() und rd() muß, wenn nicht bereits lokal ein matchendes Tuple gefunden wird, netzwerkweit auf allen Nodes nach einem matchenden Tuple gesucht werden. Die Vorteile der Verteilung ohne Replikation sind: die geringen Kosten für das Hinzufügen von Tuples (weil es sich um eine lokale Operation auf das lokale TupleSpace handelt), die geringen Kosten für die Konsistenzerhaltung (weil jedes Tuple netzwerkweit einmalig ist und beim Löschen keine vorherige Abstimmung mit anderen Nodes stattfinden muß) und der geringere Speicherbedarf (weil jedes Tuple nur einmal gespeichert und nicht repliziert wird). Die Nachteile sind die hohen Kommunikationskosten für die Suche nach einem matchenden Tuple bei einem in() und rd() (netzwerkweite Operation) und die geringe Ausfallsicherheit (wenn ein Server ausfällt, sind seine Daten nicht mehr für die anderen TupleSpaces verfügbar).

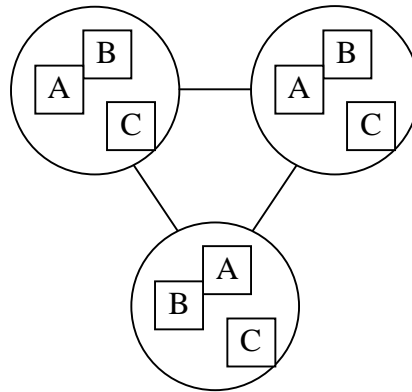


Abbildung 7.4.: Verteilung mit voller Replikation

Das komplette Gegenteil von Verteilung ohne Replikation bildet die Verteilung mit Voller Replikation [Carriero86b, Xu]. Hierbei wird bei einem Out das Tuple auf allen Nodes repliziert, so daß jeder Node eine komplette Kopie des gesamten Tuplespaces hat (Abbildung 7.4., wobei die gleich benannten Tuples die Replikate repräsentieren). Bei einem rd() oder in() muß nur lokal nach einem matchenden Tuple gesucht werden, weil jeder Node den kompletten Tuplespace enthält. Jedoch muß bei einem in() zunächst eine Sperre auf allen Nodes erlangt werden, ehe man das Tuple netzwerkweit Löschen und als Ergebnis zurückgeben darf. Der Vorteil der vollen Replikation liegt in den geringen Kosten für das Suchen nach einem matchenden Tuple bei rd() und in() und in der hohen Verfügbarkeit bzw. Ausfallsicherheit. Der Nachteil liegt in den hohen Kosten für das Hinzufügen eines Tuples (netzwerkweite Replikation auf allen Nodes), in den Kosten für die Konsistenzhaltung (in() erfordert zunächst ein netzwerkweites Sperren eines matchendes Tuples und ein anschließendes netzwerkweites Löschen bei erfolgreichem Sperren) und in dem hohen Speicherbedarf (jeder Node enthält eine komplette Kopie des gesamten verteilten Tuplespace).

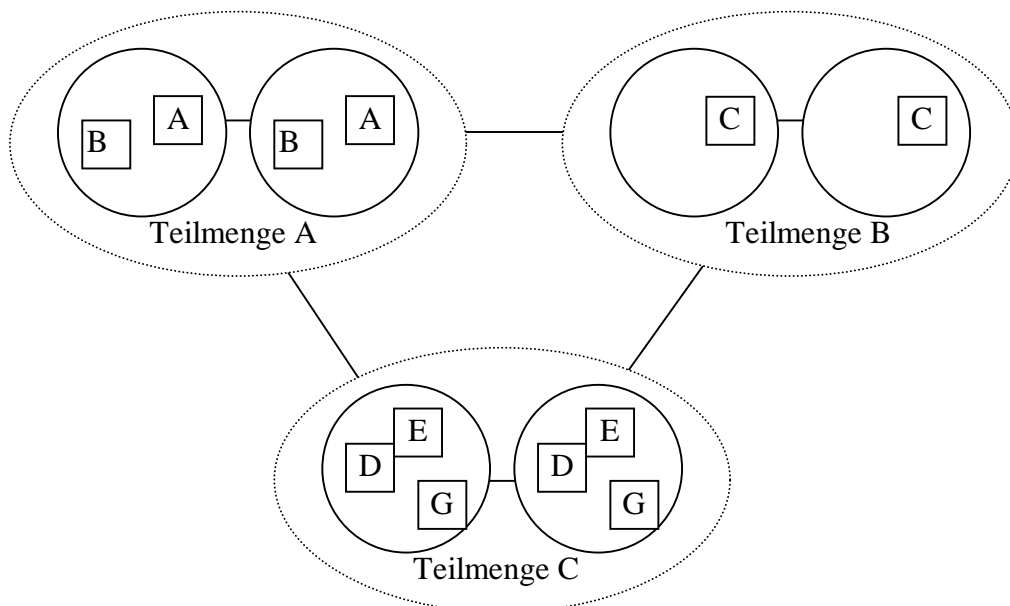


Abbildung 7.5.: Verteilung mit teilweiser Replikation

Teilweise Replikation [Faasen, Tolksdorf95] bildet einen Kompromiß aus den Vor- und Nachteilen von keiner und voller Replikation. Bei teilweiser Replikation wird das Tuple

nicht auf allen Nodes, sondern nur auf Teilmengen der Nodes repliziert. Angenommen die Nodes werden in drei disjunkte Teilmengen A, B und C unterteilt (Abbildung 7.5.). Ein Tuple wird nur innerhalb einer Teilmenge repliziert. Wenn also z.B. ein out auf einem Node in der Teilmenge A stattfindet, dann repliziert der Node dieses Tuple nur auf die anderen Nodes in der Teilmenge A. Bei einem in() und rd() muß nur auf einem Node pro Teilmenge A, B und C nach einem matchenden Tuple gesucht werden. Dies verringert die Kosten für die Suche nach einem matchenden Tuple gegenüber dem Verfahren ohne Replikation. Bei dem Löschen eines Tuples bei einem in()-Aufruf muß nur eine Sperre auf den Nodes in der Teilmenge, in der man das Tuple gefunden hat, erworben werden und auch nur auf diesen Nodes dieser Teilmenge das Tuple bei erfolgreicher Sperre gelöscht werden. Dies verringert die Kosten für die Konsistenzerhaltung gegenüber dem Verfahren mit voller Replikation.

Ein weiteres Verfahren, das vom Prinzip her zur Verteilung ohne Replikation gehört, ist die Verteilung mit einer Hashingfunktion [Ahuja86, Mattson92]. Hierbei wird wie bei dem Verfahren ohne Replikation jedes Tuple auf genau einem Node gespeichert. Jedoch wird hierbei anhand einer Hashingfunktion bestimmt, auf welchem Node das Tuple abgespeichert und auf welchem Node ein Tuple gesucht wird. Bei einem out() wird durch die Anwendung der Hashingfunktion auf die Tuplefelder bestimmt, zu welchem Node das Tuple hinzugefügt wird. Ebenso wird bei einem in() oder rd() durch die Anwendung der Hashingfunktion auf die Templatefelder bestimmt, auf welchem Node ein matchendes Tuple zu finden ist. Die Hashingfunktion muß für Template- und Tuplefelder, die miteinander matchen, die gleichen Ergebnisse liefern. Da es sich hier um keine Replikation handelt hat das Hashingverfahren zunächst die gleichen Vorteile wie das Verfahren ohne Replikation: geringer Speicherbedarf und geringe Kosten für die Konsistenzerhaltung. Ein weiterer Vorteil liegt gegenüber der einfachen Verteilung ohne Replikation jedoch darin, daß bei einem in() oder rd() jeweils immer nur ein Node gefragt werden muß und nicht wie bei dem Verfahren ohne Replikation *alle* entfernten Nodes. Der wesentliche Nachteil des Hashingverfahrens ist, daß es nicht geeignet ist, wenn das Hinzufügen und Entfernen neuer Server-Nodes zur Laufzeit unterstützt werden soll. Das Hashingverfahren geht von einer statischen Menge von Nodes aus, die sich zur Laufzeit nicht ändert. Wenn neue Nodes hinzugefügt werden, müßte die Hashingfunktion angepasst werden und evtl. bereits verteilte Tuples auf andere Nodes verschoben werden, um der neuen Hashingfunktion zu genügen. Dies könnte zu einer kompletten Reorganisation mit enormen Kosten führen. Daher ist die Verwendung des Hashingverfahrens mit einer sich zur Laufzeit ändernden Netzwerkstruktur nicht geeignet.

XMLSpace unterstützt zur Zeit die Verteilungsstrategien ohne Replikation und mit teilweiser Replikation. Das Hashingverfahren scheidet für XMLSpace aus, weil es eine statische Struktur verlangt und nicht das Hinzufügen und Entfernen von Servern zur Laufzeit erlaubt. Dies ist für XMLSpace nicht akzeptabel, weil XMLSpace (insbesondere auch im Hinblick auf Workspaces) Unterstützung für sehr langlebige Anwendungen bieten soll, bei denen sich zur Laufzeit Anforderungen ergeben können, die es erforderlich machen die Struktur des verteilten Tuplespace dynamisch zur Laufzeit durch das Hinzufügen oder Entfernen von Servern zu ändern. Verteilung mit voller Replikation ist nicht geeignet, weil die Kosten für die Replikation und Konsistenzerhaltung zu hoch sind. Volle Replikation eignet sich nur, wenn sehr wenige out()- und in()-Operationen und sehr viele rd()-Operationen auf dem verteilten Tuplespace stattfinden. Im Kontext von Workspaces ist jedoch eher zu erwarten, daß viele out()- und in()-Operationen und gleichviel oder weniger rd()-Operationen stattfinden.

Gegenüber dem Hashingverfahren erlaubt die Verteilung ohne und mit teilweiser Replikation das An- und Abmelden neuer Server zur Laufzeit und gegenüber voller Replikation sind die Kosten für die Konsistenzerhaltung wesentlich geringer. Daher werden in XMLSpace Verteilung ohne und mit teilweiser Replikation realisiert.

Die nächsten beiden Kapitel beschreiben die Verteilungsstrategien mit teilweiser (Kapitel 7.2.1.) und ohne Replikation (Kapitel 7.2.2.) im Detail. Bei den dort beschriebenen Verfahren wird die Voraussetzung gemacht, daß man es mit ausfallsicheren Servern und Netzwerken zu tun hat und die Server sich ordnungsgemäß untereinander an- und abmelden. Fehler durch Ausfälle von Server, Netzwerkpartitionierungen, Duplizieren und Verlieren von Messages werden von diesen Verfahren nicht behandelt. Eine mögliche Lösung für das Abfangen und Behandeln solcher Fehler ist in [Xu] beschrieben.

7.2.1. Teilweise Replikation

Die hier gemachten Beschreibungen zur teilweisen Replikation basieren auf den Ausführungen in den Papieren [Faasen, Tolksdorf95, Gelernter85]. Das Protokoll zum Ablauf von in()-, out()- und rd() orientiert sich an [Faasen, Tolksdorf95] und das Protokoll zum An- und Abmelden von Nodes an [Tolksdorf95]. Der wesentliche Unterschied besteht darin, daß hier nicht von Busstrukturen ausgegangen wird, sondern von direkten Punkt zu Punkt Verbindungen zwischen den einzelnen Nodes.

7.2.1.1. Gitterstruktur

Bei teilweiser Replikation werden die Tuples nur auf eine Teilmenge aller Nodes repliziert. Die Nodes bilden hierbei sogenannte In- und Out-Sets, wobei jeder Node zu *genau einem* In- und zu *genau einem* Out-Set gehört. Die Nodes werden in einer Gitterstruktur angeordnet, bei der jeder In-Set jeden Out-Set schneidet und umgekehrt.

Abbildung 7.6 zeigt die Anordnung der Nodes bei teilweiser Replikation als Gitterstruktur. In der Abbildung repräsentieren die waagerechten Verbindungslinien die Out-Sets und die senkrechten Verbindungslinien die In-Sets.

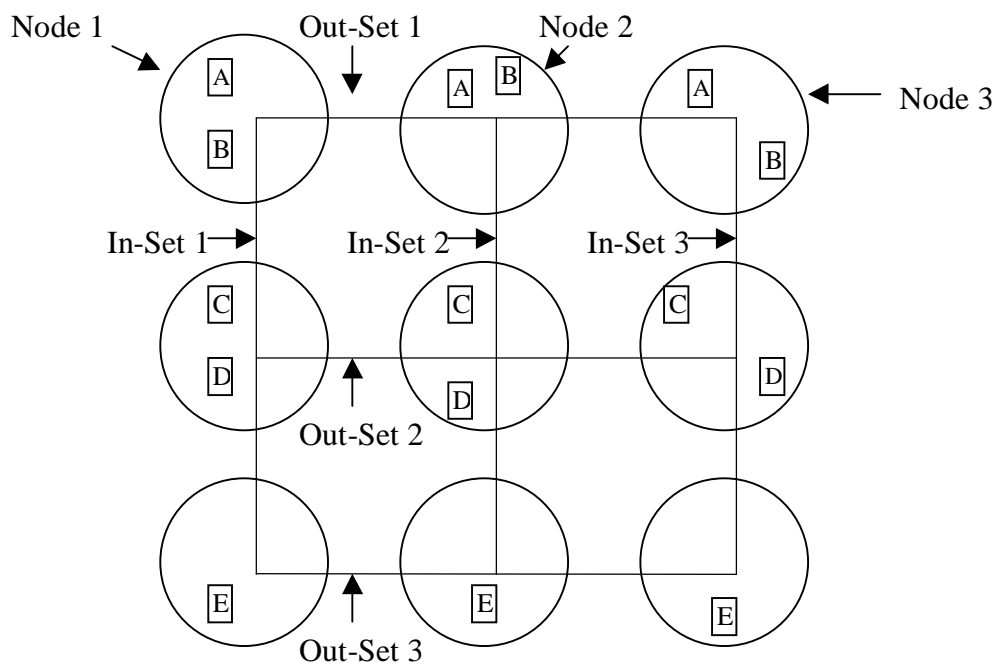


Abbildung 7.6.: Anordnung der Nodes in einer Gitterstruktur
(in leicht abgewandelter Form aus [Tolksdorf95])

Jeder Node in einem In- oder Out-Set ist mit jedem anderen Node in diesem In- oder Out-Set verbunden, d.h. daß z.B. bei dem Out-Set 1 der Node 1 mit dem Node 2, aber auch mit dem Node 3 verbunden ist und umgekehrt. [Tolksdorf95] beschreibt die Verbindung zwischen den einzelnen Nodes eines Sets über eine Busstruktur. In XMLSpace handelt es sich

dagegen um direkte Punkt zu Punkt Verbindung (Abbildung 7.7). Zur übersichtlicheren Darstellung wird weiterhin die in Abbildung 7.6. verwendete Gitterstruktur verwendet, auch wenn die tatsächlichen Verbindungen keine Busstrukturen sind, wie man vielleicht annehmen könnte, sondern den in Abbildung 7.7. dargestellten Verbindungen entsprechen.

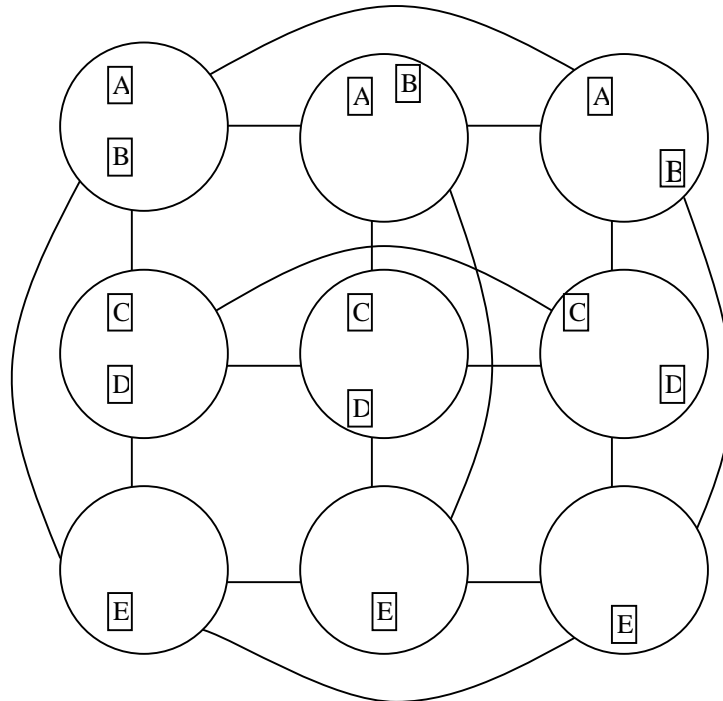


Abbildung 7.7.: Die Darstellung der Gitterstruktur als Punkt-zu-Punkt-Verbindungen

Einzufügende Tuples werden über den Out-Set, zu dem der Node gehört, auf dem das out() stattfindet, repliziert, so daß jeder Node in einem Out-Set denselben Inhalt hat.

Durch die Anordnung der In- und Out-Sets in der Gitterstruktur, bei der jeder In-Set jeden Out-Set schneidet, enthält jeder In-Set den kompletten Tuplespace. Damit hat jeder Node über den In-Set, zu dem er gehört, Zugriff auf alle Elemente im Netzwerk, da die Vereinigung der Elemente auf allen Nodes in einem In-Set den momentanen Inhalt des verteilten Tuplespace ergeben. Zum Beispiel enthält in der Abbildung 7.6. jeder In-Set als Vereinigung die Tuples A, B, C, D und E.

Für die Suche nach einem matchenden Tuple bei in() und rd() reicht es daher aus nur auf den Nodes in dem In-Set zu suchen, zu dem der Node gehört, auf dem das in() oder rd() aufgerufen wurde, weil der In-Set den Inhalt des gesamten Tuplespaces widerspiegelt.

7.2.1.2. In-Protokoll

Bei einem In()-Aufruf auf dem Node A wird zunächst auf dem lokalen Tuplespace nach einem matchenden, nicht gesperrten Tuple gesucht (wenn es bereits lokal gesperrt ist, versucht jemand anderes dieses Tuple zu löschen). Wenn lokal ein matchendes Tuple gefunden wurde, muß es auf allen Nodes im Out-Set gelöscht werden, bevor es als Ergebnis auf den In()-Aufruf zurückgegeben werden kann. Dazu muß das Tuple zunächst auf allen Nodes im Out-Set erfolgreich gesperrt werden, weil es vorkommen kann, daß mehrere Nodes im Out-Set gleichzeitig ein in() auf dasselbe Tuple ausführen wollen und nur ein Node erfolgreich sein darf.

Das Tuple wird zuerst lokal gesperrt und anschließend wird versucht die Replikate des Tuples auf allen Nodes im Out-Set zu sperren. Wenn mehrere Nodes im Out-Set dasselbe Tuple im Out-Set sperren wollen, wird durch eine Sperrstrategie bestimmt, welcher Node die Sperre erfolgreich im gesamten Out-Set setzen darf. Möglich Sperrstrategien sind unter anderem:

- Mehrheitsprinzip: Es wird versucht auf so vielen Nodes im Out-Set wie möglich eine Sperre zu setzen. Wenn man nicht alle Sperren erhalten hat, aber auf der Mehrheit der Nodes im Out-Set eine Sperre erfolgreich setzen konnte, kann man die gesetzten Sperren behalten und versucht auf den restlichen Nodes, auf denen man noch keine Sperre setzen konnte, erneut eine Sperre zu setzen, bis man auf allen Nodes erfolgreich eine Sperre setzen konnte. Wenn man beim Sperren nicht auf den Mehrheit der Nodes eine Sperre erfolgreich setzen konnte, muß man sämtliche Sperren wieder freigeben, damit der andere Node, der bereits eine Mehrheit der Sperren besitzt auch die restlichen Sperren erlangen kann. Es gewinnt also derjenige, der beim ersten Sperrversuch auf den meisten Nodes erfolgreich eine Sperre setzen konnte.
- Prioritätsprinzip: Jeder Node hat eine eindeutige Prioritätsnummer. Wenn zwei Server eine Sperre auf das selbe Tuple setzen wollen, dann gewinnt derjenige, der die höchste Prioritätsnummer hat. Dies ist ein ungerechtes Verfahren, weil die Server mit höheren Prioritätsnummern bevorzugt werden.

