A Web Service Discovery Architecture and a Peer-to-Peer Database Framework

Alexander Bilke
Technische Universität Berlin
bilke@cs.tu-berlin.de
Betreuer: Wolf-Ulrich Raffel

14. Juni 2003

Kapitel 1

A Web Service Discovery Architecture

1.1 Motivation

Die in dem Vortrag beschriebene Dissertation[1] bearbeitet das Problem der Suche nach Informationen, Ressourcen und Diensten in großen verteilten Internetsystemen, die mehrere administrative Domänen umfassen.

Hintergrund ist die Entwicklung sogenannter *virtueller Organisationen*. Dies sind Gruppen bzw. Teams von Personen, die über Organisationsgrenzen hinaus kooperieren. Die Verteilung von Informationen zwischen den Gruppenmitgliedern muss dabei unterstützt werden. Die flexible, sichere und koordinierte Verteilung von Informationen innerhalb dynamischer Gruppen von Personen, Institutionen und Ressourcen ist das Ziel von Grid-Technologie.

1.2 Struktur der Architektur

In vielen bekannten Architekturen zum Suchen und Ausführen von Diensten gibt es drei beteiligte Elemente: einen *Client*, einen *Dienstanbieter* und eine *Registry*. Diese Elemente werden im folgenden vorgestellt und die relevanten Schritte zur Nutzung eines Dienstes erläutert.

1.2.1 Schritte zum Finden und Ausführen von Diensten

In der Architekturbeschreibung sind acht Schritte beschrieben: *Description, Presentation, Publication, Request, Discovery, Brokering, Execution und Control.*

1. **Description.** Als *Description* wird der Vorgang den Definierens von Metadaten, die dem Auffinden des Dienstes dienen, verstanden. Als Beschreibungssprache wird der Standard *Web Service Description Language* (WSDL) verwendet. In der vorliegenden Dissertation wird eine vereinfachte Version dieses Standards, die *Simple WSDL* benutzt. Es ist jedoch davon auszugehen, dass in einem laufenden System der volle Standard zu benutzen ist.

Die Beschreibung eines Dienstes enthält die *Schnittstellen*, die der Dienst anbietet. Eine solche Schnittstelle besteht aus *Operationen*, die wiederum *Argumente* besitzen. Zusätzlich zur Angabe der Schnittstellen wird in einer Dienstbeschreibung angegeben, wie die Operationen und Argumente an ein konkretes Netzwerkprotokoll gebunden werden.

2. **Presentation.** Nachdem eine Dienstbeschreibung erstellt wurde, muss sie verfügbar gemacht werden, d.h. potentielle Clients müssen die Möglichkeit haben, sie abzurufen. Ein wichtiger Punkt ist hier die Integration von Altsystemen, die ohne große Änderungen des existierenden Dienstes vonstatten gehen soll. Aus diesem Grund wird die Kernfunktionalität des Dienstes von der Präsentation der Beschreibung getrennt. Als ID für eine Beschreibung wird eine URL und als Abfragemechanismus HTTP(S) gewählt. Die auf eine Dienstbeschreibung verweisende URL wird als *service link* bezeichnet. Es wird außerdem vorgeschlagen, dass der service link Bestandteil der Dienstbeschreibung wird, welche zu diesem Zweck um die Schnittstelle *Presenter* erweitert werden sollte.

3. **Publication.** In diesem Schritt wird die Existenz einer Dienstbeschreibung potentiellen Clients bekannt gemacht. Dies geschieht, indem beim Start eines Dienstes der service link an eine Registry geschickt wird. Vor dem Herunterfahren eines Dienstes sollte er sich entsprechend bei der Registry abmelden. Clients erhalten durch Abfragen der Registry die gesuchten service links, die sie benutzen können, um die aktuelle Dienstbeschreibung zu erhalten. Gründe für das Speichern von Zeigern auf die Beschreibungen und nicht der Beschreibung an sich sind Sicherheit, Vertraulichkeit, Konsistenz und möglicherweise Effizienz.

4. **Request.** Clients formulieren Anfragen (requests). Solche Anfragen sind hierarchische Strukturen, an deren Ende Operationen stehen. Ressourcen sind Dinge, die für einen bestimmten Zeitraum genutzt werden können, und die möglicherweise erneuerbar sind. Ressourcen werden durch Operationen der Schnittstellen eines Dienstes zur Verfügung gestellt, d.h. Operationen konsumieren Ressourcen.

5. **Discovery.** In diesem Schritt werden die Dienste gefunden, welche die Operationen, die in der im vorigen Schritt erstellten Anfrage enthalten sind, anbieten. Dabei werden ein oder mehrere *candidate services* gefunden. Oftmals ist es notwendig, zur Erfüllung einer Anfrage mehrere Dienste zu kombinieren.

6. **Brokering.** Nach dem Auffinden potentieller Dienste muß ein Ausführungsplan erstellt werden. Dabei können verschiedene Techniken eingesetzt werden, um die zu benutzenden Dienste auszuwählen.