Wenn das Tuple im gesamten Out-Set erfolgreich gesperrt werden konnte, kann das gesperrte Tuple auf den einzelnen Nodes im Out-Set gelöscht und als Ergebnis zurückgegeben werden. Der In()-Aufruf konnte damit gleich über den lokalen Tuplespace erfüllt werden.

Wenn das Sperren im Out-Set nicht möglich ist, werden die bereits gesetzten Sperren wieder freigegeben und lokal nach einem anderen noch nicht gesperrten Tuple gesucht und beim Finden eines weiteren matchenden Tuples nach dem eben beschriebenen Verfahren vorgegangen.

Wenn lokal kein weiteres matchendes, noch nicht gesperrtes Tuple gefunden wird, werden die einzelnen Nodes im In-Set nach einem matchenden Tuple durchsucht. Hierzu wird auf den einzelnen Nodes im In-Set der Reihe nach die Methode findAndDelete(template) aufgerufen. Die Methode findAndDelete(template) sucht auf dem Tuplespace des Node, auf dem die Methode aufgerufen wurde, nach einem matchenden Tuple, versucht es in dessen Out-Set zu sperren und löscht es bei erfolgreicher Sperre. Wenn ein Node im In-Set erfolgreich ein matchendes Tuple sperren und löschen konnte, gibt er dies als Ergebnis zurück und der In()-Aufruf in beendet. Ansonsten signalisiert er, daß er nicht erfolgreich war und es wird auf dem nächsten Node im In-Set nach einem matchenden Tuple gesucht.

Wenn weder lokal noch entfernt erfolgreich ein matchendes Tuple gefunden, gesperrt und gelöscht werden konnte, wird die In-Anfrage zu einem lokalen Request-Pool hinzugefügt. Der Request-Pool enthält sämtliche lokalen In()- und Rd()-Anfragen, die nicht erfüllt werden konnten. Die im Request-Pool abgelegten Anfragen werden in gewissen Zeitabständen von einem Request-Pool-Checker erneut lokal und auf allen Nodes im In-Set nach dem hier beschriebenen Verfahren durchgeführt, bis schließlich ein Ergebnis gefunden wird. Erfüllte Anfragen werden aus dem Request-Pool entfernt.

7.2.1.3. Rd-Protokoll

Der Ablauf von Rd ist ähnlich dem Ablauf von In, nur daß das gefundene matchende Tuple vorher nicht im Out-Set gesperrt und gelöscht werden muß.

Zunächst wird im lokalen Tuplespace nach einem matchenden Tuple gesucht. Hierbei ist es im Unterschied zu in() egal, ob das Tuple gesperrt ist oder nicht, weil Rd() nur lesend auf das Tuple zugreift und es nicht löscht.

Wenn ein matchendes Tuple gefunden wird, kann eine Kopie dieses Tuples als Ergebnis zurückgegeben werden.

Wenn lokal kein matchendes Tuple gefunden wurde, werden die Nodes im In-Set der Reihe nach nach einen matchenden Tuple durchsucht. Dies erfolgt durch den Aufruf der Methode `find(template)` auf dem jeweiligen Node im In-Set. Die Methode `find(template)` sucht auf dem Node im In-Set nach einem matchenden Tuple. Wenn die Methode ein matchendes Tuple findet (egal ob gesperrt oder nicht), gibt sie eine Kopie des Tuples als Ergebnis zurück. Wenn sie kein matchendes Tuple findet, signalisiert sie dies.

Wenn entfernt ein matchendes Tuple gefunden wurde, wird dieses als Ergebnis zurückgegeben und `Rd()` ist beendet.

Wenn weder lokal noch entfernt ein matchendes Tuple gefunden wurde, wird die `Rd`-Anfrage wie auch bei `In()` zum lokalen Request-Pool hinzugefügt. Der Umgang mit der `Rd`-Anfrage im Request-Pool ist derselbe wie bei einer `In()`-Anfrage, d.h. der Request-Pool-Checker führt die `Rd`-Anfrage in gewissen Zeitabständen erneut lokal und auf allen Nodes im In-Set durch.

7.2.1.4. Out-Protokoll

Bei einem `Out`-Aufruf auf den Node A wird zunächst überprüft, ob das hinzuzufügende Tuple wartende lokale `In()`- oder `Rd()`-Anfragen erfüllt. Dazu wird der lokale Request-Pool durchlaufen, der die wartenden `In()`- und `Rd()`-Anfragen enthält, und überprüft, ob das in der `In()`- oder `Rd()`-Anfrage angegebene Template mit dem hinzuzufügenden Tuple matcht. Dabei kann das Tuple beliebig viele wartende `Rd()`-Befehle erfüllen, aber höchstens einen `In()`-Befehl, weil `In()`-Befehle konsumierend sind. Es wird also der Request-Pool durchlaufen und dabei alle matchenden `Rd()`-Anfragen erfüllt, bis auf eine matchende `In()`-Anfrage gestoßen wird oder der Request-Pool zu Ende durchlaufen ist. Die gefundenen matchenden `In()`- oder `Rd()`-Befehle werden erfüllt und die erfüllten Anfragen aus dem Request-Pool entfernt.

Wenn ein lokaler wartender `In()`-Befehl erfüllt wurde, ist das Tuple gleich lokal konsumiert und braucht nicht im lokalen Tuplespace abgelegt oder an die anderen Nodes im `Out`-Set weitergeschickt zu werden. Dies spart Kommunikationskosten.

Wenn das Tuple nur mit `Rd`-Anfragen oder mit gar keiner wartenden Anfrage gematcht hat, wird das Tuple zum lokalen Node hinzugefügt und auf allen Nodes des `Out`-Sets repliziert. Hierzu wird bei allen Nodes im `Out`-Set die Methode `add(t,id)` aufgerufen, wobei `t` das zu replizierende Tuple und `id` der eindeutige Identifier ist, unter dem das Tuple abgelegt werden soll. Bei dem Identifier handelt es sich um eine netzwerkweit global eindeutige Tuple-Id. Alle Replikate haben dieselbe Tuple-Id. Unter der Annahme, daß die Netzwerkadresse eines Nodes eindeutig ist, ist, kann die global eindeutige Id aus der Netzwerkadresse und einem aufsteigenden Zähler erzeugt werden (durch die Angabe dieser Tuple-Id wird beim Sperren und Löschen bei einem `In()`-Aufruf sichergestellt, daß auf allen Nodes im `Out`-Set das gleiche Tuple gelöscht wird).

Die Nodes im `Out`-Set fügen das Tuple einfach zu ihrem Tuplespace hinzu.

7.2.1.5. Anmelden

7.2.1.5.1. Simulierte Nodes

Die Verfahren für das An- und Abmelden (Kapitel 7.2.1.6) von Nodes stellen die Gitterstruktur sicher, die für die in den Kapitel 7.2.1.2. bis 7.2.1.4. beschriebenen Abläufe der `in`, `out` und `rd`-Methoden notwendig sind. Für das Aufrecht erhalten der Gitterstruktur sind $n*m$ Nodes notwendig, wenn man n In-Sets und m Out-Sets hat. Es gibt jedoch Situationen, in denen nicht immer $n*m$ Nodes verfügbar sind. Angenommen man hat eine Gitterstruktur aus 2 In-Sets und 3 Out-Sets und es soll ein neuer Node zu dieser Struktur hinzugefügt

werden, indem ein neuer In-Set gebildet wird, zu dem dieser neue Node hinzugefügt werden kann. Man hätte also nach dem Einfügen 3 In-Sets und 3 Out-Sets, aber nur 7 Nodes.

Das Aufrecht erhalten der Gitterstruktur ist durch die Verwendung simulierter Nodes möglich [Tolksdorf95]. Abbildung 7.8. zeigt das Hinzufügen eines neuen Nodes durch das Hinzufügen eines neuen In-Sets unter Verwendung von simulierten Nodes. Hierbei stehen N1 bis N7 für die echten Nodes, wobei der dick umrahmte Node N7 der neue Node ist, und S1 und S2 stehen für die simulierten Nodes:

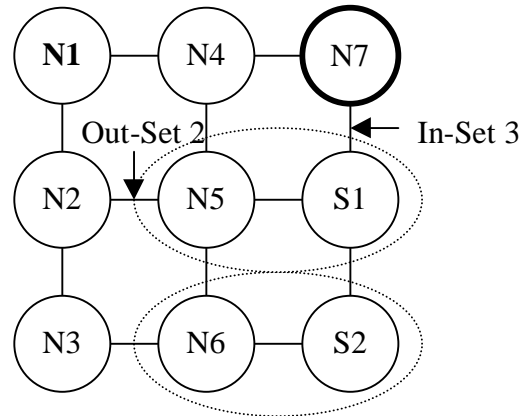


Abbildung 7.8.: Verwendung von simulierten Knoten

Der neu hinzugefügte Node bildet in dem Beispiel einen neuen In-Set 3. Die restlichen Nodes im In-Set bilden die simulierten Nodes S1 und S2, weil es nicht genügend echte Nodes gibt. Ein simulierter Node wird immer von einem Node in dem Out-Set simuliert, zu dem der simulierte Node gehört. Zum Beispiel könnte S1 von N5 und S2 von N6 simuliert werden. Dies ist in der Abbildung durch die gestrichelte Umrandung von N5 und S1 und von N6 und S2 angedeutet. N5 und S1 sind also eigentlich ein und derselbe Node. Ebenso N6 und S2. Auf diese Art entstehen keine Mehrkosten beim Replizieren und der Node der einen anderen Node simuliert muß zusätzlich zu den Anfragen in seinem eigenen In-Set die Anfragen des In-Sets, zu dem der simulierte Node gehört, behandeln. Zum Beispiel muß bei einem Out auf dem Out-Set 2 das Tuple nur auf dem Node N2 und N5 repliziert werden. S1 hat automatisch die replizierten Daten, weil S1 gleich N5 ist. Bei einem In()- oder Rd()-Aufruf im Node N7, greift der Node N7 auf S1 und S2 zu und damit eigentlich auf N5 und N6. Die folgende Abbildung 7.9. veranschaulicht noch einmal die tatsächliche (reale) Verbindung von N7 mit N5 und N6.

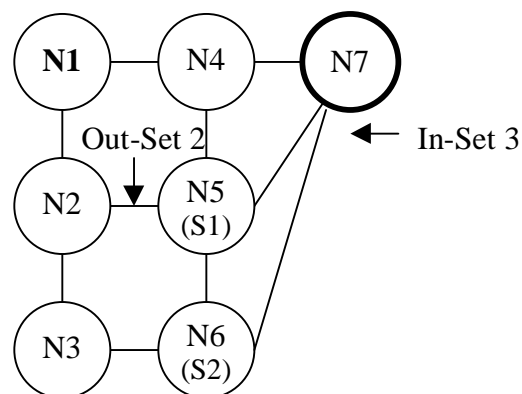


Abbildung 7.9.: Die tatsächliche Verbindung ohne Darstellung mit simulierten Knoten

7.2.1.5.2. Anmeldeverfahren

Die Anmeldung eines neuen Nodes erfolgt über einen besonderen Node in der bereits vorhandenen Gitterstruktur, dem sogenannten Rezeptionisten (Abbildung 7.10., hierbei steht N für den neuen Node und R für den Rezeptionisten).

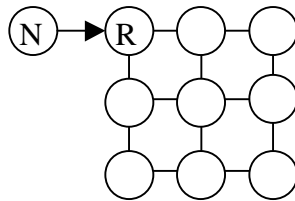


Abbildung 7.10.: Anmeldung über den Rezeptionisten

Der Rezeptionist behandelt immer nur eine An- oder eine Abmeldung, aber nie mehrere An- oder Abmeldungen gleichzeitig oder durcheinander. Alle anderen An- oder Abmeldeanfragen werden solange verzögert oder abgelehnt, bis die laufende Anmeldung beendet ist.

Der Rezeptionist enthält Informationen über die gesamte Gitterstruktur, die er jedes Mal beim An- und Abmelden von Nodes aktualisiert. Anhand dieser Informationen entscheidet der Rezeptionist, wo der neue Node hinzugefügt wird. Diese Entscheidung kann nach sehr unterschiedlichen Kriterien getroffen werden. Zum Beispiel könnte der Rezeptionist neue Nodes immer so hinzufügen, daß sie räumlich möglichst nah mit den anderen Nodes in ihrem Out-Set liegen, damit die Kommunikationskosten für die Replikation und Konsistenzerhaltung möglichst gering gehalten werden. Oder der Rezeptionist versucht so wenig wie möglich simulierte Nodes in der Gitterstruktur zu haben und daß es möglichst gleich viele In- und Out-Sets gibt.

Es gibt vom Prinzip her nur drei Möglichkeiten, wie der neue Node zu der vorhandenen Gitterstruktur hinzugefügt werden kann:

1. Der neue Node nimmt die Stelle eines simulierten Nodes in der Gitterstruktur ein.
2. Es wird ein neuer In-Set gebildet und der neue Node zu diesem In-Set hinzugefügt.
3. Der neue Node bildet einen neuen Out-Set.

Diese drei Möglichkeiten werden im Folgenden beschrieben.

Angenommen der neue Node N soll in der Gitterstruktur den simulierten Node S ersetzen, der von dem Node A simuliert wird. Dies ist in Abbildung 7.11. dargestellt.

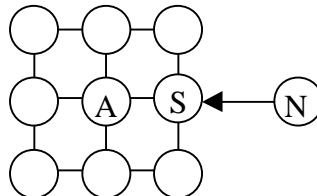


Abbildung 7.11.: Hinzufügen eines Knoten an Stelle eines simulierten Knotens

Der neue Node N wird schrittweise in der Gitterstruktur integriert. Dazu muß der neue Node sich nach und nach mit dem Node A abstimmen, so daß der neue Node N am Ende den gleichen Zustand wie die anderen Nodes im Out-Set des Nodes A hat. Während der Abstimmungsphase simuliert weiterhin der Node A den Node S. Der neue Node N ist nur für den Node A sichtbar, aber für niemanden sonst in der Gitterstruktur, d.h. der Node N ist noch nicht in einem In- oder Out-Set integriert.

Die Abstimmung beinhaltet, daß sämtliche Tuples und die gesetzten Sperren schrittweise auf den Node N übertragen werden. Dabei bleibt die Funktionalität der anderen Nodes in der Gitterstruktur weitestgehend uneingeschränkt.

Node A überträgt portionsweise die Daten an den Node N. Befehlsaufrufe vom Node A selbst oder von außen (von einem Node in seinem In- oder Out-Set), die sich auf Daten im Node A beziehen, die *gerade zur Zeit* auf den Node N übertragen werden, werden vom Node A verzögert, bis diese Datenportion komplett übertragen wurde. Erst dann dürfen diese Befehle ausgeführt werden, wobei dieser Befehl gleichzeitig von dem Node A an den Node N weitergeleitet wird, so daß dieser konsistent bleiben kann. Ebenso werden sämtliche Befehlsaufrufe, die sich auf bereits übertragene Daten beziehen, sowohl auf dem Node A als auch auf dem Node N ausgeführt. Auf diese Art bleibt der Datenzustand der bereits übertragenen Daten im Node N konsistent mit den Daten auf dem Node A. Wenn zum Beispiel Node A2 auf A ein Tuple t3 sperren will, das bereits auf den Node N übertragen wurde, so leitet der Node A diesen Sperraufruf auch an den Node N weiter, so daß das Tuple sowohl auf dem Node A als auch auf dem Node N gesperrt wird (Abbildung 7.12.). Bei einem Löschversuch des Nodes A3 auf ein Tuple t4 im Node A, das gerade auf den Node N übertragen wird, wird der Löschversuch solange verzögert, bis die Übertragung des Datenblocks, zu dem das zu löschende Tuple gehört, beendet ist. Erst dann wird das Tuple im Node A gelöscht und der Löschaufruf direkt an den Node N weitergeleitet, so daß auch der Node N das Tuple löscht. Ebenso werden Out-Aufrufe auf dem Node A direkt an den Node N weitergeleitet und das hinzugefügte Tuple als bereits übertragen gekennzeichnet.

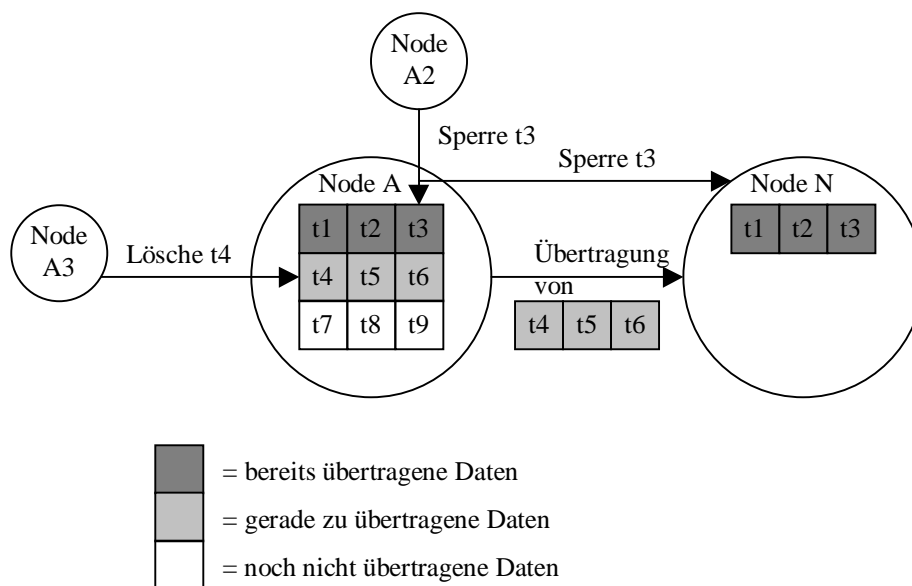


Abbildung 7.12.: Abstimmung von Node N mit Node A

Nachdem die Abstimmung des Nodes N mit dem Node A komplett ist, haben beide die gleichen Daten. Trotzdem leitet Node A weiterhin die Befehlsaufrufe an den Node N weiter, bis dieser komplett integriert ist im Out-Set.

Der Node N veranlasst alle Nodes im Out-Set alle Operationen, die den Out-Set als ganzes Betreffen anzuhalten. Dazu gehört das Sperren, Löschen und Hinzufügen von Tuples im Out-Set. Alle noch laufenden Operationen werden beendet, aber keine neuen Operationen gestartet. Wenn alle Nodes im Out-Set die noch laufenden Operationen beendet haben, teilen sie dies dem neuen Node mit. Wenn der Node von allen Nodes im Out-Set die Bestätigung erhalten hat, daß zur Zeit keine Operationen im Out-Set erfolgen, ist sichergestellt, daß alle

Nodes im Out-Set denselben Zustand haben. Der neue Node kann sich jetzt bei den anderen Nodes im Out-Set registrieren, so daß die anderen Nodes ihn bei den nächsten Aufrufen, die den Out-Set betreffen, mit einbeziehen und der neue Node N mit dem Out-Set im konsistenten Zustand bleibt. Der neue Node ist jetzt voll im Out-Set integriert und bekommt sämtliche folgenden Änderungen am Out-Set automatisch mit. Nachdem die Verbindungen zu den anderen Nodes im Out-Set aufgebaut wurden, teilt der neue Node N den Nodes im Out-Set mit, daß sie wieder Operationen im Out-Set ausführen können. Der Out-Set kann jetzt normal weiterarbeiten.

Zur vollständigen Integration in der Gitterstruktur fehlt nur noch die Intergration des Nodes N in seinem In-Set. Dazu registriert sich der neue Node N bei den einzelnen Nodes im In-Set, so daß die Nodes im In-Set ihre nächsten Aufrufe nicht mehr an den simulierten Node, sondern an den neuen Node N schicken. Der neue Node ist damit im In-Set voll funktionsfähig.

Der neue Node N ist voll integriert in der Gitterstruktur und kann seinen Betrieb aufnehmen.

Das eben beschriebene Verfahren bezog sich darauf, wenn der neue Node einen simulierten Node ersetzt. Ähnlich läuft es ab, wenn der neue Node einen neuen In-Set bilden soll. Zunächst bestimmt der Rezeptionist für jeden Node im Out-Set einen Node, der einen simulierten Node für den neuen In-Set bilden soll. Angenommen der Rezeptionist entscheidet, daß A1 A2 und A3 die jeweiligen Nodes S1, S2 und S3 im neuen In-Set simulieren sollen. Weiterhin entscheidet der Rezeptionist, daß der neue Node N an der Stelle S1 eingefügt werden soll (Abbildung 7.13.). Der restliche Ablauf ist genau derselbe wie zuvor beschrieben: Der neue Node N stimmt sich mit dem Node A1 ab, bis beide denselben Zustand haben. Dann wird der Out-Set kurz angehalten, so daß alle Nodes im Out-Set für einen Moment den gleichen Zustand haben. Der neue Node meldet sich bei den einzelnen Nodes im Out-Set an, zu dem er gehört, und ist im Out-Set integriert. Der Out-Set kann jetzt normal weiterarbeiten. Anschließend baut er die Verbindung zu den Nodes S2 und S3, die durch A2 und A3 simuliert werden auf und ist damit komplett im In-Set integriert. Damit ist der Node N vollständig in der Gitterstruktur eingegliedert und kann seine Funktion aufnehmen.

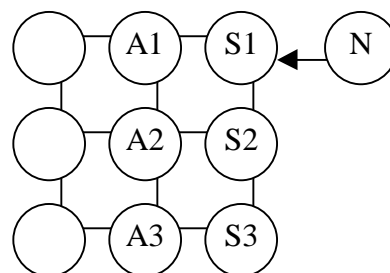


Abbildung 7.13.: Das Hinzufügen eines neuen Knotens in einem neuen In-Set.

Die letzte Möglichkeit der Integration eines neuen Nodes N in der Gitterstruktur ist, daß der neue Node N einen neuen Out-Set bildet. Hierbei fällt die Abstimmung mit anderen Nodes weg, weil der Node N allein im Out-Set ist. Der Node N simuliert die restlichen Nodes im Out-Set. Der Node N meldet sich bei den Nodes in seinem In-Set und bei allen anderen In-Sets, für die er einen Node simuliert, an, so daß die Nodes in den In-Sets ihn bei ihren nächsten Anfragen mit einbeziehen. Der neue Node N ist damit im In-Set und vollständig in der Gitterstruktur integriert und kann seinen Betrieb aufnehmen (Abbildung 7.14).

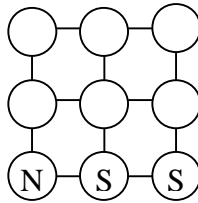


Abbildung 7.14.: Das Hinzufügen eines neuen Knotens als einen neuen Out-Set

Bei dem hier beschriebenen Verfahren wird davon ausgegangen, daß der neu hinzugefügte Node leer ist. Wenn der neue Node jedoch bereits Daten beinhaltet, dann dürfen diese solange nicht für die anderen Nodes in der Gitterstruktur sichtbar sein, bis der Node komplett integriert ist. Dies ist zum Beispiel machbar, indem die Tuples in einem rein lokalen Tuplespace zwischengespeichert werden, das nicht für die anderen Nodes in der Gitterstruktur sichtbar ist. Erst wenn der neue Node voll integriert ist kann er seine eigenen Daten über normale Out-Aufrufe zu dem verteilten Tuplespace hinzufügen.

7.2.1.6. Abmelden

Wie auch beim Anmelden wird beim Abmelden die für die in()-, out()- und rd()-Aufrufe notwendige Gitterstruktur aufrecht erhalten.

Beim Abmelden gibt es im Wesentlichen zwei Varianten:

1. Es handelt sich bei dem abmeldenden Node um den letzten echten Node im Out-Set, d.h. der Out-Set enthält nur noch den sich abmeldenden Node und evtl. von ihm simulierte Nodes (Abbildung 7.15).
2. Es handelt sich um das Abmelden eines Nodes innerhalb eines Out-Sets, der noch weitere echte, nicht simulierte Nodes enthält (Abbildung 7.16)

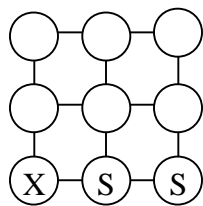


Abbildung 7.15.: Das Abmelden des letzten *echten* Nodes in einem Out-Set

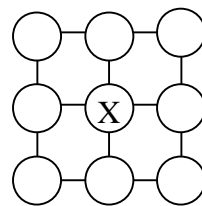


Abbildung 7.16.: Das Abmelden eines Nodes in einem Out-Set, der noch weitere *echte* Nodes enthält.

Das Abmelden des Nodes muß generell verzögert werden, wenn lokale Clients noch laufende Befehle auf dem abzumeldenden Node haben, zum Beispiel wartende In()- oder Rd()-Anfragen im Request-Pool. Neue Client-Aufrufe werden vom abmeldenden Node nicht mehr entgegen genommen und nur die noch laufenden abgearbeitet.

Anschließend wendet sich der abzumeldende Node an den Rezeptionisten. Wenn es neben dem abzumeldenden Node noch weitere echte Nodes im Out-Set gibt, entscheidet der Rezeptionist, welche anderen Nodes im Out-Set den abzumeldenden Node und sämtliche von ihm simulierte Nodes nach seinem Wegfall simulieren. Es müssen hierbei keine Zustände vom wegfallenden auf den anschließend simulierenden Node übertragen werden, da der simulierende Node, da er zum selben Out-Set gehört, bereits einen komplett replizierten Zustand des wegfallenden Nodes hat. Der Node N, der sich abmelden will, muß den anderen

Nodes in seinem In-Set und allen anderen Nodes in den In-Sets der Nodes, die er simuliert, bescheid sagen, daß keine weiteren Aufrufe mehr an ihn geschickt werden sollen, sondern an den Node, der ihn und seine simulierten Nodes simuliert. Es findet also ein Art Verbiegen der In-Set-Verbindungen von dem Node N und all der Nodes, die er simuliert auf die anderen Nodes im Out-Set statt, so daß zukünftige Anfragen nicht mehr an ihn gerichtet werden. Die Nodes im In-Set führen noch laufende Befehle zu Ende aus und schicken an den Node N anschließend eine Nachricht, daß sie keine weiteren Aufrufe mehr an ihn schicken, sondern statt dessen den Node aufrufen, der den wegfallenden Node N simuliert. Nachdem der wegfallende Node von allen Nodes im In-Set, zu dem er und die von ihm simulierten Nodes gehören, diese Nachricht erhalten hat, ist der Node N aus dem In-Set raus. Der Node N sagt nun den Nodes im Out-Set bescheid, daß sie ebenfalls keine Aufrufe mehr an ihn schicken sollen. Wenn auch diese die im Moment noch laufenden Befehle beendet haben und den Node N aus ihrem Out-Set entfernt haben, schicken sie eine Bestätigung. Wenn sie dies bestätigen, weiß der Node, daß er einfach stopen kann, weil er für niemanden mehr sichtbar ist, und hält an.

Wenn der sich abmeldende Node N allein im Out-Set ist, muß er nur allen anderen Nodes in seinem eigenen In-Set und in den In-Sets, für die er einen Node simuliert, bescheid sagen, daß keine weiteren Aufrufe mehr an ihn geschickt werden sollen. Der Node N existiert dann nicht mehr für die Gitterstruktur. Die Daten, die er noch enthält, verteilt er auf die einzelnen Out-Sets. Anschließend kann der gesamte Out-Set wegfallen.

Wenn der sich abmeldende Node N der Rezeptionisten-Node ist entspricht der Ablauf wie oben beschrieben. Zusätzlich muß der Rezeptionisten-Node die Rezeptionisten-Aufgabe und die dafür notwendigen Informationen auf einen anderen Node übertragen.

7.2.2. Verteiltheit ohne Replikation

Die Verteilung ohne Replikation [Ahuja86] ist ein Sonderfall der zuvor beschriebenen teilweisen Replikation. Hierbei bilden alle Nodes zusammen einen einzigen großen In-Set und jeder Node bildet für sich einen eigenen Out-Set. Bei n Nodes hat man also n Out-Sets, aber nur einen In-Set (Abbildung 7.17). Ebenso kann die volle Replikation durch das Schema der teilweisen Replikation realisiert werden, indem alle Nodes einen einzigen Out-Set bilden (Abbildung 7.18).

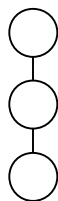


Abbildung 7.17.: Ohne Replikation als ein einzelner In-Set

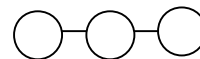


Abbildung 7.18.: Volle Replikation als ein einzelner Out-Set

Der wesentliche Unterschied der Verteilung ohne Replikation zur teilweisen Replikation liegt eigentlich nur in der Art und Weise, wie die Integration eines neuen Nodes in der Gitterstruktur erfolgt. Hierbei wird nicht versucht eine gleichmäßige Gitterstruktur aufzubauen, sondern es werden alle Nodes in einen In-Set gepackt. Damit entspricht der Ablauf der Anmeldung dem bei der teilweisen Replikation beschriebenen Verfahren, wenn bei der Anmeldung ein neuer Out-Set gebildet wird, weil jeder Node für sich allein einen eigenen Out-Set bildet. Entsprechend ist die Abmeldung gleich dem Abmeldeverfahren der teilweisen Replikation, wenn es sich um den letzten echten Node im Out-Set handelt, weil jeder Node generell für sich allein im Out-Set ist.

Auch die in()-, out()- und rd()-Methoden entsprechen vom Ablauf her den im vorherigen Kapitel gemachten Ausführungen:

- Out(): Da jeder Node für sich einen eigenen Out-Set bildet, muß das Tuple nur zum lokalen Tuplespace hinzugefügt werden. Auf diese Art entsteht eine Verteilung ohne Replikation, weil es keine weiteren Nodes im Out-Set gibt, auf denen die Tuples repliziert werden müssten.
- In(): Bei dem Ablauf von in() ist die Konsistenzerhaltung einfacher, da es keine Replikate gibt. Damit fällt die ganze Sperrproblematik weg. Wenn lokal oder auf einem Node im In-Set ein matchendes noch nicht gesperrtes Tuple gefunden wird, kann dieses einfach lokal gesperrt und gelöscht werden. Es muß kein vorheriger Wettkampf um die Sperre mit anderen Nodes stattfinden, weil das Tuple einmalig und nicht repliziert ist.
- Rd(): Bzgl. Rd gibt es überhaupt keinen Unterschied zum Ablauf bei teilweiser Replikation. Es wird lokal und auf allen Nodes im In-Set nach einem matchenden Tuple gesucht, egal ob gesperrt oder nicht und bei Erfolg zurückgegeben.

7.3. Realisierung in TSpace

Dieses Kapitel gibt einen Überblick über die Umsetzung des in den vorherigen Abschnitten beschriebenen Verteilungskonzeptes in TSpace. Kapitel 7.3.1. zeigt, wie das Distributor-Objekt in die TSpace-Architektur integriert werden kann, so daß der Tuplespace-Server die verteilten Befehle an den Distributor weiterleitet, der sich dann um dessen Umsetzung nach einer bestimmten Verteilungsstrategie kümmert. Kapitel 7.3.2. beschreibt das Distributor-Interface, das von den Klassen, die eine Verteilungsstrategie für XMLSpace realisieren wollen, implementiert werden muß. Kapitel 7.3.3. beschreibt die Distributor-Klasse PartialReplicationDistributor, die die teilweise Replikation realisiert, und zeigt, wie die Distributor-Klasse die dafür notwendigen Zugriffe auf das lokale und die entfernten Tuplespaces umsetzen kann. Kapitel 7.3.4. enthält einen zusammenfassenden Überblick über die gesamte Klassenstruktur zur Erweiterung von TSpace um Verteiltheit. Kapitel 7.3.5. beschreibt die notwendigen Schritte zum Starten des XMLSpaceServers und Kapitel 7.3.6. zeigt einen Beispielablauf für den Client-seitigen verteilten Zugriff. Kapitel 7.3.7. schließlich zeigt, ob und wie sonstige Eigenschaften und Methoden von TSpace, wie z.B. Transaktionen, Events usw. unterstützt werden.

7.3.1. Integration des Distributors in TSpace

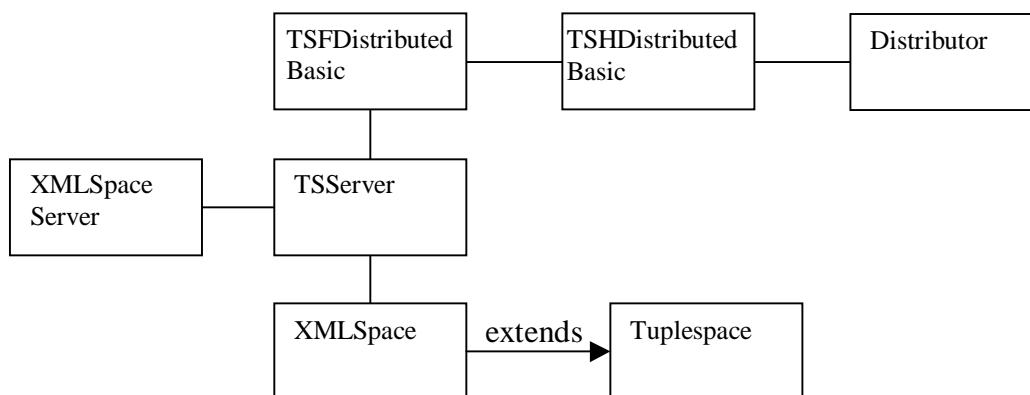


Abbildung 7.19.: Klassenstruktur für die Erweiterung von TSpace um ein Distributor-Objekt

Dieses Kapitel beschreibt die Integration des Distributor-Objektes, das sich um die Realisierung der Verteilung kümmert, in die TSpace-Architektur.

In TSpace wird ein Tuplespace durch den TSServer (=Tuplespace-Server) verwaltet. Der Zugriff eines Clients auf den TSServer erfolgt über ein TupleSpace-Objekt. Zu diesem Zweck erzeugt der Client ein TupleSpace-Objekt, das eine Verbindung zum TSServer aufbaut und die Befehlsaufrufe des Clients an den TSServer weiterleitet. Der Client kann entweder die TupleSpace-Methode `write`, `waitToTake` usw. für den Zugriff auf den TSServer nehmen oder die Methode `command(cmdString, argTuple)`. Die Methode `command` erhält als Parameter einen Kommando-String, der angibt, welche Methode auf dem TSServer ausgeführt werden soll und ein Tuple, das die Argumente für die auszuführende Methode enthält. Zum Beispiel führt der Aufruf von `TupleSpace.command(TupleSpace.WAITTOREAD, template)` ein `WAITTOREAD` (entspricht einem `rd()` in TSpace) auf dem TSServer aus. Die TupleSpace-Methoden `waitToRead`, `write` usw. werden letztendlich auf entsprechende `command`-Aufrufe (`command(TupleSpace.WRITE,...)` usw.) abgebildet.

Der TSServer nimmt den Befehlsaufrufe vom Client-seitigen TupleSpace-Objekt entgegen und führt ihn auf dem von ihm verwalteten Tuplespace aus. Zur besseren Unterscheidung zwischen dem vom TSServer verwalteten Tuplespace und dem TupleSpace-Objekt, über das der Client auf den TSServer und damit auf das Tuplespace zugreift, wird im Folgenden das serverseitige Tuplespace kurz `TS` genannt und das TupleSpace-Objekt auf der Client-Seite weiterhin `TupleSpace`. Abbildung 7.20 zeigt, wie ein Client mit Hilfe eines TupleSpace-Objektes auf einen TSServer zugreifen kann, der das `TS` verwaltet.

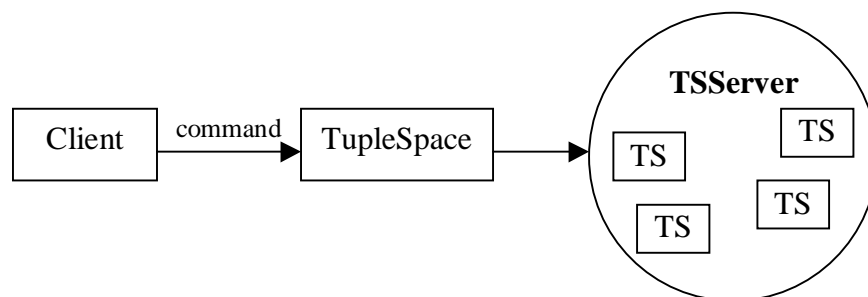


Abbildung 7.20: Der Zugriff auf den TSServer über ein Tuplespace-Objekt

Wie in der Abbildung angedeutet, kann ein TSServer mehrere `TS` verwalten. Der Client gibt beim Erzeugen des `TupleSpace`-Objektes mit an, auf welches `TS` sich seine Befehlsaufrufe richten. Zum Beispiel wird durch

```

TupleSpace ts = new TupleSpace("Test");
  
```

ein `TupleSpace`-Objekt erzeugt, das die Befehle auf einem `TS` namens "Test" ausführt. Der TSServer sucht nach dem Erhalten eines Client-Aufrufs den `TS` heraus, auf den sich der Client-Befehl bezieht. Jeder `TS` ist mit einem `TSFactory`-Objekt verknüpft. Über dieses `TSFactory`-Objekt kann der TSServer ein `TSHandler`-Objekt heraussuchen, das den vom Client aufgerufenen Befehl ausführt. Wenn der Client zum Beispiel

```

TupleSpace ts = new TupleSpace("Test");
ts.command(TupleSpace.WAITTOREAD, template);
  
```

aufruft, so sucht der TSServer über das `TSFactory`-Objekt des `TS` "Test" einen entsprechenden `TSHandler` für den Kommandostring `TupleSpace.WAITTOREAD` heraus und

lässt von diesem TSHandler-Objekt das Kommando mit dem Argument `template` ausführen. Der TSHandler führt in diesem Beispiel den `WAITTOREAD`-Befehl aus und gibt als Ergebnis das mit dem `template` matchende Tuple zurück. Das Ergebnis wird wiederum vom TSServer an den Client zurückgeschickt.

TSpace bietet die Möglichkeit zur Laufzeit neue TSpaceFactory- und TSpaceHandler-Objekte hinzuzufügen und auf diese Art den TSServer um neue Befehle, wie z.B. ein `DISTRIBUTED_WAITTOREAD` zu erweitern.

Genau dies wird von der Klasse `XMLSpaceServer` gemacht: Sie startet den TSServer und erweitert ihn um neue TSpaceFactory- und TSpaceHandler-Objekte, so daß der TSServer erkennt, wenn ein verteiltes `waitToRead` oder ein verteiltes `write` ausgeführt werden soll und den Aufruf an das Distributor-Objekt weiterleitet, das sich um die Umsetzung dieser verteilten Methoden kümmert.