7. **Execution.** Der erstellte Ausführungsplan muß nun umgesetzt werden, d.h. die Operationen der ausgewählten Dienste werden aufgerufen. Dafür werden die in den Dienstbeschreibungen angegebenen Protokolle genutzt.

8. **Control** Methoden können synchron oder asynchron aufgerufen werden. Der asynchrone Aufruf bietet entscheidene Vorteile. So kann beispielsweise der Dienst weiterarbeiten, ohne dass der Client mit ihm verbunden ist. Der

Client kann später das Ergebnis abrufen, wenn er wieder verfügbar ist. Außerdem kann der Lebenszyklus des Dienstes überwacht und kontrolliert werden. Wegen der höheren Komplexität der Implementierung asynchroner Aufrufe wird für die meisten Dienste jedoch ein synchrones Aufrufmodell gewählt.

1.2.2 Zusammenspiel der einzelnen Elemente

In der vorliegenden Arbeit wird hauptsächlich auf Presentation, Publication und Discovery eingegangen. Die Zusammenhänge zwischen den Elementen der Architektur ist schematisch in Abbildung 1.1 dargestellt. Ein Dienstanbieter tritt sowohl als Presenter und als Publisher auf. Über seine Presenter-Schnittstelle bietet er seine Beschreibung nach außen an. Als Publisher schickt er seinen service link an die Registry, welche diesen in ihrer Datenbank speichert. Der Mediator[1] übersetzt zwischen den generischem Datenmodell und der internen Datenmodell des Dienstanbieters. Ein Client sucht nach bestimmten Diensten, indem er eine Anfrage an die Registry schickt. Diese liefert ihm als Ergebnis service links, über die er die aktuelle Dienstbeschreibung von dem Anbieter bekommen kann.

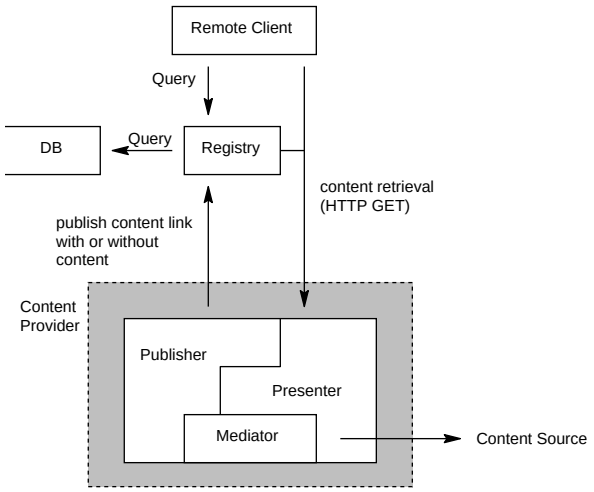

Abbildung 1.1: Elemente der Architektur(Quelle: [1])

1.2.3 Das Generische und das Dynamische Datenmodell

Nach dem *Generischen Datenmodell* ist die Gesamtheit der Datentupel auf verschiedene Knoten im Netz verteilt. Diese Tupel sind XML-basiert. Ein Schema ist dabei optional. Anfragen bekommen als Eingabe und liefern als Ausgabe Instanzen dieses Generische Datenmodells, d.h. sie operieren auf XML-Daten.

Das *Dynamische Datenmodell* ist eine Verfeinerung des Generischen Datenmodells, das der dynamischen Natur der hier betrachteten verteilten Systeme

[1] Die Bezeichnung Mediator ist eher unglücklich, da dieser Begriff im Bereich verteilter, heterogener Datenbanksysteme eine andere Bedeutung besitzt. *Wrapper* wäre hier die bessere Wahl gewesen.

Tabelle 1.1: Dynamic Data Model

Link	Context	Type	TS1	TS2	TS3	Metadata	Content
http://sched001.cern.ch/ getServiceDescription	Parent	Service	10	20	30	<owner name = "http://cms.cern.ch" />	<service> A </service>

Rechnung trägt. Wie in Tabelle 1.1 schematisch dargestellt, werden neben dem eigentlichen Inhalt zusätzliche Informationen angegeben. Die Bedeutung der Zusatzinformationen wird später erläutert. Auch das Dynamische Datenmodell ist XML-basiert, was im Gegenzug bedeutet, dass Anfragen sich neben der eigentlichen Dienstbeschreibung auch auf die Zusatzinformationen beziehen können.

1.3 Die Hyper Registry

Hauptaugenmerk dieses Abschnitts liegt auf der Hyper Registry, welche die Datentupel verwaltet. Dabei wird besonders der Vorgang der Publizierung von Dienstbeschreibungen und Anfragen an die Registry betrachtet.

1.3.1 Publication

Bei der Publizierung von Dienstbeschreibung wird der Registry eine Instanz des Dynamischen Datenmodells übergeben. Dabei werden, wie in Abschnitt 1.2.3 beschrieben, neben dem eigentlichen Inhalt — in diesem Fall der Dienstbeschreibung — auch zusätzliche Informationen übertragen. Das sind im einzelnen ein content link, context, type, TS1 - TS3 und metadata.