Die neuen TSpaceFactory- und TSpaceHandler-Objekte werden durch die Klasse `TSpaceFDistributedBasic` und `TSpaceSHDistributedBasic` realisiert. Das `TSpaceFDistributedBasic`-Objekt erkennt die verteilten Befehlsaufrufe `XMLSpace.DISTRIBUTED_WAITTOREAD`, `XMLSpace.DISTRIBUTEDWAITTOTAKE` und `XMLSpace.DISTRIBUTEDWRITE` und erzeugt dafür ein `TSpaceSHDistributedBasic`-Objekt, das sich um die Umsetzung dieses Befehls kümmern soll. Das `TSpaceSHDistributedBasic`-Objekt leitet den Befehlsaufruf einfach an den Distributor weiter, der sich um die tatsächliche Umsetzung kümmert und nach der Ausführung des Befehls ein Ergebnis an das `TSpaceSHDistributedBasic`-Objekt zurückgibt, das das Ergebnis über den TSServer an den Client zurückschickt.

Auf diese Art ist der Distributor integriert und der TSServer leitet über die `TSpaceFDistributedBasic`- und `TSpaceSHDistributedBasic`-Objekte sämtliche verteilten Befehle automatisch an ihn weiter.

Für den Aufruf der verteilten Methoden auf den TSServer verwendet der Client nicht mehr länger ein `TupleSpace`-Objekt, sondern ein `XMLSpace`-Objekt. Die Klasse `XMLSpace` ist von `TupleSpace` abgeleitet und realisiert für einen Client den Aufruf der verteilten Methoden auf dem TSServer. Hierzu überschreibt `XMLSpace` die Methoden `waitToRead`, `write` usw., so daß auf Serverseite nicht mehr die normalen lokalen `waitToRead`, `write`-Methoden aufgerufen werden, sondern die verteilten. Zum Beispiel ruft die Methode `XMLSpace.waitToRead` nicht `command(TupleSpace.WAITTOREAD)`, sondern `command(XMLSpace.DISTRIBUTED_WAITTOREAD)` auf:

```
public Tuple waitToRead(Tuple template){
    command(XMLSpace.DISTRIBUTED_WAITTOREAD, template);
}
```

Für den Client bleibt der verteilte Zugriff verborgen, weil er weiterhin die normalen Methoden wie z.B. `write` oder `waitToRead` verwendet.

Die entsprechenden Kommandostrings, die vom TSServer durch das `TSpaceFDistributedBasic`-Objekt als verteilte Kommandos erkannt werden, sind in `XMLSpace` als entsprechende Konstanten deklariert:

```
public static final String DISTRWRITE="DistrWrite";
public static final String DISTRWAITTOREAD="DistrWaitToRead";
public static final String DISTRWAITTOTAKE="DistrWaitToTake";
```

Damit ergibt sich folgender Gesamtprozess: Der `XMLSpaceServer` startet den TSServer und erweitert ihn um `TSpaceFDistributedBasic`- und `TSpaceSHDistributedBasic`-Objekte. Ein Client, der auf den TSServer zugreifen will erzeugt ein `XMLSpace`-Objekt. Wenn der Client ein Kommando wie z.B. `waitToRead` aufruft, so leitet das `XMLSpace`-Objekt einen Kommandoaufruf für

einen verteilten Befehl wie z.B. XMLSpace.DISTRIBUTED_WAITTOREAD an den TSServer weiter. Der TSServer gibt den Aufruf an das TSFDistributedBasic-Objekt weiter, das den Aufruf als einen verteilten Befehl erkennt und den Aufruf an das TSHDistributedBasic-Objekt weitergibt, das den Aufruf wiederum an das Distributor-Objekt weiterleitet, das die eigentliche Verteilung nach einer bestimmten Verteilungsstrategie realisiert. Das Distributor-Objekt führt den verteilten Befehl aus und gibt das Ergebnis an den TSHDistributedBasic, der es über den TSServer an den Client zurückschickt.

7.3.2. Das Distributor-Interface

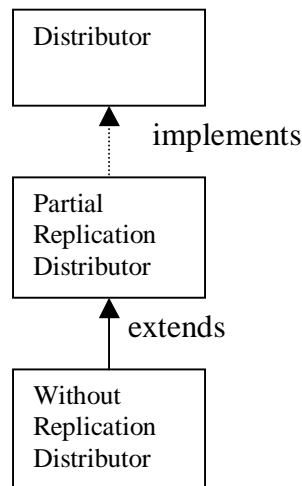


Abbildung 7.21.: Die Klassen zur Realisierung der Verteilungsstrategie

Das Distributor-Objekt realisiert die Verteilung nach einer bestimmten Verteilungsstrategie und wird vom TSServer aufgerufen, um einen verteilten Befehl auszuführen. Klassen, die Distributor-Objekte realisieren, müssen das Distributor-Interface implementieren. Das Distributor-Interface spezifiziert die Methoden, die vom TSServer aufgerufen werden können, um verteilte Befehle an den Distributor weiterzuleiten. Dazu gehört das An- und Abmelden und das verteilte Ausführen von in, out und rd. Das folgende ist ein Ausschnitt aus dem Distributor-Interface. Das komplette Distributor-Interface befindet sich im Anhang C.

```

public interface Distributor{
    public SuperTuple register(...);
    public SuperTuple deregister(...);
    public SuperTuple distributedWrite(...);
    public SuperTuple distributedWaitToTake(...);
    public SuperTuple distributedWaitToRead(...);
}
  
```

Hierbei sind die Methoden register und deregister für das An- und Abmelden des lokalen TSServer im verteilten Tuplespace, und die Methoden distributedWrite, -WaitToTake und -WaitToRead für das verteilte Ausführen von out, in und rd verantwortlich.

In XMLSpace gibt es zur Zeit zwei Klassen, die Verteilungsstrategien realisieren:

- Die Klasse PartialReplicationDistributor realisiert eine Verteilung mit teilweiser Replikation wie im Kapitel 7.2.1. beschrieben.

- Die Klasse WithoutReplicationDistributor realisiert eine Verteilung ohne Replikation wie im Kapitel 7.2.2. beschrieben. Die Klasse WithoutReplicationDistributor ist von der Klasse PartialReplicationDistributor abgeleitet und überschreibt nur die Methoden für die An- und Abmeldung. Alle anderen Methoden werden unverändert übernommen. Wie im Kapitel 7.2.2. beschrieben unterscheidet sich das Verfahren ohne Replikation von dem Verfahren mit teilweiser Replikation nur in der Art, wie die Gitterstruktur aufgebaut wird.

Welche Art von Distributor (Partial- oder WithoutReplicationDistributor) verwendet werden soll, wird beim Serverstart angegeben. Nähere Informationen dazu gibt es im Kapitel 7.3.5..

7.3.3. Realisierung der teilweisen Replikation

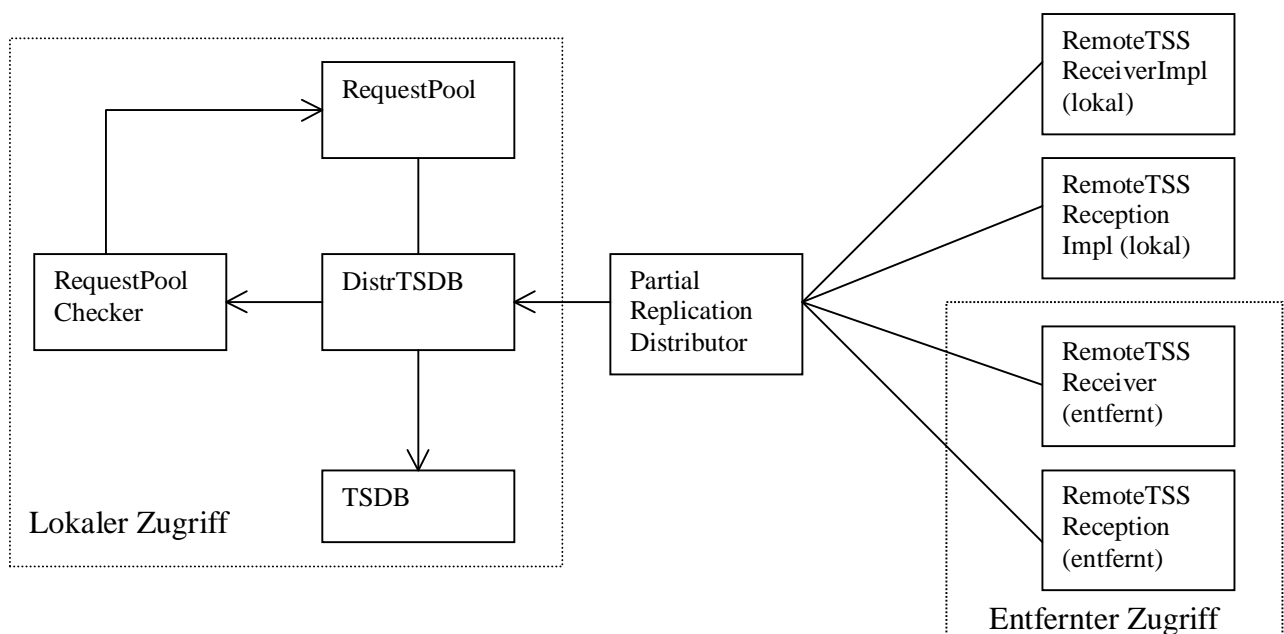


Abbildung 7.22.: Die Klassen für den Zugriff auf den lokalen und die entfernten Tuplespaces

Die PartialReplicationDistributor-Klasse realisiert Verteilung mit teilweiser Replikation. Wie im vorherigen Abschnitt beschrieben implementiert PartialReplicationDistributor das Distributor-Interface, so daß der TSServer die Aufrufe der verteilten Befehle an den PartialReplicationDistributor weiterleiten kann.

Wenn der XMLSpaceServer gestartet wird integriert er den PartialReplicationDistributor-Objekt im TSServer und ruft die Methode register() auf, durch die der PartialReplicationDistributor die im Kapitel 7.2.1.5. beschriebene Anmeldung in der Gitterstruktur durchführt. Dafür erzeugt der PartialReplicationDistributor einerseits ein RemoteTSSReceptionImpl- und ein RemoteTSSReceiverImpl-Objekt, über die andere entfernte TSServer auf ihn zugreifen können, und andererseits registriert er sich über RemoteTSSReception-Objekte auf den entfernten TSServern, die zu seinen In- und Out-Sets gehören, und erhält als Ergebnis auf seine Anmeldung eine Menge von Referenzen auf RemoteTSSReceiver-Objekte, die er intern in seine In- und Out-Sets ablegt. Über die RemoteTSSReceiver-Objekte kann der PartialReplicationDistributor auf die entfernten TSServer zugreifen und Tuples suchen, löschen, sperren oder einfügen. RemoteTSSReceiver-

Objekte müssen das RemoteTSSReceiver-Interface implementieren, das folgende Methoden spezifiziert:

```
public interface RemoteTSSReceiver extends Remote{

    public void addTuple(...);
    public bool lockTuple(...);
    public void unlockTuple(...);
    public void deleteTuple(...);
    public SuperTuple findAndLockTuple(...);

}
```

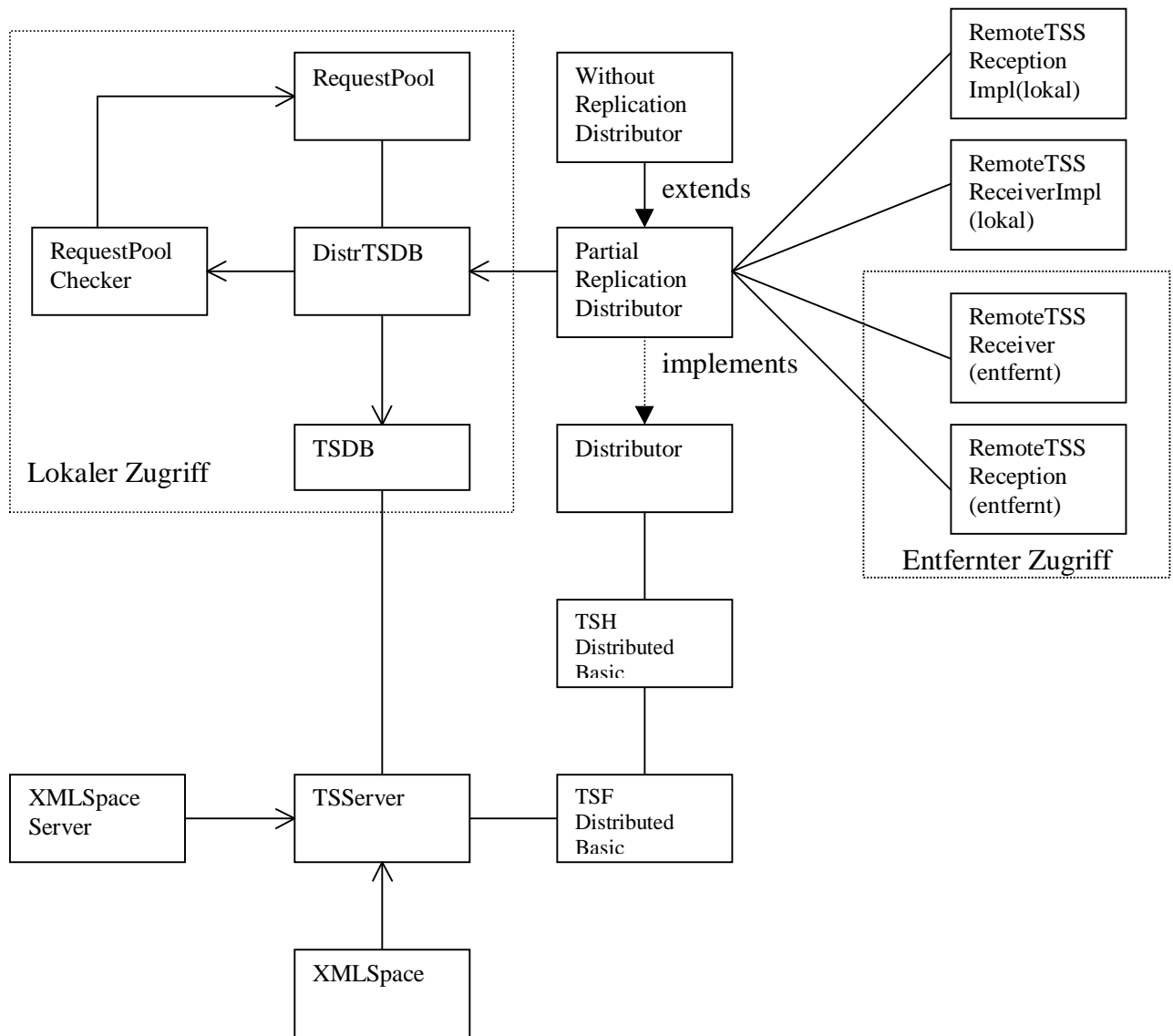
Für den Zugriff auf den lokalen TSServer enthält das PartialReplicationDistributor-Objekt einen Vektor mit Referenzen auf DistrTSMMDB-Objekte. Jedes TS hat ein TSDB (Tuplespace-Database-Manager), das die Daten im Tuplespace auf unterster Ebene verwaltet. Das DistrTSMMDB kapselt dieses TSDB-Objekt und erweitert es um Funktionalitäten zur Unterstützung der Verteiltheit. So verwenden zum Beispiel die Methoden der TSDB-Klasse TupleIDs zum Löschen oder Sperren von Tuples. Diese TupleIDs sind jedoch nur lokal eindeutig. Für die Ausführung der verteilten Methoden benötigt man jedoch global eindeutige TupleIDs. Daher gibt es im DistrTSMMDB eine Tabelle zum Abbilden der global eindeutigen TupleIDs, die für die verteilten Methoden benötigt werden, auf die lokalen TupleIDs, die von TSDB verwendet werden. Auf diese Art können die TSDB-Methoden weiterverwendet werden, indem vor dem Zugriff auf dem TSDB die globale TupleId über die Tabelle in eine lokale TupleId umgewandelt wird.

Das DistrTSMMDB enthält ein RequestPool-Objekt, das die In- und Rd-Anfragen enthält, die nicht sofort erfüllt werden konnten und später periodisch erneut aufgerufen werden. Das RequestPool-Objekt ist durch einen einfachen Vektor realisiert mit Methoden zum Hinzufügen, Löschen und Durchsuchen des RequestPools.

Der RequestPoolChecker ist ein Thread, der periodisch nacheinander die Einträge im RequestPool überprüft. Dazu nimmt er die erste Anfrage aus dem RequestPool und führt sie lokal und entfernt aus. Wenn er erfolgreich ist, gibt er das Ergebnis an den Client zurück, der diese Anfrage aufgerufen hat. Ansonsten wird die Anfrage am Ende des RequestPools hinzugefügt, um nach einem gewissen Zeitabstand erneut aufgerufen zu werden.

7.3.4. Gesamtüberblick der Klassenstruktur

Die folgende Grafik stellt noch einmal die Klassenstruktur für die Realisierung der Verteiltheit auf der Basis von TSpaces als Ganzes dar.



7.3.5. Starten des XMLSpace-Servers

Der XMLSpaceServer wird durch den Aufruf von

```
java XMLSpaceServer -distr ConfigFileName -sonstigeOptionen
```

gestartet. Die sonstigen Optionen sind hierbei die Optionen, die beim Starten des normalen TSServer mit angegeben werden können. Zum Beispiel wird über „-p Portnummer“ festgelegt, daß der TSServer Client-Aufrufe über den angegebenen Port entgegen nimmt (weitere Optionen sind im Anhang A beschrieben).

Der hinter -distr angegebene ConfigFileName muß auf eine Datei verweisen, die die notwendigen Informationen zum Initialisieren des Distributors und zum Aufbau der Verbindungen zu den anderen entfernten TSServern enthält.

Die Konfigurationsdatei enthält immer als ersten Eintrag die Art der Verteilungsstrategie, die verwendet werden soll, in der Form STRATEGY = Klassenname, z.B. STRATEGY=PartialReplicationDistributor. Die restlichen Inhalte sind von der verwendeten Verteilungsstrategie abhängig. Zum Beispiel benötigt der PartialReplicationDistributor für die Anmeldung bei den anderen Servern die Adresse des Rezeptionisten, über den die Anmeldung erfolgt. Der Eintrag sieht für das PartialReplicationDistributor-Objekt folgendermaßen aus: RECEPTION=www-Adresse.

Der XMLSpaceServer erzeugt anhand des Strategy-Eintrages das entsprechende Distributor-Objekt und das Distributor-Objekt erhält die restlichen Informationen zur Initialisierung und zum Aufbau der Verbindungen aus der Konfigurationsdatei.

Wenn kein Distributor-Objekt angegeben wird, wird standardmäßig rein lokal gearbeitet.

Die komplette Konfigurationsdatei sieht für den PartialReplicationDistributor z.B. folgendermaßen aus:

```
STRATEGY=PartialRecliationDistributor
RECEPTION=www.cs.tu-berlin.de/~glaubitz/rezeption
```

7.3.6. Beispielablauf für die Client-seitige Verwendung des XMLSpaces bei Verteiltheit

Die Verteilung bleibt für den Client komplett verborgen. Der Client erzeugt ein XMLSpace-Objekt mit denselben Parametern wie bei einem normalen TupleSpace-Objekt. Zum Beispiel erzeugt

```
XMLSpace xs = new XMLSpace("test")
```

ein XMLSpace-Objekt für den Zugriff auf den Tuplespace namens „Test“, das vom lokalen TSServer verwaltet wird.