Content link. Der content link ist ein Zeiger auf den eigentlichen Inhalt. Er wird in Form einer HTTP(S) URL angegeben, mit der ein Client den aktuellen Inhalt abfragen kann. Im Kontext der Web Service Discovery Architecture handelt es sich bei dem content link um einen *service link*.

Context. Der Kontext beschreibt, warum der Inhalt veröffentlicht wurde und wie er zu benutzen ist. Es ist oft nicht sinnvoll, Kontextinformationen direkt im Inhalt anzugeben, weil der gegebene Inhalt kann in mehreren Kontexten in verschiedenen Registries publiziert werden.

Zeitstempel TS1 - TS3. Eine Beschreibung der Zeitstempel folgt in Abschnitt 1.3.2.

Metadata. Hier können weitere Zusatzinformationen abgelegt werden, die nicht durch die bisher diskutierten Elemente abgedeckt sind. Dieses Element muss ebenfalls ein XML-Dokument sein.

Jeder Tupel kann innerhalb einer Registry durch einen eindeutigen Schlüssel identifiziert werden. Dieser besteht aus dem Link und dem Kontext.

1.3.2 Caching und Soft State Publication

Caching des Inhalts, also der Dienstbeschreibungen, ist wichtig, um die Effizienz der Clients zu gewährleisten. Wenn die Dienstbeschreibung direkt als Ergebnis

Tabelle 1.2: Zeitstempel

Zeitstempel	Semantik
TS1	Zeitpunkt der letzten Änderung des Inhalts beim Dienstanbieter
TC	Zeitpunkt der letzten Änderung des Tupels (z.B. durch Vermittler)
TS2	Erwarteter Zeitraum, in dem der Inhalt beim Anbieter gültig ist
TS3	Erwarteter Zeitraum, in dem der content link beim Anbieter gültig ist

einer Anfrage an die Registry geliefert wird, muss der Client dafür keine Verbindung zum Dienstanbieter aufbauen. Eine Hyper Registry kann, muss aber nicht, Caching unterstützen. Beim Caching kann es aber zu Unstimmigkeiten kommen. Um dies zu verhindern, wird die Strategie *server invalidation* verfolgt. Danach muss ein Dienstanbieter die Registry benachrichtigen, wenn eine Änderung erfolgt ist. Eine Registry kann dabei eine von drei Formen annehmen.

Pull Hyper Registry. Wenn der Anbieter eine Änderung durchführt, benachrichtigt er die Registry, welche dann den aktuellen Inhalt abrufen kann.

Push Hyper Registry. Bei einer Änderung schickt der Dienstanbieter den aktuellen Inhalt direkt an die Registry.

Hybrid Hyper Registry. Eine hybride Hyper Registry implementiert beide Strategien.

Eine Registry, die Caching nicht unterstützt, ignoriert den vom Client übertragenen Inhalt.

Um veraltete und nicht aktualisierte Daten zu vermeiden, werden Tupel in einer Hyper Registry außerdem als *soft state* gehalten, d.h. sie sind nur über einen bestimmten Zeitpunkt gültig und müssen vom Anbieter in regelmäßigen Abständen aufgefrischt werden. Zu diesem Zweck wird jedes Tupel mit vier Zeitstempeln TS1, TS2, TS3 und TC versehen, deren Semantik in Tabelle 1.2 erläutert ist.

Es gilt außerdem $TS1 \leq TS2 \leq TS3$. TC gibt an, wann der Inhalt durch die letzte Zwischenstation, z.B. die Hyper Registry, geändert wurde. Eine Registry kann z.B. durch TC=0 angeben, dass sie keine Caching des Inhalts vornimmt, d.h. ihre Tupel enthalten keinen Inhalt.

Einfügen, Ändern und Löschen von Tupeln sind zeitstempel-getriebene Zustandsübergänge, wie in Abbildung 1.2 dargestellt. Ein Tupel wird dabei anhand seines Schlüssels erkannt, der aus dem Kontext und dem content link besteht.

Neben der Registry kann indirekt auch der Client entscheiden, wann der zwischengespeicherte Inhalt eines Tupels aktualisiert werden soll. Bei der als *refresh-on-client-demand* bekannten Strategie schickt der Client eine Anfrage ab, die auch die Zeitstempel der Tupel inspiziert. Soll der Inhaltsanteil der Tupel aktualisiert werden, kann dies der Registry mit dem XQuery-Befehl `document(URL contentLink)` mitgeteilt werden, wodurch die Registry den aktuellen Inhalt vom Dienstanbieter bezieht. Weil die Funktion `document` auf der Registry ausgeführt wird, kann sie so implementiert werden, dass automatisch der Inhalt im Cache aktualisiert wird.

Eine letzte wichtige Funktion ist das *Throttling*. Ein hohes Maß an Aktualität der Inhalte führt offensichtlich auch zu hohem Resourcenverbrauch, bspw.

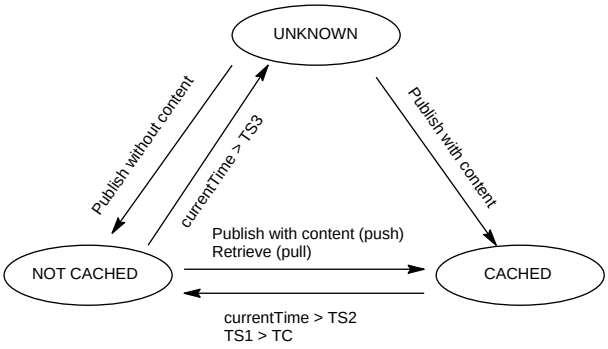

Abbildung 1.2: Zustandsübergänge bei Soft State Publication(Quelle: [1])

durch die Verbindungen zwischen Registry und Dienstanbieter. Hier muss ein Kompromiss geschlossen werden. Teil der hier beschriebenen Architektur ist ein Mechanismus, der Überlast der Registry durch zu hohe Aktualisierungsfrequenz und zu hohen Resourcenverbrauch verhindern soll. Das sogenannte *Throttling* wird durch zwei zusätzliche Zeitstempel TS4 und TS5 erreicht, die von der Registry als Ergebnis der Operation `publish` an den Dienstanbieter zurück gegeben werden. TS4 ist die *minimum idle time*. Der Anbieter darf erst nach Ablauf dieser Zeitspanne den Tupel aktualisieren. TS5 ist die *maximum idle time*. Der Dienstanbieter muss vor Ablauf dieser Zeitspanne den Tupel aktualisieren. Dienstanbieter, die diese Anweisungen ignorieren, können durch Ausschluss aus der Registry bestraft werden.

1.3.3 Anfragen an die Hyper Registry

Um im Discovery-Schritt geeignete Dienste zu finden, muss ein Client Anfragen an eine Registry schicken können. Es werden drei Arten von Anfragen unterschieden. Die einfachste Form sind *simple queries*. Diese beziehen sich immer auf ein einzelnes Tupel, z.B.

> Finde alle Musikstücke von Sergey Rachmaninoff.
> Finde alle Dienste und liefere ihre service links.

Etwas komplexer sind *medium queries*, welche die Lösung über die gesamte Ergebnismenge berechnen, z.B.

> Liefere die Anzahl der Kopien von Rachmaninoffs 'Piano Sonata #1'.

Complex queries erlauben eine Kombination von Ergebnismengen ähnlich dem *join* in Datenbanken. Bsp.:

> Finde alle Paare von Diensten, wo beide Dienste in der gleichen Domäne existieren.

Existierende Verzeichnisdienste bieten nur begrenzte Anfragemöglichkeiten. Aus diesem Grund wurde XQuery als Anfragesprache gewählt, da sie auch komplexe Anfragen ermöglicht. Da die Umsetzung einer XQuery-Schnittstelle der viel Aufwand erfordert, wurden zwei verschiedene Schnittstellen für die Registry

definiert. `MinQuery` enthält die beiden Methoden `getTuples()` und `getLinks()`, die sämtliche gespeicherten Tupel mit bzw. ohne Inhaltsanteil liefert. Auf dieser einfach Anfragemöglichkeit aufbauend kann eine `XQuery`-Schnittstelle implementiert werden, welche die vollständige Mächtigkeit von XQuery besitzt.

1.4 Zusammenfassung

In der hier beschriebenen Dissertation wurden acht Schritte zum Finden und Ausführen von Diensten beschrieben. Die Web Service Discovery Architecture beschäftigt sich im wesentlichen mit der Präsentation und Publikation von Dienstinformationen und dem Auffinden von passenden Diensten. Es wurden vier für die Architektur relevante Schnittstellen definiert.

Presenter. Diese Schnittstelle muss von jedem Dienst angeboten werden. Sie dient dem Abfragen der Dienstbeschreibung, und enhält als einzige Methode `getServiceDescription()`.

Consumer. Eine Hyper Registry erhält über diese Schnittstelle Dienstbeschreibungen von den Anbietern. Ihre einzige Methode ist `(T4, T5) publish (XML tupleSet)`.

MinQuery. Über die Schnittstelle MinQuery stellt eine Hyper Registry sämtliche Tupel bereit. Die Methode `getTuples()` liefert alle Tupel inklusive content. Das Ergebnis eines Aufrufs von `getLinks()` sind ebenfalls alle Tupel, aber mit leerem content.

XQuery. Die XQuery-Schnittstelle bietet eine Methode `query` an, mit deren Hilfe komplexe XQuery-Anfragen an die Hyper Registry gestellt werden können.

Jeder Dienst muss die Schnittstelle `Presenter` implementieren. Eine *hypermin registry* bietet zusätzlich die Schnittstellen Consumer und MinQuery an, d.h. sie ist eine Hyper Registry ohne die komplexen XQuery-Anfragemöglichkeiten.