Der Client kann die normalen von TupleSpace geerbten Methoden write, waitToRead, waitToTake usw. aufrufen. Die Methoden werden verteilt ausgeführt, ohne daß der Client davon etwas mitbekommt. Wenn er z.B.

```
Tuple ergebnis = xs.waitToTake(template);
```

aufruft, weiß er nicht, ob das Ergebnis-Tuple vom lokalen oder einem entfernten Tuplespace-Server stammt.

Die Verteiltheit ist damit komplett verborgen vor dem Client.

7.3.7. Unterstützung sonstiger Methoden und Eigenschaften von TSpace

Die in den Kapitel Teilweise Replikation (Kapitel 7.2.1.) und Ohne Replikation (Kapitel 7.2.2.) beschriebenen Verfahren beschränken sich auf die lindaspezifischen Operationen out, in und rd, die in TSpace durch die Methoden write, waitToTake und waitToRead dargestellt werden. TSpace bietet neben den lindaspezifischen Methoden write, waitToRead und waitToTake, aber auch nicht-lindaspezifische Methoden wie z.B. scan, consumingScan, delete und ähnliches an. Die Tabelle 7.23 enthält einen Überblick über die mögliche Realisierung solcher Methoden im verteilten Fall für teilweise Replikation. Die Realisierung für Verteilung ohne Replikation entspricht dem Ablauf der teilweisen Replikation mit einem einzigen In-Set.

Darüber hinaus bietet TSpace bestimmte Eigenschaften wie z.B. Transaktionsunterstützung, Events und ähnliches. Die Tabelle 7.24 enthält eine Übersicht über diese Eigenschaften und ob und wie diese Eigenschaften im verteilten Fall unterstützt werden.

Name der Methode	Funktionalität	Ablauf bei teilweiser Replikation
consumingScan(template)	Löscht sämtliche mit dem template matchenden Tuples und gibt sie als Ergebnis zurück.	Es wird versucht so viele matchende Tuples wie möglich sowohl im lokalen Tuplespace als auch auf allen Nodes im In-Set zu sperren. Die erfolgreich gesperrten Tuples werden gelöscht und als Ergebnis zurückgegeben.
count(template)	Gibt die Anzahl der dem template entsprechenden Tuple zurück.	Sucht lokal und auf allen Nodes im In-Set nach matchenden Tuples (egal ob gesperrt oder nicht) und gibt die Summe der gefundenen Tuples als Ergebnis zurück.
delete(template)	Löscht alle Tuple, die dem angegebenen template entsprechen.	Entspricht einem consumingScan, nur daß als Ergebnis nicht die gelöschten Tuples zurückgegeben werden.
deleteAll()	Löscht sämtliche Tuples in einem Tuplespace.	Wie bei consumingScan, nur daß nicht nur die matchenden, sondern sämtliche Tuples zu sperren und zu löschen versucht werden.
deleteTupleById(id)	Löscht ein Tuple mit der angegebenen Tuple-Id. Die Tuple-Id erhält man z.B. indem man das Tuple über ein waitToRead aus dem Tuplespace ausliest.	Entspricht dem Ablauf von take, nur daß ein Tuple nicht anhand eines Templates, sondern anhand der TupleId herausgesucht wird und das Tuple einfach gelöscht wird, ohne es zurückzugeben.
multiWrite(Tuples)	Schreibt mehrere Tuples mit einem mal ins Tuplespace.	Fügt die einzelnen Tuples der Reihe nach entsprechend dem im Kapitel 7.2.1.4. angegebenen Verfahren für ein out zum Tuplespace hinzu.
read(template)	Nicht blockierendes waitToRead: Wenn kein matchendes Tuple gefunden wird, wird nicht gewartet, sondern ein Nullwert zurückgegeben.	Entspricht dem Ablauf von waitToRead (d.h. dem in Kapitel 7.2.1.3. beschriebenen rd), nur daß bei nicht Finden eines Tuples die read-Anfrage nicht in den Request-Pool gesteckt wird, sondern ein Null-Wert zurückgegeben wird.
readTupleById(id)	Führt ein Read auf der angegebenen Tuple-Id aus.	Entspricht dem Ablauf von read, nur daß ein Tuple nicht anhand eines Templates, sondern anhand der TupleId herausgesucht wird.
scan(template)	Gibt sämtliche matchenden Tuples zurück ohne sie zu löschen.	Sucht lokal und auf allen Nodes im In-Set nach sämtlichen matchenden Tuples (egal ob gesperrt oder nicht) und gibt sie als Ergebnis zurück.
take(template)	Nicht blockierendes WaitToTake: Wenn kein matchendes Tuple gefunden wird, wird nicht gewartet, sondern ein Nullwert zurückgegeben.	Entspricht dem Ablauf von waitToTake(d.h. dem in Kapitel 7.2.1.2. beschriebenen in), nur daß bei nicht Finden eines Tuples die take-Anfrage nicht in den Request-Pool gesteckt wird, sondern ein Null-Wert zurückgegeben wird.
takeTupleById(id)	Führt ein Take auf der angegebenen Tuple-Id aus.	Entspricht dem Ablauf von take, nur daß ein Tuple nicht anhand eines Templates, sondern anhand der TupleId herausgesucht wird.

Tabelle 7.23.: Nicht-lindaspezifische Methoden von TSpace und deren Realisierung im verteilten Fall

Eigenschaft	Unterstützung bei Verteilung
Events	Die Realisierung von verteilten Events ist im Kapitel 8 beschrieben.
Security	XMLSpace bietet zur Zeit keine Unterstützung für Security bei Verteiltheit. Siehe auch Kapitel 10 (Ausblick und offene Punkte).
Transaktionen	TSpace unterstützt Transaktionen über Methoden zum Starten (beginTransaction), Bestätigen (commitTransaction) und Abbrechen von Transaktionen (abortTransaction). Diese Transaktionen eignen sich nur für lokale, nicht verteilte Anwendungen. XMLSpace unterstützt zur Zeit keine verteilten Transaktionen (siehe auch Kapitel 10). Bei der Verwendung von Transaktionen im Zusammenhang mit dem verteilten XMLSpace kann es zu unvorhersehbaren, fehlerhaften Verhalten kommen. Die Verwendung von Transaktionen sollte deshalb im Zusammenhang mit XMLSpace vermieden werden.
Tuple Expiration	TSpace bietet die Möglichkeit für Tuples über die Methode Tuple.setExpire() eine Art Verfallsdatum anzugeben. Wenn das Verfallsdatum abgelaufen ist, wird das Tuple automatisch aus dem Tuplespace gelöscht. Tuple Expiration wird zur Zeit nicht von XMLSpace unterstützt. Es kann zu Inkonsistenzen führen, wenn Tuple Expiration im Zusammenhang mit dem verteilten XMLSpace verwendet wird.

Tabelle 7.24.: Eigenschaften von TSpaces und deren Unterstützung im verteilten Fall

8. Verteilte Events

XMLSpace unterstützt das registrieren von verteilten Events. Dadurch ist es für einen Client möglich automatisch informiert zu werden, wenn ein bestimmtes Ereignis im verteilten Tuplespace eintritt, wie z.B. das Hinzufügen oder Löschen eines Tuples. Auf diese Art können zum Beispiel die Engines in der Workspace-Architektur automatisch informiert werden, wenn neue zu bearbeitende Schritte vorliegen.

Das Kapitel 8.1. beschreibt, wie die Behandlung von Events im Zusammenhang mit teilweiser Replikation erfolgt. Dazu gehört das verteilte An- und Abmelden der Events, das Auslösen von Events und die Behandlung der bereits registrierten Events, wenn neue Server zur Gitterstruktur hinzukommen oder Server wegfallen. Das Kapitel 8.2. beschreibt kurz die Event-Behandlung bei Verteilung ohne Replikation. Da die Verteilung ohne Replikation ein Sonderfall der teilweisen Replikation ist, lässt sich die Behandlung der Events für Verteilung ohne Replikation direkt aus der Behandlung der Events bei teilweiser Replikation ableiten. Das Kapitel 8.3. gibt schließlich einen Überblick, wie die verteilten Events auf TSpace aufbauend realisiert werden können.

8.1. Behandlung der Events bei teilweiser Replikation

8.1.1. Anmelden neuer Events

Wenn ein Client auf seinem lokalen TSServer (Tuplespace-Server) ein Event anmeldet, muß der lokale TSServer dafür Sorgen, daß das Event so im verteilten Tuplespace registriert wird, daß sämtliche entsprechenden Ereignisse im verteilten Tuplespace erfasst werden. Zu diesem Zweck muß der lokale TSServer das Event sowohl bei sich selbst im lokalen Tuplespace als auch auf allen Nodes in seinem In-Set registrieren. Wie in Kapitel 7.2.1.1. beschrieben, bildet die Vereinigung des Inhalts aller Nodes eines In-Sets den gesamten Inhalt des verteilten Tuplespace. Auf diese Art sind sämtliche Änderungen im verteilten Tuplespace für den Client sichtbar (Abbildung 8.1.).

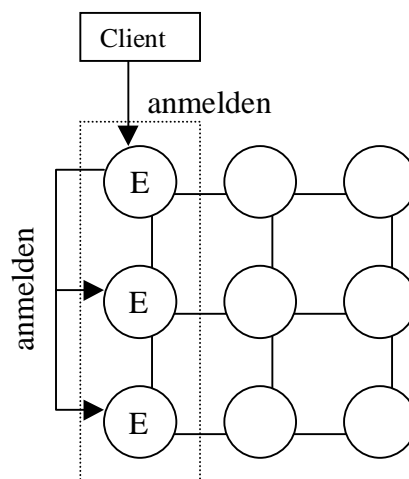


Abbildung 8.1.: Das Anmelden eines verteilten Events

8.1.2. Auslösen von Events

Wenn irgendwo im In-Set ein registriertes Event erfüllt wird, dann gibt es zwei Varianten:

- Entweder handelt es sich um ein lokales Event, d.h. das Event fand in dem Tuplespace statt, auf dessen TSServer der Client die Event-Registrierung aufgerufen hat. In diesem Fall benachrichtigt der TSServer den Client direkt (Abbildung 8.2.).

- Oder es handelt sich um ein entferntes Event, d.h. das Event fand auf einem anderen Node im In-Set statt. In diesem Fall wird das Event von dem Node, auf dem das Event stattfand, an den Node weitergeleitet, der dieses entfernte Event für einen lokalen Client registriert hat. Dieser Node leitet das Event schließlich an den Client weiter. (Abbildung 8.3).

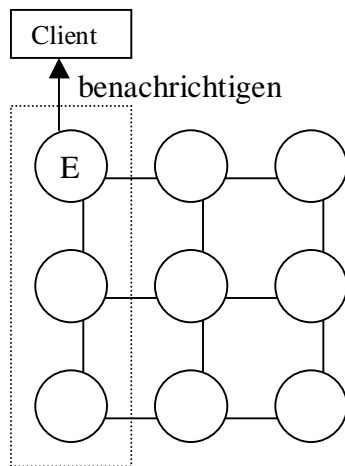


Abbildung 8.2: Lokale Event-Benachrichtigung

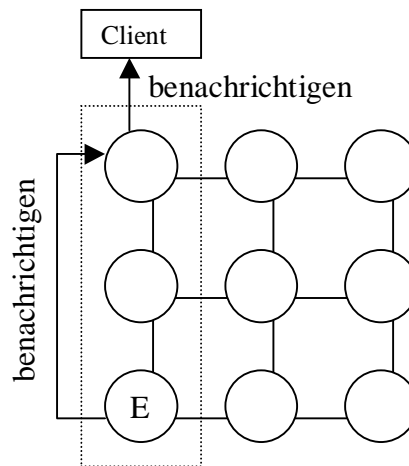


Abbildung 8.3.: Entfernte Event-Benachrichtigung

In dem Kapitel 7.2.1.4. wurde beschrieben, daß ein Out() zunächst gegen die wartenden Anfragen im Request-Pool überprüft wird und daß bei einem Erfüllen eines wartenden lokalen In() das Tuple nicht erst an die anderen Nodes im Out-Set geschickt werden muß, weil es gleich lokal konsumiert wurde und auf diese Art Kommunikationskosten eingespart werden können. Dadurch entsteht die Problematik, daß evtl. Events verschluckt werden, weil das sofort lokal konsumierte Tuple nicht für die anderen Nodes im Out-Set sichtbar ist.

Eine mögliche Lösung ist, daß für den Fall, daß ein über Out() hinzugefügtes Tuple gleich lokal durch ein wartendes In() konsumiert wird, die anderen Nodes im Out-Set benachrichtigt werden, daß ein Out() und ein In()-Event auf dieses Tuple stattfand und die anderen Nodes im Out-Set dieses Event bei Bedarf weiterleiten können, falls es ein registriertes Event dafür gibt (Abbildung 8.4). Auf diese Art bleiben weiterhin alle Events im Out-Set sichtbar. Natürlich entstehen dadurch wieder Kommunikationskosten, die eigentlich durch die sofortige lokale Konsumierung vermieden werden sollten, aber diese Kommunikationskosten sind wesentlich geringer, als wenn das Tuple erst komplett auf dem Out-Set repliziert und dann wieder über einen In() entfernt wird.

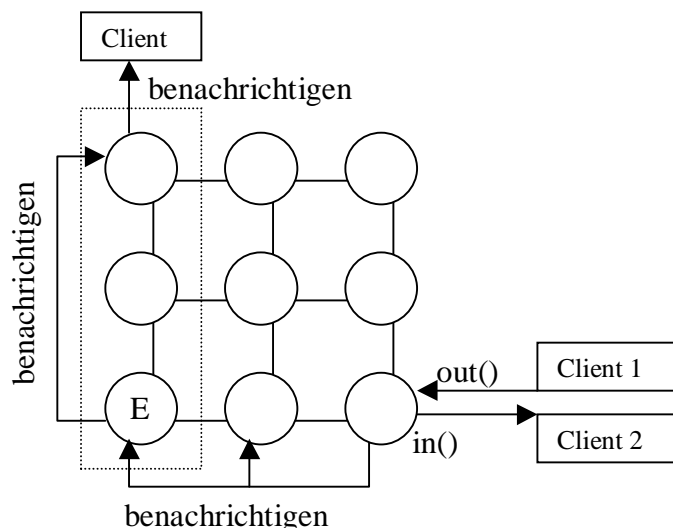


Abbildung 8.4: Indirekte Event-Benachrichtigung

8.1.3. Abmelden von Events

Wenn ein Client ein Event auf seinem lokalen TSServer abmeldet, dann muß der TSServer das entsprechende Event lokal und auf allen Nodes in seinem In-Set abmelden. Auf diese Art ist das verteilte Event aus dem verteilten Tuplespace abgemeldet (Abbildung 8.5.).

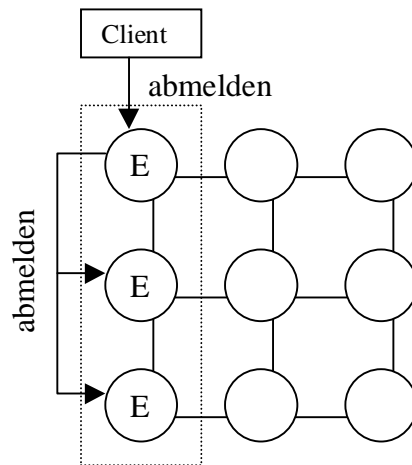


Abbildung 8.5.: Das Abmelden eines verteilten Events

8.1.4. Events und das Anmelden neuer Server

Zusätzlich zu dem im Kapitel 7.2.1.5. beschriebenen Verfahren bei der Anmeldung eines neuen Nodes, muß der neue Node sich mit den bereits registrierten Events abstimmen.

Beim Anmelden eines neuen Servers in der Gitterstruktur gibt es drei Varianten

1. Der neue Server ersetzt einen simulierten Node in der Gitterstruktur.
2. Der neue Server wird zu einem neuen In-Set hinzugefügt.
3. Der neue Server bildet einen neuen Out-Set.

Wenn der neue Server N einen von Node A simulierten Node S in der Gitterstruktur ersetzt, gibt es zwei Möglichkeiten, wie mit den bereits registrierten Events auf dem simulierten Node umgegangen werden kann (Abbildung 8.6). Entweder bleiben die Events auf dem Server A registriert, der den simulierenden Node S simuliert, weil es im Prinzip egal ist, auf welchem Server im Out-Set das Event registriert ist, da alle Nodes im Out-Set denselben Zustand haben und sämtliche Events mitbekommen. Oder die registrierten Events werden auf den neuen Server N verschoben.

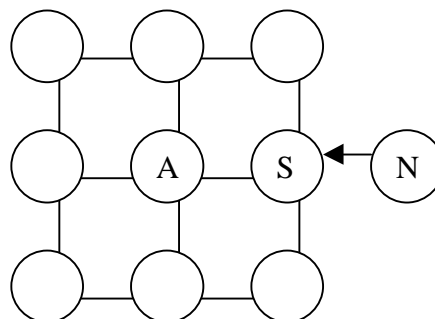


Abbildung 8.6.: Das Anmelden eines neuen Servers an Stelle eines simulierten Nodes.

Die erste Möglichkeit ist auf den ersten Blick die einfachere. Jedoch bedeutet dies, daß der Node A weiterhin die Verbindungen zu den Nodes im In-Set von S aufrecht erhalten muß,

um beim Eintreten eines Events die entsprechenden Nodes im In-Set von S benachrichtigen zu können.

Die zweite Möglichkeit erfordert, daß in der Zeit, in der die Events auf den neuen Node verschoben werden, sichergestellt wird, daß in dieser Zeit keine Events verschluckt oder dupliziert werden. Um dies zu gewährleisten muß der komplette Out-Set für einen Moment angehalten werden, d.h. es werden noch laufende Befehle, die sich auf den Out-Set als ganzes beziehen, wie z.B. das Hinzufügen oder Löschen von Tuples, ausgeführt, aber keine neuen Befehle mehr gestartet. Auf diese Art haben alle Nodes im Out-Set denselben Zustand und es ist gewährleistet, daß keine Events stattfinden können, die verschluckt oder dupliziert werden könnten. Die zu übertragenden registrierten Events können jetzt problemlos vom alten auf den neuen Node verschoben werden. Anschließend kann der Out-Set wieder seinen normalen Betrieb aufnehmen.

Wenn der anmeldende Node zu einem neuen In-Set hinzugefügt wird, brauch er sich mit niemanden bzgl. registrierter Events abzustimmen, weil alle anderen Nodes in seinem In-Set nur simulierte Nodes sind (Abbildung 8.7.).

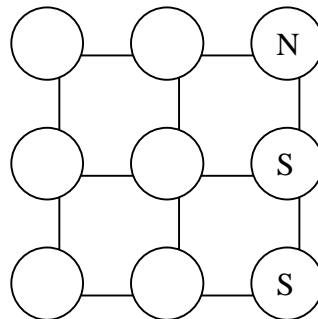


Abbildung 8.7.: Das Hinzufügen eines Nodes in einem neuen In-Set

Wenn der anmeldende Node einen neuen Out-Set bildet (Abbildung 8.8) muß er sich mit seinem eigenen In-Set und mit allen In-Sets, für die er einen Node simuliert, bzgl. der bereits registrierten Events abstimmen. Hierfür blockiert der Node N der Reihe nach die In-Sets bzgl. Event-Registrierung und –Deregistrierung, bis er sich mit A, B und C abgestimmt hat. In dieser Zeit dürfen keine Event-Registrierungen oder De-Registrierungen auf dem jeweiligen In-Set ausgeführt werden, damit der Node N beim Abstimmen keine Eventanmeldungen oder –abmeldungen verpasst. Erst wenn N sich abgestimmt und voll integriert ist (also alle nachfolgenden Event-Registrierungs- und –Deregistrierungsaufrufe mitbekommt), dürfen die In-Sets normal weiterlaufen. Alle anderen Methoden (in, out, rd usw.) dürfen ungehindert in dieser Zeit weiterlaufen.