Kapitel 2

A Peer-to-Peer Database Framework

2.1 Motivation

In einem typischen Peer-to-Peer-Netzwerk sind die einzelnen Datentupel über mehrere Knoten verteilt. Ziel eines Peer-to-Peer Database Framework ist es, diese Verteilung transparent zu machen und Anfragen auf eine globale Sicht zu ermöglichen. Dabei ist besonders die Dynamik der Daten zu beachten, d.h. deren Aktualität ist sicherzustellen.

2.2 Topologie und Routing

In dem Peer-To-Peer Database Framework werden als Knoten nur solche Dienste verstanden, die neben der Presenter-Schnittstelle mindestens auch Publication- und Anfrageschnittstellen anbieten. Registries sind ein Beispiel für Knoten. In einem Peer-to-Peer-Netzwerk können neben solchen Knoten auch andere Dienste auftreten. Möchte ein solcher Dienst eine Suchanfrage starten, muss er sich als *Originator* an einen Knoten wenden, der dann für ihn als *Agent* auftritt.

Wie die Anfrage vom Agenten weitergeleitet wird, hängt von der Art des Routings ab. Beim *Routed Response (RR)* wird sowohl die Anfrage als auch das Ergebnis von Nachbarknoten zu Nachbarknoten weitergeleitet. Bei *Direct Reponse* gibt es zwei unterschiedliche Formen: mit und ohne Einladung. Bei der Version ohne Einladung wird das Ergebnis der Suchanfrage direkt an den Agenten geschickt. Im Gegensatz dazu wird in der zweiten Version zuerst eine Einladung an den Agenten geschickt. Dieser kann auf Grundlage dieser Einladung entscheiden, von welchen Knoten er das Ergebnis abruft. Da die Form mit Einladung die bessere ist, wird im folgenden unter Direct Response nur noch die Version mit Einladung verstanden und entsprechend DR abgekürzt. Der Austausch von Nachrichten ist schematisch in Abbildung 2.1 dargestellt. Zu den drei Routingarten gibt es je eine Version *mit Metadaten*. Das Ergebnis der Suchanfrage ist dabei nicht das gesuchte Dokument, sondern Informationen darüber. Diese Metadaten können vom Originator benutzt werden, um das Dokument direkt vom Anbieter zu erhalten.

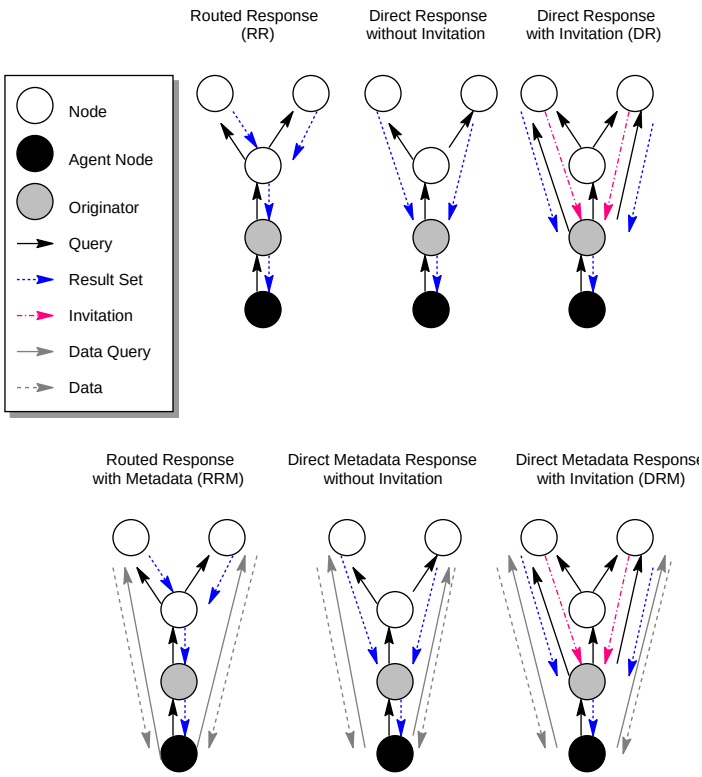

Abbildung 2.1: Routingarten(Quelle: [1])

Sowohl RR als auch DR haben ihre Vorteile. Bei DR wird das Ergebnis direkt an den Agenten geschickt, wodurch der Gesamtverkehr im P2P-Netz geringer wird. Auf der anderen Seite wird durch das Routen des Ergebnis von Knoten zu Knoten ein Caching von Ergebnissen möglich, und Duplikate können früh gelöscht werden. Da keine Übertragungsform unter allen Ümständen optimal ist, wurde die Möglichkeit der Änderung der Übertragungsform während der Anfragebearbeitung in Betracht gezogen. Bei einem *switch* wird an einem Knoten von RR zu DR oder von DR zu RR gewechselt. Wird von RR zu DR gewechselt, so ist das Ziel des Ergebnis nicht der ursprüngliche Agent, sondern der Knoten, an dem der switch durchgeführt wurde. Bei einem *shift* ändert sich das Ziel der Ergebnismenge. Ein shift macht nur bei DR Sinn, da bei RR die Ergebnisse immer von Nachbar zu Nachbar wandern. Welche Art des Routing benutzt wird sollte vom Originator als Teil der Anfrage spezifiziert werden.

2.3 Anfragebearbeitung

In Bezug auf Anfragebearbeitung ist ein P2P-Datenbanksystem wie ein verteiltes Datenbanksystem mit rekursiver Struktur zu betrachten. Dementsprechend wurden Konzepte zur Anfragebearbeitung aus dem Gebiet verteilter Datenbanken übernommen. Eine Anfrage beinhaltet eine Menge von Operatoren. Die Semantik der Anfrage kann durch verschiedene Pläne erfüllt werden. Der Anfrageoptimierer versucht, den effizientesten Plan zu finden. Ein solcher Plan ist dabei ein Baum aus Operatoren, wobei ein Elternknoten immer seine Nachfolger aufruft und auf deren Ergebnissen arbeitet.

Im P2P Database Framework folgt die Anfragebearbeitung dem *Template Query Execution Plan*, wie in Abbildung 2.2 dargestellt. Neben einer Anfrage an die lokale Datenbank (local query) werden auch Anfragen an die Nachbarknoten (neighbor query) weitergeleitet. Die Ergebnisse dieser Anfragen werden durch den Unionizer-Operator vereinigt. Der Merge-Operator nimmt als Eingabe die vereinigte Ergebnismenge und liefert das Ergebnis der Benutzeranfrage.

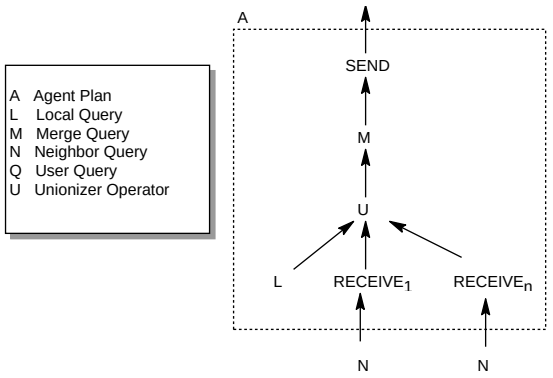

Abbildung 2.2: Template Query Execution Plan(Quelle: [1])

Eine besondere Form von Anfragen sind *rekursiv partitionierbare Anfragen.* Eine Anfrage ist rekursiv partitionierbar, wenn für einen Template Plan A eine Merge-Query M und ein Unionizer U existieren, welche die Semantik der

Anfrage unter der Voraussetzung erfüllen, dass die lokale Anfrage gleich der Benutzeranfrage und der Anfrageplan des Nachbarknoten gleich dem Anfrageplan des Agenten ist. Eine Anfrage ist demzufolge rekursiv partitionierbar, wenn derselbe Plan auf allen Knoten angewendet werden kann. Die Anfragebearbeitung kann in einem solchen Fall effizienter gestaltet werden. Einfache und mittlere Anfrage (siehe Abschnitt 1.3.3) sind per Definition rekursiv partitionierbar.

Das Ziel von *Pipelining* ist das Übertragen von Teilergebnissen noch bevor das Gesamtergebnis bekannt ist. Oft ist der Benutzer bereits mit wenigen, frühen Ergebnissen zufrieden, um mit diesen weiterarbeiten zu können. Eine Anfrage eignet sich zum Pipelining, wenn sie mindestens ein Ergebnistupel liefern kann, bevor sie selbst alle Eingabetupel erhalten hat. Operatoren mit monotoner Semantik, z.B. SELECT, UNION und SEND, ermöglichen Pipelining. Operatoren mit nicht-monotoner Semantik, z.B. SORT, MAX oder JOIN, lassen dies nicht zu.

Ein weiterer wichtiger Punkt bei der Bearbeitung von Anfragen in einem P2P-Netzwerk ist die Definition von Timeouts. Dabei wird zwischen einem *Abort Timeout*, der zum Abbrechen einer Anfrage führt, und dem *Loop Timeout*, der Aufrufschleifen verhindern soll, unterschieden. Beim Abbrechen von Anfragen ohne Pipelining kann es zum *simultaneous abort timeout* kommen. Dies bedeutet, dass wenn auch nur ein Nachbar keine Ergebnisse liefert, alle vorgeschalteten Knoten gleichzeitig die Anfrage abbrechen. Aus diesem Grund wird der Abort Timeout dynamisch gestaltet, d.h. beim Aufruf des Nachbarknotens wird der Timeout gesenkt. Dabei kommt ein *exponential decay with halving* zum Einsatz. Das Zeitfenster zur Übergabe der Ergebnisse wird dabei bei jedem Aufruf der Nachbarknoten halbiert.