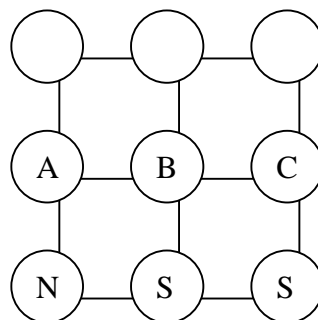


Abbildung 8.8.: Das Hinzufügen eines neuen Nodes als einen neuen Out-Set

8.1.5. Events und das Abmelden von Servern

Ein Server, der sich abmelden will, muß zunächst sämtliche lokalen Events, d.h. die Events die von einem Client direkt bei ihm angemeldet und von dem Server dann in den anderen Nodes in seinem In-Set angemeldet wurde, wieder lokal und auf allen Nodes im In-Set abmelden.

Anschließend muß der sich abmeldende Node die registrierten Events auf den Node im Out-Set verschieben, der ihn simulieren soll. Hierbei gibt es zwei Möglichkeiten: Entweder handelt es sich bei dem abmeldenden Node um den letzten Node im Out-Set oder es gibt noch weitere echte Nodes im Out-Set.

Wenn es sich um den letzten Node handelt, kann der abzumeldende Node N einfach wegfallen. Die bei ihm registrierten Events können verloren gehen, da in seinem Out-Set keine Events mehr passieren können, da der Out-Set nach seinem Wegfall leer ist (Abbildung 8.9.).

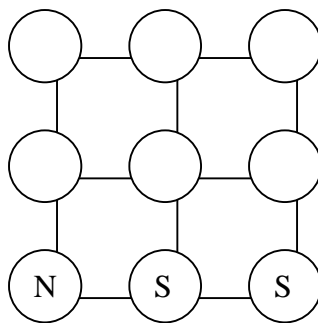


Abbildung 8.9.: Das Abmelden eines letzten Nodes im Out-Set.

Wenn sich der abmeldende Node nicht allein im Out-Set befindet (Abbildung 8.10), muß der wegfallende Node N die Events auf den Node A übertragen, der ihn simuliert. Für das Übertragen der Events auf den anderen Node müssen wie auch beim Anmelden neuer Server der Out-Set für einen kurzen Moment angehalten werden, so daß keine Events auf dem Out-Set stattfinden können. Die Events können dann übertragen werden und gehen damit durch den Wegfall des Nodes N nicht verloren.

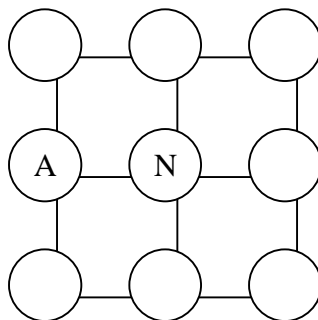


Abbildung 8.10.: Das Abmelden eines Nodes in einem Out-Set, der noch weitere Nodes enthält.

8.2. Behandlung der Events bei Verteilung ohne Replikation

Da die Verteilung ohne Replikation ein Sonderfall der teilweisen Replikation mit nur einem In-Set ist, kann der Umgang mit den Events aus den in den vorherigen Abschnitten gemachten Beschreibungen abgeleitet werden. Daraus ergeben sich zusammengefasst folgende Abläufe:

- Anmelden von Events: Das Event wird auf dem lokalen Server, auf dem die Anmeldung initiiert wurde, und auf allen Nodes in seinem In-Set angemeldet. Da es bei teilweiser Replikation nur einen In-Set gibt bedeutet dies, daß die Events generell auf allen Nodes registriert werden.
- Abmelden von Events: Das Event wird auf dem lokalen Server und auf allen Nodes im In-Set abgemeldet, also generell auf allen Nodes.
- Behandlung von registrierten Events beim Anmelden neuer Server: Dies entspricht bei der Verteilung mit teilweiser Replikation der Anmeldung, wenn der neue Node einen neuen Out-Set bildet (weil jeder Node für sich allein einen Out-Set bildet), d.h. der neue Node muß sich mit allen Nodes im In-Set abstimmen. Zur Abstimmung wird das gleiche Verfahren wie im Kapitel 8.1.4. beschrieben verwendet.
- Behandlung von registrierten Events beim Abmelden von Servern: Dies entspricht dem Abmelden eines Nodes bei teilweiser Replikation, wenn der sich abmeldende Node allein im Out-Set ist. Der abzumeldende Node kann einfach wegfallen und die bei ihm registrierten Events verloren gehen, da in seinem Out-Set keine Events mehr passieren können.

8.3. Realisierung in TSpace

TSpace bietet die Möglichkeit lokale Events zu registrieren. Diese Events müssen für XMLSpace um Verteiltheit erweitert werden.

Das An- und Abmelden eines Events erfolgt in TSpace über die TupleSpace-Methoden `eventRegister` und `eventDeRegister`. Zum Beispiel registriert der folgende Aufruf ein WRITE-Event für das angegebene Template `t` im Tuplespace „Test“, d.h. der Client wird benachrichtigt, wenn ein Tuple, das mit dem Template `t` matcht, zum Tuplespace „Test“ hinzugefügt wird:

```
TupleSpace ts = new TupleSpace("Test");
Tuple t = new Tuple("Key2",new Field(String.class) );
Callback cb = new CallbackObject();
int seqNum = ts.eventRegister(TupleSpace.WRITE,t,cb);
```

Der erste Parameter in der `eventRegister`-Methode gibt an, ob ein WRITE- oder ein DELETE-Ereignis registriert wird. Bei einem WRITE-Ereignis wird der Client benachrichtigt, wenn ein Tuple zum Tuplespace hinzugefügt wird (z.B. über ein `write()`), und bei einem DELETE-Ereignis wird der Client benachrichtigt, wenn ein Tuple aus dem Tuplespace gelöscht wird (z.B. über ein `waitToTake()`). Es gibt nur diese beiden Arten von Events. Es ist nicht möglich ein READ-Event zu registrieren. Der zweite Parameter ist ein Template, das angibt, für welche Art von Tuples das Ereignis stattfinden soll. Es werden nur Ereignisse angezeigt für Tuples, die mit diesem Template matchen. Der dritte Parameter ist ein Callback-Objekt, das benachrichtigt wird, wenn das Ereignis eingetreten ist. Die Klasse des Callback-Objektes muß das Callback-Interface implementieren. Das Callback-Interface spezifiziert die eine Methode

```
public call(String eventName,String tsName,
            int sequenceNumber, Tuple tuple, boolean isException)
```

die vom TSServer aufgerufen wird, wenn das Ereignis eingetreten ist. Über die Argumente eventName, tsName und tuple erhält der Client die Informationen in welchem Tuplespace auf welchem Tuple welches Ereignis stattgefunden hat.

Die Abmeldung eines Events erfolgt mit Hilfe der Sequenznummer, unter die das Event auf dem TSServer registriert ist, und die der Client beim eventRegister-Aufruf erhalten hat:

```
ts.eventDeRegister(seqNum);
```

Wenn der Client im verteilten XMLSpace eventRegister aufruft, muß der Distributor, der sich um die verteilte Event-Anmeldung kümmert, das Event auf dem lokalen TSServer und entfernt auf allen Nodes in seinem In-Set registrieren. Abbildung 8.11 veranschaulicht das Prinzip für die verteilte Registrierung des Events. Hierbei wird nur das prinzipielle Konzept wiedergegeben, ohne auf die internen Details von TSpace einzugehen.

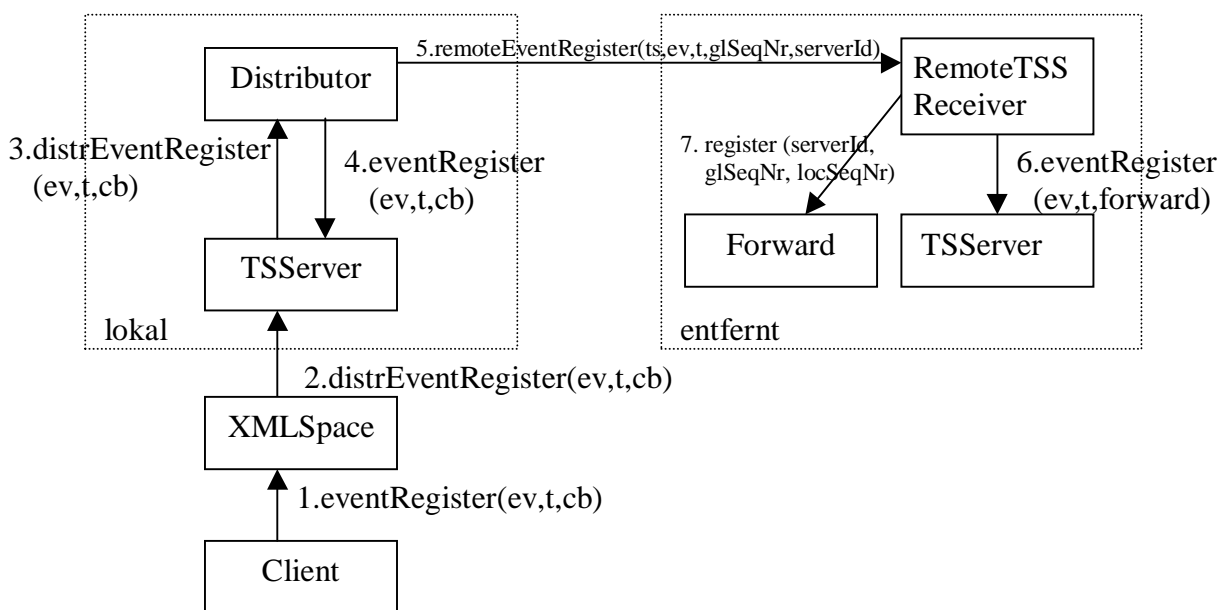


Abbildung 8.11.: Die Registrierung eines verteilten Events

Der Client ruft ein eventRegister(ev,t,cb) auf dem XMLSpace auf, das die Verbindung zum TSServer realisiert und einen distrEventRegister-Aufruf an den TSServer weiterleitet. Der TSServer erkennt, daß es sich um einen verteilten Befehl handelt und leitet den Aufruf an den Distributor weiter, der sich um die Verteilung kümmert. Der Distributor registriert das Event lokal und auf allen entfernten TSServern in seinem In-Set.

Die lokale Registrierung bleibt unverändert, d.h. es wird für den Client ein normales lokales Event registriert, so daß für den Fall, daß im lokalen Tuplespace ein entsprechendes Event auftritt, das Event an das Callback-Objekt des lokalen Clients weitergeleitet wird.

Für die entfernte Registrierung ruft der Distributor die Methode remoteEventRegister im RemoteTSSReceiver des jeweiligen Nodes im In-Set auf. Die Methode remoteEventRegister erhält als Argumente:

- ts: den Namen des Tuplespace, in dem das Event registriert werden soll.
- ev: das zu registrierende Event (WRITE oder DELETE)
- t: das Template, das beschreibt, für welche Arten von Tuples das Event stattfinden soll
- glSeqNr: eine global eindeutige Sequenznummer, die der Distributor für dieses verteilte Event erzeugt hat. Diese global eindeutige Sequenznummer hat der

Distributor zusammen mit der lokalen Sequenznummer, unter der das Event auf dem lokalen TSServer registriert wurde, sowie mit einer Referenz auf das Callback-Objekt des Clients, das im Falle eines Events benachrichtigt werden soll, in einer Tabelle abgespeichert. Bei einem entfernt ausgelösten Event bekommt der Distributor die globale Sequenznummer mit übertragen und kann mit ihrer Hilfe über die Tabelle bestimmen, für welchen Client dieses Event registriert war und das entfernt ausgelöste Event an das entsprechende Callback-Objekt des Clients weiterleiten.

- serverId: die Adresse des RemoteTSSReceiver des lokalen TSServer, so daß der entfernte RemoteTSSReceiver weiß, von wem dieser Event-Registrierungsaufruf stammt und an wen er das Event weiterleiten soll, wenn es auftritt.

Der RemoteTSSReceiver des entfernten TSServer registriert daraufhin bei sich ein entsprechendes lokales Event „ev“ für das Template „t“ im Tuplespace „ts“ und gibt als Callback-Objekt ein lokales Forward-Objekt an. Dieses Forward-Objekt wird beim Auslösen eines Events aufgerufen und ist dafür verantwortlich das Event an den Server weiterzuleiten, der dieses entfernte Event registriert hat. Als Ergebnis auf die lokale Event-Registrierung im entfernten TSServer erhält der RemoteTSSReceiver eine lokale Sequenznummer, unter der das Event lokal auf dem entfernten TSServer registriert wurde. Die lokale Sequenznummer teilt der RemoteTSSReceiver dem Forward-Objekt zusammen mit der serverId und der global eindeutigen Sequenznummer mit. Diese Informationen legt das Forward-Objekt intern in einer Tabelle ab, so daß es bei einem ausgelösten Event anhand dieser Informationen bestimmen kann, an wen er das Event weiterleiten muß.

Abbildung 8.12 zeigt den Ablauf, wenn auf einem entfernten Server ein Event ausgelöst wird.

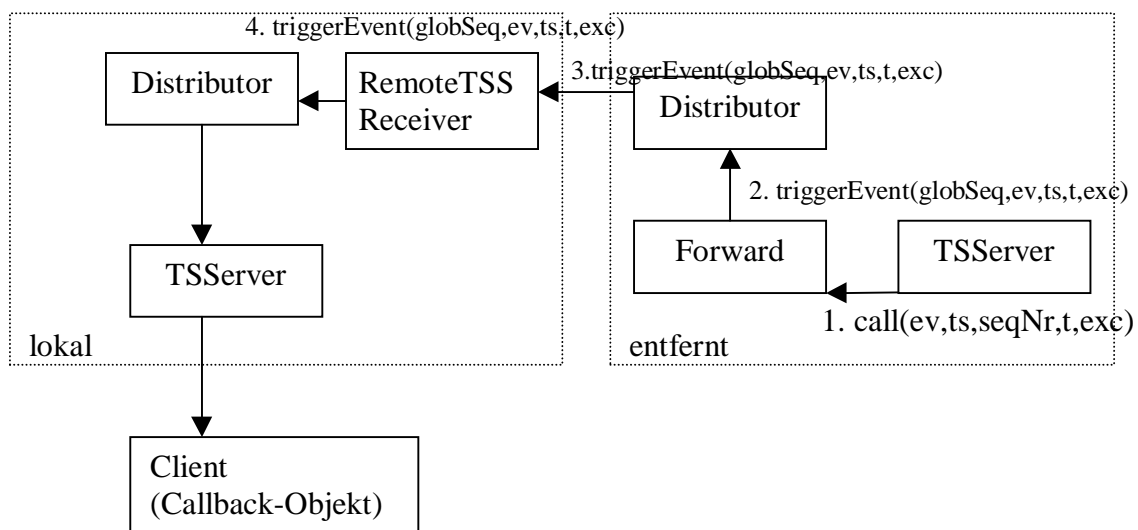


Abbildung 8.12.: Das Auslösen eines verteilten Events

Wenn auf einem entfernten Server ein Event eintritt, dann wird die call-Methode des Forward-Objektes aufgerufen:

```
public call(String eventName,String tsName,
            int sequenceNumber, Tuple tuple, boolean isException)
```

Die Sequenznummer ist hierbei die Sequenznummer, unter der das Event auf dem entfernten TSServer lokal registriert wurde. Anhand der Sequenznummer kann das Forward-

Objekt aus seiner Tabelle herausfinden, für wen das Event registriert ist, indem er anhand der Sequenznummer die globale Sequenznummer und die serverId heraussucht. Über den Distributor, das die Referenzen auf die RemoteTSSReceiver enthält, kann das Forward-Objekt das Event an den entsprechenden RemoteTSSReceiver weiterleiten, der durch die serverId angegeben ist. Hierbei übergibt es sämtliche relevante Informationen für das Event (eventName, tsName, tuple) sowie die globale Sequenznummer. Der RemoteTSSReceiver übergibt den triggerEvent-Aufruf an den Distributor, der anhand der globalen Sequenznummer aus seiner Tabelle bestimmen kann, für welchen Client der Eventaufruf ist, und das Event über den TSServer an das Callback-Objekt des Client weitergibt.

Bei der Event-Deregistrierung muß der Distributor das Event zunächst lokal und dann auf allen Nodes im In-Set abmelden. Der Ablauf ist in Abbildung 8.13 dargestellt.

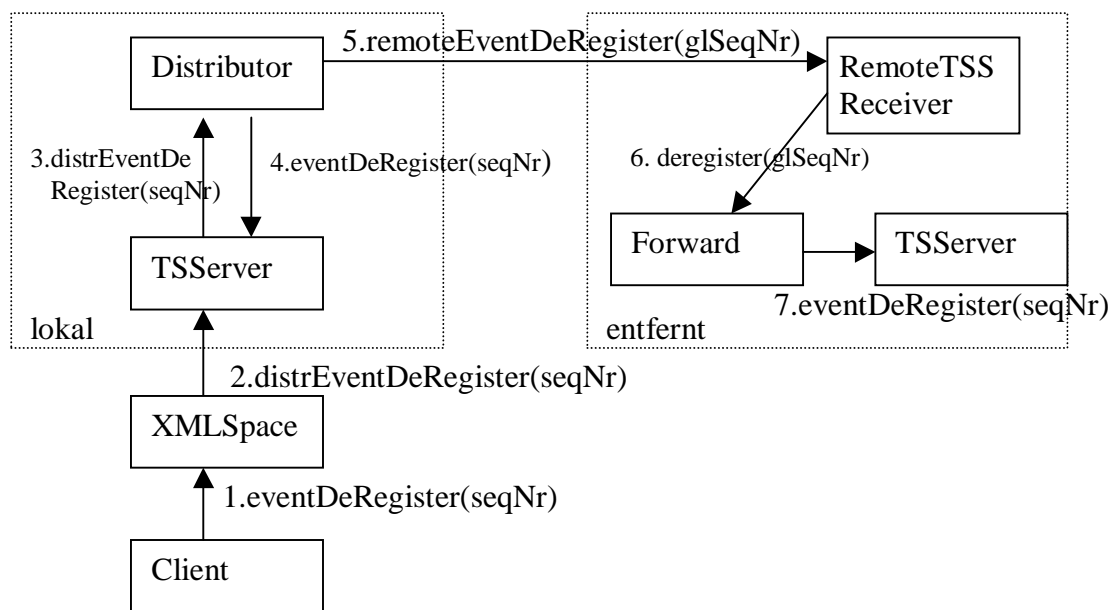


Abbildung 8.13.: Die Deregistrierung eines verteilten Events

Der Client ruft die Methode `eventDeRegister` mit der Sequenznummer auf, die er als Ergebnis auf seinen `eventRegister`-Aufruf erhalten hat, und unter der das Event auf dem lokalen TSServer registriert ist. XMLSpace leitet einen entsprechenden `distrEventDeRegister`-Aufruf an den TSServer weiter, der diesen Aufruf an den Distributor gibt. Der Distributor deregistriert zunächst lokal das Event über die normale lokale `eventDeRegister`-Methode. Dann bestimmt der Distributor über seine Tabelle anhand der Sequenznummer die global eindeutige Sequenznummer, mit dessen Hilfe er der Reihe nach auf den einzelnen Nodes im In-Set das Event deregistriert, indem er auf den einzelnen RemoteTSSReceivern die Methode `remoteEventDeRegister` mit der global eindeutigen Sequenznummer aufruft. Der RemoteTSSReceiver leitet den Deregistrierungsaufruf an das Forward-Objekt weiter, das die Informationen für die Abbildung der globalen Sequenznummer auf die lokale Sequenznummer enthält, unter der das entfernte Event auf dem entfernten Server registriert wurde, und führt die normale lokale `eventDeRegister`-Methode mit der lokalen Sequenznummer auf dem entfernten TSServer aus.