Der Loop Timeout kann nicht dynamisch gestaltet werden, da es sonst zu einem *non-simultaneous loop timeout* kommen kann. In einigen Situationen kann es vorkommen, dass eine Anfrage über eine eigentlich kürzere Strecke (weniger Hops) ein zweites Mal bei einem Knoten auftaucht. Da beim ersten Erreichen die Anfrage über mehr Zwischenpunkte gelaufen ist, könnte ihr Timeout beim zweiten Eintreffen bereits erreicht sein, d.h. die Aufrufschleife wurde nicht erkannt. Aus diesem Grund sind Loop Timeouts statisch.

Die Eingabetupel für die Bearbeitung der Anfrage werden durch den *query scope* bestimmt. In bestimmten Situationen ist es sinnvoll, den Scope einzuschränken, indem man nur bestimmte Knoten aufruft. Dies kann auf sehr vielfältige Weise entweder direkt oder indirekt geschehen. Eine Möglichkeit wäre *neighbor selection*. Durch eine *neighbor selection query* kann vom Benutzer in Form einer XQuery angegeben werden, welche Nachbarknoten aufgerufen werden sollen. Die Menge der zu bearbeitenden Knoten wird aber auch durch oben beschriebene *Timeouts* eingegrenzt. Eine weitere Möglichkeit wäre die Definition eines *Radius*, also einer maximalen Anzahl von Hops ausgehend vom Agenten.

Um die Anfragebearbeitung zu beschleunigen, kann die Kolokation mehrerer Knoten innerhalb eines *node container* ausgenutzt werden. Eine solcher Container ist eine transparente Lebensumgebung für mehrere Knoten, die nach aussen hin aber immer noch als Netz von Knoten auftreten. Bei der Bearbeitung von Anfragen wird zwischen internen und externen Knoten, also Knoten, die innerhalb bzw. ausserhalb eines Containers liegen, unterschieden. Es wurden drei Möglichkeiten der Optimierung aufgezeigt. Bei der *normal query execution* findet die Ausführung der Anfragen wie gewohnt statt. Es werden lediglich bei Aufrufen zwischen Knoten innerhalb eines Containers performantere Verbindun-

gen, z.B. IPC, benutzt. Das *collecting traversal* findet in zwei Schritten statt. Im ersten Schritt werden die internen und die direkt an den Container grenzenden externen Knoten berechnet, welche von der Query berührt werden, d.h. die zum query scope gehören. Im zweiten Schritt wird die Anfrage ausgeführt, allerdings mit einigen Besonderheiten. Die lokale Anfrage des Knotens innerhalb des Containers, an den die Anfrage gestellt wurde, arbeitet nicht nur auf seinen lokalen Daten, sondern auf den Daten aller interner Knoten, die von der Anfrage berührt sind. Auch die Weiterleitung der Anfrage funktioniert anders als üblich. Die Anfrage geht nicht nur an die direkt mit dem internen Knoten verbundenen externen Knoten, sondern an alle vorher berechneten externen Knoten. Diese Methode reduziert also die Anzahl der Weiterleitungsschritte. Während die zwei bisher beschriebenen Methoden den Scope der Anfrage erhalten, wird der Scope von der *quick scope violating query* ignoriert. Dies kann geschehen, wenn beispielsweise kein Scope angegeben wurde. Diese Methode arbeitet ähnlich wie das collecting traversal, nur dass das Traversieren des Graphen entfällt. Stattdessen wird die lokale Anfrage auf der Gesamtheit der internen Tupel ausgeführt, und die Nachbaranfrage an sämtliche externen Knoten geschickt.

2.4 Nachrichtenmodell, Kommunikationsmodell und Netzwerkprotokoll

Das Nachrichtenmodell beinhaltet vier Anfragenachrichten: QUERY zum Absenden einer Anfrage, RECEIVE zum Abrufen der Ergebnisse, INVITE als Einladung beim direct response routing, und CLOSE zum Schliessen der Verbindung. Zum Antworten wird die Nachricht SEND verwendet. Eine *Transaktion* im Kontext des P2P Database Framework ist eine Sequenz von einer oder mehrerer Nachrichten in Bezug auf *eine* Query. Eine Transaktion kann durch ihre Transaktions-ID identifiziert werden, welche beim Absenden von QUERY festgelegt wurde. Neben dieser ID enthält die QUERY-Nachricht auch den gewünschten Routing-Modus.

Eine RECEIVE-Nachricht enthält als Parameter auch eine Anweisung, ob die Ergebisse synchron oder asynchron geliefert werden sollten. Im synchronen Modus folgt einer RECEIVE-Nachricht nur eine SEND-Nachricht. Im asynchronen Modus können einem RECEIVE mehrere SEND-Nachrichten folgen. Eine RECEIVE-Nachricht enthält außerdem die minimale und die maximale Anzahl von Ergebnistupeln, die geliefert werden dürfen. Eine CLOSE-Nachricht beendet eine Transaktion und zeigt dem Empfänger an, dass eventuell noch vorhandene Ergebnistupel gelöscht werden können.

Jede Transaktion ist in einem der drei Zustände Open, Closed und Unknown. Wenn eine Anfrage mit einer QUERY-Nachricht ankommt, gelangt sie in den Open-Zustand. Der Übergang zum Closed-Zustand kann verschiedene Ursachen haben. Durch eine explizite CLOSE-Nachricht kann eine Transaktion beendet werden. Wenn keine weiteren Ergebnistupel vorhanden sind, wird ebenfalls abgebrochen. Gleiches gilt, wenn die INVITE-Nachricht abgelehnt wird oder der Abort Timeout zuschlägt. Durch Erreichen den Loop Timeouts gelangt eine Transaktion vom Close-Zustand in den Unknown-Zustand und wird aus der Zustandstabelle des Knotens gelöscht.

Ein Kommunikationsmodell operiert auf abstrakten Entitäten wie Sessions

oder Kanälen. Diese abstrakten Entitäten werden durch ein Netzwerkprotokoll auf physikalische Entitäten abgebildet. Es wurde das Kommunikationsmodell von BEEP[2][3], einem IETF-Standard, übernommen. In diesem Modell können zwei Peers eine Session erzeugen, innerhalb derer ein oder mehrere Kanäle geöffnet werden können. Über diese Kanäle können Nachrichten versendet werden, die wiederum aus Frames unterschiedlicher Länge aufgebaut sind. Kanäle sind voneinander isoliert. Innerhalb eines Kanals werden Nachrichten fortlaufend abgearbeitet. Im P2P Database Framework werden unterschiedliche Transaktionen auf unterschiedlichen Kanälen übertragen.

Das Netzwerkprotokoll sagt aus, wie die abstrakten Entitäten des Kommunikationsmodells auf physikalische Entitäten abgebildet wird. Bei der Abbildung der Sessions gibt es zwei Möglichkeiten. Zum einen kann für jede Query eine neue Session erstellt werden, zum anderen könnte eine Session zwischen zwei Peers von mehreren Anfragen benutzt werden. Es wurde die zweite Alternative gewählt, wobei im konkreten Fall immer zwei Sessions zwischen zwei Peers existieren: eine für ankommende und eine für abgehende Anfragen. Wie bereits gesagt wurde, wird ein Kanal je Transaktion benutzt. Nun stellt sich die Frage, wie TCP-Verbindungen abgebildet werden: entweder eine TCP-Verbindung je Kanal (TCP multiplexing, TM) oder eine TCP-Verbindung je Session (application mutiplexing, AM). TM wäre einfacher zu implementieren, da es bereits Teil des Protokollstacks ist. Es wurde jedoch AM gewählt, da der Aufbau einer Verbindung immer mit einem Overhead verbunden ist.

2.5 Zusammenfassung

In diesem Kapitel wurde das Peer-To-Peer Database Framework vorgestellt. Es basiert auf einer Topologie, in der zwischen einfachen Diensten und Knoten, welche mindestens Publication- und Anfrageschnittstellen anbieten, unterschieden. Wenn ein Dienst eine Anfrage stellen möchte, muss er sich als Originator an einen Knoten wenden, der dann als Agent auftritt. Es wurden verschiedene Arten des Routing von Anfragen beschrieben und festgestellt, dass keine Form unter allen Umständen die beste ist. Deshalb kann während des Routing der Modus gewechselt werden.

Die Bearbeitung der Anfrage basiert auf dem Template Query Execution Plan, welcher in Abbildung 2.2 dargestellt ist. Wie ein konkreter Plan zu generieren ist, wurde in der vorliegenden Dissertation nicht beschrieben. Eine besondere Form von Anfragen sind rekursiv partitionierbare Anfragen, die dadurch gekennzeichnet sind, dass der Agentenplan gleich dem Nachbarplan ist. Dadurch kann die Anfragebearbeitung parallelisiert werden, was leider nur bei einfachen und mittleren Anfragen möglich ist. Es wurden außerdem zwei Arten von Timeouts vorgestellt: ein dynamisches Timeout zum Abbrechen von Transaktionen und ein statisches Timeout zum Verhindern von Aufrufschleifen. Desweiteren wurden drei Möglichkeiten der Begrenzung des query scope beschrieben, und Möglichkeiten der Optimierung durch Ausnutzung eines Node Container vorgestellt.

Das Peer-To-Peer Database Framework basiert auf einem Nachrichtenmodell, das fünf Nachrichtentypen QUERY, RECEIVE, INVITE, CLOSE und SEND beinhaltet und den Transaktionsbegriff als Sequenz von Nachrichten definiert. Eine Zustandsmodell für Transaktionen und Regeln für die Zustands-

übergänge wurden ebenfalls angegeben. Desweiteren wurde ein Kommunikationsmodell beschrieben, dessen abstrakte Entitäten Session und Kanal durch ein Netzwerkprotokoll auf physikalische Entitäten abgebildet wurde.

Literaturverzeichnis

[1] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework For XQueries over dynamic distributed content and its application for scalable service discovery.* PhD thesis, Technische Universität Wien, 2002.

[2] Marshall T. Rose. The blocks extensible exchange protocol core. IETF RFC 3080, March 2001.

[3] Marshall T. Rose. Mapping the beep core onto tcp. IETF RFC 3081, March 2001.