9. Integration von XMLSpaces in Workspaces

Die vorherigen Kapitel haben die Realisierung von XMLSpace beschrieben, das einen Bestandteil der Workspaces-Architektur bildet. Was als nächster Schritt noch fehlt ist die Integration von XMLSpace in die vorhandene Realisierung von Workspaces. Die Integration von XMLSpace in Workspaces ist nicht mehr Bestandteil dieser Diplomarbeit. Dieses Kapitel beschreibt nur, wie die Integration vom Prinzip her ablaufen müsste.

Die momentane Implementierung von Workspaces [Stauch] simuliert einen XMLSpace durch das Anlegen von Files in verschiedenen Verzeichnissen und Unterverzeichnissen. [Stauch] spezifiziert ein XMLSpace-Interface, das von Workspaces für den Zugriff auf das simulierte XMLSpaces verwendet wird. Das Interface hat folgenden Aufbau:

```
package workspaces;
import java.io.FileNameFilter;
import java.io.File;
import org.w3c.dom.Document;
public interface XMLSpace {

    // Read Directory with "key" type from XMLSpace
    public String[] read(String key);

    // Read Step (BUT DO NOT LOCK IT IN XMLSPACE);
    public Step getStep(String key, String name);

    // Get the DOM Document (Coordination Data) located in
    // "key"
    public Document getDocument(String key, String name);

    // Delete the Files with corresponding name (date) in the
    // "split" key.
    public void deleteSplitFiles(String date);

    // Get Step with name (LOCK FILE IN XMLSPACE !);
    public Step in(String key, String name);

    // Delete File with "key" dir in XMLSpace
    public void delete(String dir, String file);

    // put file into XMLSpace using "key" dir and "name" name,
    // delete local File
    public void out(File source, String dir, String name);

    // put this Step to XMLSpace (user key defined within the
    // Step).
    public void out(Step step);

    // init, create Dirs
    public void checkForDirs();
}
```

Für die Integration der in dieser Diplomarbeit beschriebenen Realisierung von XMLSpaces in Workspaces muß eine Wrapper-Klasse geschrieben werden, die das XMLSpace-Interface implementiert und die darin von Workspace aufgerufenen Methoden getStep usw. auf die eigentlichen Methoden in der in dieser Diplomarbeit beschriebenen

XMLSpace-Realisierung (write, waitToRead, waitToTake) abbildet (Abbildung 9.1.). Dazu müsste auch noch mal durchdacht werden, was die einzelnen Methoden vom XMLSpace-Interface, wie z.B. getStep, deleteSplitFiles usw., konkret für eine Bedeutung haben.

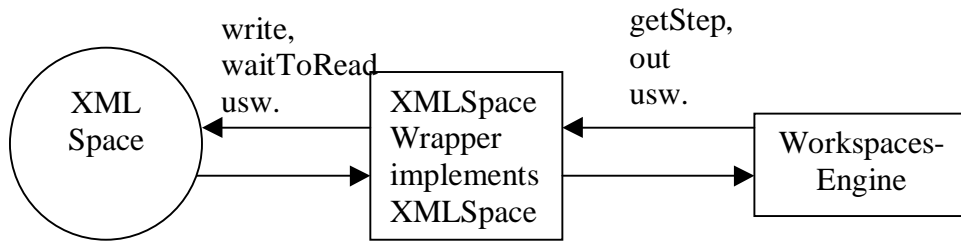


Abbildung 9.1.: Integration von XMLSpaces in Workspaces durch eine Wrapper-Klasse

10. Ausblick und offene Punkte

10.1. Fehlertoleranz

Das hier beschriebene XMLSpace realisiert zur Zeit noch keine Fehlertoleranz und geht von ausfallsicheren Servern, Netzwerken und Anwendungen aus. Im Folgenden werden kurz mögliche Konzepte für den Umgang mit Fehlern beschrieben. Die Ausführungen orientieren sich an dem Papier [Tolksdorf00c], das einen guten Überblick über die Möglichkeiten zur Realisierung von Fehlertoleranz in Linda-ähnlichen Systemen gibt und zeigt, daß die Behandlung von Fehlertoleranz in asynchronen Systemen wie Linda schwierig, oft unbefriedigend und teuer ist (die Performance nimmt ab und die Komplexität zu).

In dem Papier wird zwischen Fehlertoleranz auf Application- und auf Runtime-Level unterschieden.

Application-Level-Fehlertoleranz erlaubt es dem Programmierer einer Anwendung sicherzustellen, daß der Inhalt des Tuplespaces auch im Falle eines Auftretens eines Fehlers in der Anwendung (bis hin zum Absturz der Anwendung) konsistent bleibt. Wenn z.B. eine Anwendung ein Tuple entfernt, das zur Abstimmung zwischen verschiedenen Anwendungen dient (wie z.B. das Counter-Tuple aus dem Matrixbeispiel in Kapitel 3.2.2.), und abstürzt, ehe es das Tuple wieder zurück ins Tuplespace packen konnte, dann ist das Tuple verloren und die anderen Anwendungen, die von dem Tuple abhängig sind, können nicht weiterarbeiten, weil sie vergeblich auf das verloren gegangene Tuple warten.

Die Runtime-Level-Fehlertoleranz behandelt Fehler, die innerhalb des Servers auftreten können. Wenn zum Beispiel ein Server abstürzt, dann dürfen die von dem Server verwalteten Tuplespaces nicht verloren gehen.

Bereits das Erkennen von Fehlern bereitet Schwierigkeiten, weil in asynchronen Systemen Fehler nicht von Netzwerk-Verzögerungen unterschieden werden können. Mögliche Lösungen sind daß die Prozesse regelmäßig „I am alive“-Messages an den Server schicken und damit signalisieren, daß alles in Ordnung ist, oder zeitliche Grenzen für das Bearbeiten von Tuples festgelegt werden und beim Überschreiten der zeitlichen Grenze angenommen wird, daß die Anwendung abgestürzt ist und entsprechende Maßnahmen eingeleitet werden.

Das Papier beschreibt fünf Methoden für den Umgang mit Fehlern: 1. Ignorieren, 2. Transaktionen, 3. Mobile Koordination, 4. Checkpointing und 5. Replikation. Transaktionen und Mobile Koordination gehören zur Fehlertoleranz auf Application-Level und Checkpointing und Replikation zur Fehlertoleranz auf Runtime-Level.

Das Ignorieren von Fehler eignet sich nur für kurzlebige Anwendungen, die problemlos erneut gestartet werden können, wenn Fehler auftreten, und scheiden für XMLSpace aus.

Das Transaktionskonzept bietet dem Programmierer Methoden zum Starten, Bestätigen und Abbrechen einer Transaktion. Eine Transaktion umfaßt eine oder mehrere Zugriffe auf das Tuplespace. Zugriffe, die das Tuplespace verändern, wie „in“ und „out“ werden erst für andere sichtbar, wenn die Transaktion bestätigt wurde. Wenn eine Transaktion abgebrochen wurde, werden sämtliche während der Transaktion durchgeführten Änderungen am Tuplespace automatisch rückgängig gemacht und bleiben damit unsichtbar für die anderen Prozesse. TSpaces realisiert Transaktionen, die für Verteiltheit erweitert werden müssten.

Mobile Koordination orientiert sich an dem Konzept der Mobilen Agenten. Mobile Agenten sind Programme, die auf anderen Computer übertragen werden und dort zu Gunsten des Users ausgeführt werden. Ebenso wird bei Mobiler Koordination der auszuführende Programmcode, der die Linda-Koordinationsprimitiven enthält, auf den Server übertragen und dort ausgeführt. Da der Programmcode auf dem Server ausgeführt wird, ist es irrelevant, ob

der Application-Computer ausfällt oder nicht. Ein weiterer Vorteil ist, daß der auf den Server übertragende Code gute Performance zeigt, weil die Linda-Operationen direkt vor Ort auf Serverseite ausgeführt werden und die Kosten für die Übertragung wegfallen. Der Nachteil liegt darin, daß der Server eine Ausführungsumgebung für den Code bereitstellen muß und sich die Belastung des Servers stark erhöht, weil er nicht nur Linda-Primitiven, sondern ganze Programmabschnitte ausführen muß.

Checkpointing speichert die Tuples in den Tuplespaces zu einem bestimmten Zeitpunkt in einem persistenten Speicher und führt über alle nach dem Checkpointing durchgeführten Änderungen am Tuplespace ein Protokoll. Im Fehlerfall kann das Tuplespace durch die persistente Kopie und das Protokoll wieder hergestellt werden. TSpace verwendet Checkpointing zum Realisieren der Persistenz und stellt damit diese Art der Fehlerbehandlung automatisch zur Verfügung.

Bei Replikation wird die Fehlertoleranz durch das Replizieren von Tuplespaces auf verschiedenen Servern realisiert. Dadurch können einzelne Server ausfallen oder durch Netzwerkpartitionierungen abgetrennt sein und trotzdem weiterhin auf den verfügbaren Replikaten gearbeitet werden². Das Replizieren bedeutet jedoch auch, daß hohe Kosten für die Konsistenzerhaltung entstehen, weil die Replikate sich ständig untereinander abstimmen müssen, wenn Änderungen an dem Tuplespace vorgenommen werden. Daher scheidet volle Replikation, bei der alle Tuplespaces auf allen Servern repliziert werden, in aller Regel aus. Statt dessen wird im allgemeinen eine teilweise Replikation verwendet. Die in XMLSpace realisierte Verteiltheit arbeitet mit teilweiser Replikation. Daher können einzelne Server wegfallen und trotzdem die Tuplespaces weiterhin zugänglich bleiben. Jedoch wurde bei der Realisierung der Verteiltheit von ausfallsicheren Netzwerken und von ausfallsicheren Servern ausgegangen, die sich ordnungsgemäß untereinander an- und abmelden. Daher wurden Partitionierungen durch Netzerkerausfälle oder plötzliche Ausfälle von Servern nicht berücksichtigt. Das in dem Papier [Xu] beschriebene Verfahren bietet eine Möglichkeit zur Behandlung von Fehlern durch Netzwerk-Partitionierungen und ausgefallenen Servern durch die Verwendung von Viewpoints.

10.2. Disconnected Operation

Zukünftige Implementierungen von Workspaces sollen Disconnected Operation unterstützen. D.h. es soll möglich sein, daß eine Workspace-Engine auch Schritte ausführen kann, obwohl sie offline ist und nicht mit dem globalen XMLSpace, über den die verschiedenen Workspace-Engines miteinander koordiniert werden, direkt verbunden ist. Ein Anwendungsbeispiel ist ein Mitglied eines Programmkomitees einer Konferenz, der unterwegs im Zug oder im Flugzeug als einen Schritt eines Workflows, ein eingereichtes Konferenz-Papier begutachten will, ohne dabei mit dem XMLSpace verbunden sein zu können.

Zur Unterstützung der Disconnected Operation brauchen keine Änderungen an XMLSpace vorgenommen zu werden.

Bezüglich XMLSpace ergibt sich ungefähr folgender Ablauf für eine Disconnected Operation (der in den Abbildungen 10.1, 10.2 und 10.3 veranschaulicht ist):

- Vorbereiten zum Offline gehen: Bevor ein Benutzer offline geht muß er die notwendigen Schritte und Dokumente, die er fürs Offline arbeiten benötigt über entsprechende in()-Operationen aus dem globalen, verteilten XMLSpace

² Bei einer Netzwerkpartitionierung, bei der die Server in eine Teilmenge A und eine Teilmenge B unterteilt werden, die untereinander in der Teilmenge aber nicht mehr zwischen der Teilmenge kommunizieren und sich abstimmen können, darf nur auf der Teilmenge weitergearbeitet werden, die die Mehrheit der Replikate besitzt. Ansonsten könnte es zu einer Verfälschung der Semantik der Lindaoperatoren kommen, weil z.B. ein in() zweimal auf dasselbe Tuple ausgeführt werden könnte: nämlich einmal in der Teilmenge A und einmal in der Teilmenge B.

herausholen und in einem lokalen XMLSpace ablegen, auf den nur er Zugriff hat. (so daß er lokal über die Daten verfügt) (Abbildung 10.1.)

- Offline gehen: Der Benutzer kann nun offline gehen und auf dem lokalen XMLSpace normal operieren. Er kann die Schritte über in() auslesen, ausführen und die Ergebnisse über out() in das lokale XMLSpace ablegen. (Abbildung 10.2.)
- Online gehen: Die im lokalen XMLSpace während des Offline-Betriebes erzeugten Schritte und Dokumente werden über out()-Aufrufe auf das globale, verteilte XMLSpace übertragen und damit für andere verfügbar. (Abbildung 10.3.)

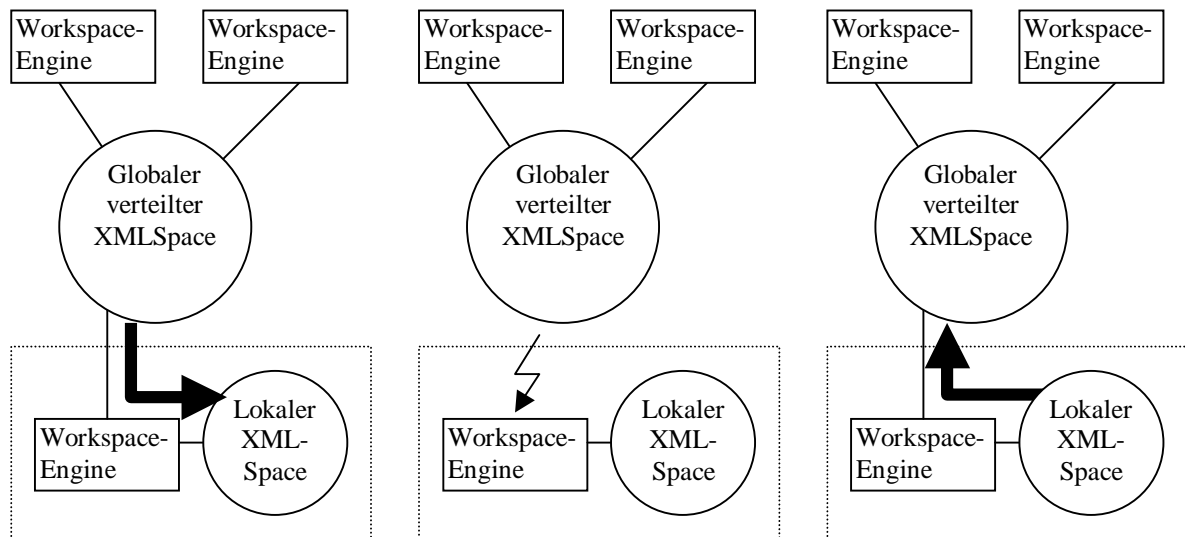


Abbildung 10.1.: Vorbereiten zum Offline gehen

Abbildung 10.2.: Offline gehen

Abbildung 10.3.: Wieder Online gehen

Solange eine Workspace-Engine offline ist kann sie nicht auf die in dieser Zeit von den anderen noch Online-arbeitenden Workspace-Engines erzeugten Dokumente zugreifen und umgekehrt. Folglich sollte die offline arbeitende Workspace-Engine beim Überschreiten irgendwelcher zeitlichen Grenzen den Benutzer darauf aufmerksam machen, daß es an der Zeit wäre wieder online zu gehen.

Die wesentlichen Erweiterungen für die Unterstützung für Disconnected Operation liegen also in der Workspace-Engine, weil die Workspace-Engine entscheiden muß, welche Schritte und Dokumente für das Offline-Arbeiten notwendig sind, damit sie aus dem XMLSpace heruntergeladen werden können.

10.3. Umstieg von DOM1 auf DOM2

XMLSpace verwendet zur Zeit XML4J Version 2.0.15 zum Parsen von XML-Dokumenten. Diese Version erzeugt nur DOM1-Dokumente und bietet keine Unterstützung für DOM2. Ab XML4J 3 werden auch DOM2-Dokumente unterstützt.

Der Wesentliche Unterschied von DOM2 zu DOM1 ist die Unterstützung von Namespaces und die Möglichkeit für das Abfragen der SystemId und der PublicId in der <!Doctype...>-Deklaration des XML-Dokumentes. Beides wird in DOM1 nicht angeboten.

Damit das Doctype-Matching auch das Matching über die SystemId und die PublicId in der Doctype-Deklaration unterstützen kann und damit Namespaces prinzipiell beim Matching unterstützt werden können, ist es notwendig von DOM1 auf DOM2 und damit von XML4J 2 auf XML4J 3 umzusteigen.

Für den Umstieg von XML4J 2 auf XML4J 3 müssen statt des `com.ibm.xml.parsers.NonValidating.DOMParser` der `org.apache.xerces.parser.DOMParser`

verwendet werden und die Matchingrelationen wie z.B. DoctypeMatch.java müssten angepasst werden.

XMLSpaces bietet über den Konstruktor `XMLDocField(org.w3c.dom.Document doc)` die Möglichkeit direkt DOM-Document-Objekte in einem XMLDocField abzulegen. Diese DOM-Document-Objekte wurden außerhalb von XMLSpaces mit anderen Parsern (evtl. mit XML4J 3 vielleicht aber auch mit XML4J 2 oder einem ganz anderen Parser) erzeugt. Daher besteht die Möglichkeit, daß die über diesen Konstruktor im XMLDocField abgelegten DOM-Document-Objekte vom Typ DOM1 oder vom Typ DOM2 sind. Es kann also vorkommen, daß in einem XMLSpace DOM1- und DOM2-Dokumente vermischt sind. Dies muß beim Aufruf von DOM2-Methoden innerhalb von XMLSpace berücksichtigt werden (z.B. beim Doctype-Matching).

Die in DOM2 spezifizierten Interfaces enthalten die Methoden von DOM1 und einige zusätzliche Methoden. DOM2-Objekte kommen also problemlos mit Programmcode klar, der für DOM1-Objekte geschrieben ist. Jedoch gibt es Probleme, wenn ein Programm, das DOM2-spezifische Methoden verwendet, auf ein DOM1-Objekt angewendet wird. So gibt es zum Beispiel in DOM2 die Attribute `DocumentType.publicId` oder `Node.namespaceURI`. In DOM1 gibt es diese Attribute nicht.

In DOM1 und DOM2 gibt es leider keine Methode, mit der man überprüfen kann, ob es sich um ein DOM1- oder ein DOM2-Dokument-Objekt handelt. Ein Programmcode, der sowohl mit DOM1 als auch mit DOM2-Objekten umgehen können soll, muß also vorher durch Reflection-Methoden überprüfen, ob die Methoden überhaupt verfügbar sind oder die Exceptions, die beim nicht Vorhandensein einer aufgerufenen Methode, ausgelöst werden, abfangen und entsprechend behandeln.

10.4. Unterstützung von Namespaces

Namespaces dienen dazu Element- und Attributnamen global eindeutig zu machen. Wie in Kapitel 6.2.7. beschrieben ist es sinnvoll Namespaces in XML-Matchingrelationen zu unterstützen, um die zu matchenden XML-Dokumente eindeutiger zu beschreiben.

Vom Prinzip her unterstützen alle im Kapitel 6 beschriebenen Matchingrelationen Namespaces. Das Problem liegt jedoch darin, daß XMLSpace DOM1 verwendet und Namespaces erst ab DOM2 unterstützt werden. Daher ist es für die Unterstützung von Namespaces notwendig wie im Kapitel 10.3. beschrieben von DOM1 auf DOM2 umzusteigen.

10.5. Security

TSpace bietet die Möglichkeit des Zugriffsschutz auf den Tuplespace-Server durch die Verwendung von Benutzernamen und Passwörter, die ein Client beim ersten Zugriff auf den Tuplespace-Server angeben muß um sich zu authentifizieren. Dadurch wird sichergestellt, daß nur Clients Zugriff haben, die bei dem Tuplespace-Server registriert sind und eine Berechtigung für den Zugriff haben.

Darüber hinaus bietet TSpace die Möglichkeit für Tuplespaces Access Control Lists (ACLs) anzugeben, über die festgelegt werden kann, welche Operationen ein Client auf einem Tuplespace ausführen darf. So ist es z.B. möglich festzulegen, daß eine bestimmte Gruppe von Clients nur lesend, aber nicht schreibend auf ein Tuplespace zugreifen darf.

Die für XMLSpace zu realisierende Security könnte auf den von TSpace bereitgestellten Schutzmechanismen aufbauen und sie um die speziellen Anforderungen von XMLSpace erweitern.

So erfordert zum Beispiel Workspace, daß die ACLs nicht nur auf Tuplespace-, sondern auch auf Tuple-Ebene erfolgen können, da es sicherlich Anwendungsfälle gibt, in denen alle Benutzer die gleichen Zugriffsrechte (Erlaubnis für in, out und rd) auf das

Tuplespace haben, aber auf Tuple-Ebene nur bestimmte Benutzer ein in oder rd ausführen dürfen.

Zusätzlich müssten Sicherheitsrisiken, die durch die Verteiltheit von XMLSpace entstehen, wie z.B. das Abhören von Daten bei der Übertragung, behandelt werden. Mögliche Lösungen wären z.B. die Verwendung von Verschlüsselungsalgorithmen.

Anhang A: Die Konfigurationsdatei von TSpace

Beim Starten eines TSServers können eine Reihe von Optionen mit angegeben werden:

```
java com.ibm.tspaces.server.TSServer [options]
```

Diese Optionen können auch beim Starten des XMLSpaceServers angegeben werden, der intern den TSServer startet und die angegebenen Optionen an den TSServer weiterreicht. Wichtige Optionen im Bezug auf XMLSpaces sind :

<code>[-d checkpointDirectory]</code>	schaltet die Persistenz von Tspace ein und speichert die Daten des Tuplespaces in dem angegebenen checkpointDirectory-Verzeichnis.
<code>[-i interval]</code>	Spezifiziert alle wieviel Minuten ein komplettes Checkpointing gemacht werden soll. Zwischen dem Checkpointing werden lediglich Änderungen Mitprotokolliert. Auf diese Art ist jederzeit eine komplette Kopie des Tuplespace persistent vorhanden.
<code>[-b]</code>	Erzeugt einen leeren Tuplespace, d.h. beim Starten werden nicht die im Checkpointing-Directory abgelegten Daten berücksichtigt.
<code>[-p port#]</code>	Spezifiziert die Port-Number, über die der TSServer Client-Aufrufe erwartet [der Default ist 8200]

Zum Beispiel startet

```
java XMLSpacesServer -d kopie -i 10 -p 8204
```

einen XMLSpaceServer auf dem Port 8204, der alle 10 Minuten eine Kopie der Tuplespaces in dem Verzeichnis `kopie` abspeichert. Wenn keine Optionen angegeben werden, ist das Tuplespace standardmäßig nicht persistent und läuft über den Port 8200.

Statt die einzelnen Optionen beim Serverstart jedes mal von Hand anzugeben, kann man auch eine Konfigurationsdatei verwenden, das über `-c ConfigFileName` angegeben wird, z.B.

```
java XMLSpacesServer -c xmlspaces.cfg
```

Standardmäßig sollte für XMLSpace die mitgelieferte Konfigurationsdatei `xmlspaces.cfg` verwendet werden. Diese Konfigurationsdatei entspricht der von TSpace mitgelieferten Konfigurationsdatei `tspaces.cfg`, das die Standardeinstellungen für einen TSServer enthält, ist aber den XMLSpaces-Anforderungen angepasst. Zum Beispiel ist in `xmlspaces.cfg` die Unterstützung der von TSpace bereitgestellten Security standardmäßig abgeschaltet, weil XMLSpaces zur Zeit keine Security unterstützt. Die Konfigurationsdatei `xmlspaces.cfg` enthält folgende Einträge, wobei die Kommentare aus der Original-`tspaces.cfg`-Datei übernommen sind:

```
# tspaces.cfg
# This is the config file for TSServer that is distribute
# with the T Spaces package
#-----
```

```

# The [Server] section contains general specifications for the
# TSServer.
#-----

[Server]

# The port that it listens to for requests.
Port                =          8200

#-----
# The pathname of the directory used for checkpointing
# If not specified, then no checkpointing will be done and
# all TupleSpaces are transient. Can be overridden with the
# -d dir operand on the cmd line.
#-----
#CheckpointDir     =  c:/tmp/checkpoint
# The interval between dumping the checkpoint data (in
# Minutes)
CheckpointInterval = 10.0

# The number of updates before checkpoint is requested.
# (-1 to disable)
checkpointWriteThreshold = -1

# Specify the interval (in seconds) between checking for
# Deadlocked threads
DeadLockInterval = 15

#-----
# The [AccessControl] section contains parameters that are
# needed for AccessControl
#-----
[AccessControl]

# If CheckPermissions is set false, than no access checking
# will be done.
CheckPermissions                =          false
...

```

Neben den von TSpace bereitgestellten Optionen, die in XMLSpaceServer wiederverwendet werden, gibt es die XMLSpaceServer spezifische Option `-distr DistrConfigFile`, z.B.

```
java XMLSpaceServer -distr PartialRepl.cfg [sonstigeOptionen]
```

Hierbei entsprechen die sonstigen Optionen den oben beschriebenen Standardoptionen von TSpaces, wie z.B. `-p PortNumber`, `-d CheckpointDirectory` usw.

Die Datei `DistrConfigFile` (in diesem Fall `PartialRepl.cfg`) enthält die notwendigen Informationen für die Unterstützung der Verteilung in XMLSpace. Der erste Eintrag ist immer `STRATEGY=DISTRIBUTORKLASSE`, der angibt, welche Verteilungsstrategie in XMLSpace verwendet werden soll. Die anderen Einträge in der Konfigurationsdatei sind von dem verwendeten Distributor-Objekt, das die Verteilungsstrategie realisiert, abhängig.

Zum Beispiel enthält das `PartRepl.cfg` folgende Einträge
`STRATEGY=PartialRepliationDistributor`
`RECEPTION=www.cs.tu-berlin.de/~glaubitz/rezeption`

Der Reception-Eintrag gibt hierbei die Adresse des Rezeptionisten an, über den die Anmeldung in der Gitterstruktur erfolgt.

Anhang B: Verwendete Software

Name	Hersteller	weitere Online-Informationen unter...	Verwendungszweck
Java 2 SDK, Version 1.2	Java Software of Sun Microsystems, Inc.	http://java.sun.com	<ul style="list-style-type: none"> - Basisprogrammiersprache - Realisierung von Verteiltheit über RMI
TSpaces, Version 2.1.1	IBM (Almaden Research Center)	http://www.almaden.ibm.com/cs/TSpaces und unter http://www.alphaworks.ibm.com/tech/tspaces	<ul style="list-style-type: none"> - Implementierungsplattform von XMLSpaces. - Realisiert das Linda-Konzept in XMLSpaces. - Realisierung der XML-Erweiterung von Linda.
XML4J, version 2.0.15	IBM	http://www.alphaworks.ibm.com/tech/	<ul style="list-style-type: none"> - Zum Parsen von XML-Dokumenten und Umwandeln in DOM-Document-Objekte. - Zum Realisieren des DTD-Matchings.
Xalan-Java, version 1.2.	Apache Software Foundation	http://xml.apache.org/xalan/index.html	<ul style="list-style-type: none"> - XPath-Engine zum Realisieren des XPath-Matchings in XMLSpace.
GMD-IPSI-XQL-Engine, Version 1.0.2	GMD – Forschungszentrum Informationstechnik GmbH	http://xml.darmstadt.gmd.de/xql/	<ul style="list-style-type: none"> - XQL-Engine zum Realisieren des XQL-Matchings in XMLSpace.
XML-QL, version 0.9	AT&T Corp.	http://www.research.att.com/~mff/xmlql/doc	<ul style="list-style-type: none"> - XMLQL-Engine zum Realisieren des XMLQL-Matchings in XMLSpace.

Anhang C: Interfaces

Dieser Anhang gibt noch einmal die wichtigsten Interfaces von XMLSpaces wieder: XMLMatchable und Distributor.

Das Interface XMLMatchable muß von Klassen implementiert werden, die XML-Matchingrelationen realisieren:

```
package xmlspaces;

import java.io.*;
import org.w3c.dom.Document;

/**
 * Das XMLMatchable-Interface muss von allen Klassen,
 * die eine XMLMatchingroutine für XMLSpaces
 * realisieren, implementiert werden.
 */
public interface XMLMatchable extends Serializable {

    /**
     * Diese Methode wird von XMLDocField.matches(Field f)
     * aufgerufen, um zu überprüfen, ob ein XML-Dokument
     * mit der im XMLMatchable-Objekt angegebenen
     * Matchingrelation matcht.
     *
     * @param xmlAsDoc Das zu matchende XML-Dokument als DOM-Document-
     * Objekt.
     * @return True, wenn die im XMLMatchable-Objekt angegebene
     * Matchingrelation mit xmlAsDoc matcht, ansonsten false.
     */
    public boolean xmlMatch(Document xmlAsDoc);

}
```

Das Interface Distributor muß von Klassen implementiert werden, die eine Verteilungsstrategie realisieren wollen. Es enthält zum einen die Methoden, die vom TSServer aufgerufen werden, um den Distributor eine verteilte Methode ausführen zu lassen, und Methoden zum Anmelden und Abmelden bei anderen Servern im verteilten Tuplespace:

```
package xmlspaces;

/**
 * Das Distributor-Interface muss von Klassen implementiert
 * werden, die eine Verteilungsstrategie für XMLSpace
 * realisieren. Die vom Client aufgerufenen verteilten
 * Methoden werden vom TSServer über den TSFDistributedBasic
 * und den TSHDistributedBasic an das Distributor-Objekt
 * weitergeleitet, das für die Ausführung der verteilten
 * Methode verantwortlich ist.
 */
public interface Distributor{
    /**
     * Methoden zum An- und Abmelden im verteilten
     * Tuplespace.
     */

    public SuperTuple register(Hashtable[] args);
}
```

```

public SuperTuple deRegister();

//*****
// Lindaspezifische Methoden
// ts:          auf welchem Tuplespace die Methode angewendet
//              werden soll.
// argTuple:    Argument-Tuple für das Kommando, z.B.
//              das zum Tuplespace hinzuzufügende Tuple bei
//              einem write.
// clientId:    Die Id des Clients, der den Befehl aufgerufen hat.
// communicator: Objekt, über das eine Antwort an den Client
//              zurückgeschickt werden kann.
//*****
public SuperTuple distributedWrite(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

public SuperTuple distributedWaitToTake(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedWaitToRead(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedTake(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

public SuperTuple distributedRead(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

//*****
// sonstige nicht-Lindaspezifische Methoden.
// Die gleichen Argumente wie bei den linda-spezifischen Methoden.
//*****

public SuperTuple distributedReadTupleById(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedTakeTupleById(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedMultiwrite(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedUpdate(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

public SuperTuple distributedMultiupdate(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedDeleteTupleById(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedScan(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

public SuperTuple distributedConsumingScan(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedDeleteAll(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedDelete(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

```

```
public SuperTuple distributedCount(String ts, SuperTuple argTuple,
    String clientID, TSResponse communicator);

//*****
// Methoden zum An- und Abmelden von verteilten
// Events.
// Die gleichen Argumente wie bei den linda-spezifischen Methoden.
//*****

public SuperTuple distributedEventRegister(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

public SuperTuple distributedEventDeRegister(String ts, SuperTuple
    argTuple, String clientID, TSResponse communicator);

}
```

Literaturverzeichnis

- [Abiteboul97] Serge Abiteboul. Querying Semi-Structured Data. In F.N. Afrati and P. Kolaitis, editors, *Proceedings of the International Conference on Database Theory – (ICDT)'97, 6th International Conference*, number 1186 in LNCS, pages 1-18. Springer, 1997.
- [Abraham] John Abraham, Hai Le, Chris Cedro. XML Repository In TSpaces & UIA Event Notification Application, January 1999.
<http://www.cse.scu.edu/projects/1998-99/project19/>
- [Ahuja86] Sudhir Ahuja, Nicholas Carriero and David Gelernter. Linda and Friends. *IEEE Computer*, vol.19, number 8, pages 26-34, August 1986
- [Anderson] B. G. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J. P. Banatre and D. Le Metayer, editors, *Proceedings of Research Directions in High Level Parallel Programming Languages*, number 574 in LNCS, pages 93 – 109. Springer, 1991.
- [Aranya] Amarilis Macedo Aranya. Ein graphisches Prozessmodellierungstool für ein XML-basiertes, verteiltes WfMS. Diplomarbeit, TU-Berlin, 2000.
- [Banville] M. Banville. SONIA: an Adaptation of Linda for Coordination of Activities in Organizations. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models First International Conference, COORDINATION '96*, number 1061 in LNCS, pages 57 - 74. Springer-Verlag, 1991.
- [Bjornson] Robert Bjornson, Nicholas Carriero and David Gelernter. From weaving threads to untangling the web: a view of coordination from Linda's perspective. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, number 1282 in LNCS, pages 1 – 17. Springer, 1997.
- [Cabri00] G. Cabri, L. Leonardi, and F. Zambonelli. XML Dataspaces for Mobile Agent Coordination. In *Proc. of the 2000 ACM Symposium on Applied Computing*, pages 181-188, 2000
- [Carriero86] Nicholas Carriero, David Gelernter and Jerry Leichter. Distributed Data Structures in Linda. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236 – 242. ACM, 1986.
- [Carriero86b] Nicholas Carriero and David Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, volume 4, number 2, pages 110-129, 1986.
- [Carriero88] Nicholas Carriero and David Gelernter. Applications Experience with Linda. In *Proceedings of the ACM/SIGPLAN PPEALS 1988, SIGPLAN Notices*, volume 23, number 9, pages 173-187, 1988.
- [Carriero89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, volume 32, number 4, pages 444-458, 1989.
- [Carriero92] N. Carriero and D. Gelernter. Data Parallelism and Linda. In U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 145 – 159. Springer, 1992.

- [Carriero94] Nicholas Carriero and David Gelernter and Timothy G. Mattson and Andrew H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, volume 20, number 4, pages 633-655, 1994.
- [Coord96] P. Ciancarini, C. Hankin, C., editors. *Coordination Languages and Models First International Conference, COORDINATION '96*, number 1061 in LNCS. Springer-Verlag, 1996.
- [Coord97] D. Garlan, D. Le Metayer, editors. *Coordination Languages and Models Second International Conference, COORDINATION'97*, number 1282 in LNCS. Springer-Verlag, 1997.
- [Coord99] P. Ciancarini, A.L. Wolf, editors. *Coordination Languages and Models Third International Conference, COORDINATION'99*, number 1594 in LNCS. Springer-Verlag, 1999.
- [DOM1] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification, Version 1.0. W3C Recommendation, 1. October 1998. <http://www.w3c.org/TR/REC-DOM-Level-1>
- [DOM2] World Wide Web Consortium. Document Object Model (DOM) Level 2 Specification, Version 1.0. W3C Candidate Recommendation, 10. May 2000. <http://www.w3c.org/TR/DOM-Level-2>
- [Faasen] C. Faasen. Intermediate Uniformly Distributed Tuple Space on Transputer Meshes. In J. P. Banatre and D. Le Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 157 – 173 . Springer, 1991.
- [Gelernter85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, volume 7, number 1, pages 80 – 112, 1985.
- [Gelernter92] David Gelernter and Nicholas Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, volume 35, number 2, pages 97 – 107, 1992.
- [Hupfer91] S. Hupfer and D. Kaminsky and N. Carriero and D. Gelernter. Coordination Applications of Linda. In J. P. Banatre and D. Le Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 187 – 194. Springer, 1991.
- [Malone94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, volume 26, number 1, pages 87 – 119, March 1994.
- [Mattson92] T.G.Mattson, R.Bjornson, D. Kaminsky. The C-Linda Language for Networks of Workstations. In *Workshop on Cluster Computing, Florida State University*, 1992.
- [Moffat] David Moffat. XML-Tuples and XML-Spaces, V0.7, 25 March 1999. <http://uncled.oit.unc.edu/XML/XMLSpaces.html>
- [Papadopoulos98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, volume 46, pages 329 – 400, Academic Press, 1998.
- [SQL] ISO. *ISO/IEC 9075:1992: Information Technology – Database Languages – SQL*. International Organization of Standardization, 1992.
- [Stauch] Marc Stauch. Design and Implementation of a System for Distributed Workflows using XML / XSL. Diplomarbeit, TU-Berlin, 2000.
- [Tolksdorf95] Robert Tolksdorf. *Coordination in Open Distributed Systems*, VDI-Fortschrittsberichte, Reihe 10: Informatik/Kommunikationstechnik, Nr. 362. VDI Verlag, 1995.

- [Tolksdorf00a] Robert Tolksdorf. Coordinating Work on the Web with Workspaces. In *Proceedings of the IEEE Ninth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises WET ICE 2000*. IEEE Computer Society Press, 2000.
- [Tolksdorf00b] Robert Tolksdorf. Coordination Technology for Workflows on the Web: Workspaces. In *Proceedings of the Fourth International Conference on Coordination Models and Languages COORDINATION 2000*, LNCS-Reihe. Springer-Verlag, 2000.
- [Tolksdorf00c] Robert Tolksdorf, Antony Rowstron. Evaluating Fault Tolerance Methods for Large-scale Linda-like Systems. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, 2000
- [TSpace] P. Wyckoff and S. W. McLaughry and T. J. Lehman and D. A. Ford. T Spaces. *IBM Systems Journal*, volume 37, number 3, pages 454 - 474, 1998.
- [WfMC98a] Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model, 1998. <http://www.wfmc.org>
- [WfMC98b] Workflow Management Coalition. Workflow and Internet: Catalysts for Radical Change. WfMC White Paper, 1998. <http://www.wfmc.org>.
- [W3C99] World Wide Web Consortium. Namespaces in XML. W3C Recommendation, 14. January 1999. <http://www.w3.org/TR/REC-xml-names>
- [W3C00a] World Wide Web Consortium. XML Query Requirements. W3C Working Draft, 15. August 2000. <http://www.w3c.org/TR/xmlquery-req>.
- [W3C00b] World Wide Web Consortium. XML Query Data Model. W3C Working Draft, 11 May 2000. <http://www.w3c.org/TR/query-datamodel>
- [XML] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. W3C Recommendation, 1998. <http://www.w3c.org/TR/REC-xml>
- [XMLQL] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium, 19-August-1998. <http://www.w3.org/TR/NOTE-xml-ql>
- [XPath] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16. November 1999. <http://www.w3c.org/TR/xpath>
- [XPointer] World Wide Web Consortium. XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, 7 June 2000. <http://www.w3.org/TR/xptr>
- [XQL] Jonathan Robie, Joe Lapp, David Schach. XML Query Language (XQL). Proposal to the W3C QL98 Workshop. <http://www.w3.org/TandS/QL/QL98/pp/xql.html>
- [Xu] A. Xu and B. Liskov. A Design for a Fault-Tolerant, Distributed Implementation of Linda. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS) '89*, pages 199 – 207. IEEE Computer Society Press, 1989.