Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Bachelorarbeit

Optimierung der verteilten und parallelen Datenverarbeitung in der GINSENG-Middleware

Verfasser:

Andreas Meister

17. April 2012

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Andreas Lübcke

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Dr.-Ing. Anja Jugel,
Dipl. Inf. Thomas Heinze

SAP Research Dresden
Chemnitzer Strasse 48, D-01187 Dresden
Germany

Meister, Andreas:
Optimierung der verteilten und parallelen Datenverarbeitung in der GINSENG-Middleware
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2012.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Abkürzungen

GPB	Google Protocol Buffers
DP	Distributed Placement
$\boldsymbol{DT(S)}$	Delaunay-Triangulation einer Punktmenge $\boldsymbol{S}$
DT	Delaunay-Triangulation
DDT	verteilte Delaunay-Triangulation
CEP	Complex Event Processing

Kapitel 1

Einführung

In den letzten Jahren gewann die zeitnahe Verarbeitung von Ereignisse (z.B. Messwerte, Werte von Aktienkurse usw.) z.B. bei der Verarbeitung von Sensordaten, Verkehrsanalyse und Aktienhandel immer mehr an Bedeutung [CJ09]. Diese Anwendungsbeispiele müssen eine potentiell unendliche Datenmenge verarbeiten, so erzeugen Sensoren periodisch Ereignisse, z.B. durch die periodische Messung eines Drucks in einer Rohrleitung. Diese periodische Messung führt zu einem potentiell unendlichen Strom aus Ereignissen. Um logische Zusammenhänge zwischen den einzelnen Ereignissen verschiedener Ereignisströme erkennen zu können, müssen die einzelnen Ereignisse weiter verarbeitet werden und falls nötig aggregiert werden. Ereignisse werden hierbei kontinuierlich über einen längeren Zeitraum verarbeitet. Die Bearbeitung der Daten muss effizient durchgeführt werden, um die gewünschten logischen Zusammenhänge zeitnah erkennen zu können. Da die Geräte (z.B. Sensoren), die Ereignisse erzeugen, sich meisten an unterschiedlichen Orten befinden (z.B. verteilt auf ein Fabrikgelände) müssen diese Geräte (Knoten) zu einem Netzwerk zusammengeschlossen werden. Die Erkennung von logischen Zusammenhängen erfolgt in einem Netzwerk auf einem oder mehreren Knoten des Netzwerks. Mit der Entwicklung eines drahtlosen Sensornetzwerks, das eine effiziente und zeitnahe Verarbeitung von Sensordaten gewährleistet, beschäftigt sich das Ginseng-Projekt.
Ziel des Ginseng-Projekts ist ein drahtloses Sensornetzwerk zu entwickeln, das anwendungsspezifische Leistungsanforderungen erfüllt, und bestehende Ressourcenmanagementsysteme (z.B. Produktlebenszyklusmanagement-Systeme zum Planen und Überwachen von Wartungsarbeiten [SRAZ08]) integriert [KJ10b]. Neben der Entwicklung eines drahtlosen Netzwerks wird eine ereignisbasierte Middleware entwickelt.
Zielanwendung des Ginseng-Projektes ist eine Öl-Raffinerie. Hier werden von Druck-, Durchfluss-, Temperatur-, Luftfeuchtigkeit-, Gas- und Füllstand-Sensoren periodisch Ereignisse (Events) erzeugt. Die Sensoren erzeugen jeweils einen unendlichen Ereignis-/ Event-Strom, der z.B. zur Gefahrenerkennung (Öl- oder Gasleck) zeitnah verarbeitet werden muss. Um zu vermeiden, dass Gefahren nicht rechtzeitig erkannt werden, wird eine kontinuierliche Performanz-Kontrolle durchgeführt.
Die Middleware selbst besteht aus vier Komponenten [KJ10a], ein Adapter-Framework, ein Data-Processing Framework, eine Query-Processing- und Distribution-Komponente und eine Monitoring- und Management-Komponente. Das Adapter-Framework dient zur Integration von bestehenden Ressourcenmanagementsystemen bzw. heterogenen Datenquellen wie z.B. Sensornetzwerke, Maschinendaten, aber auch Datenbanken oder Webdatenquellen. Dabei werden die ggf. unterschiedlichen Datenformate der Datenquel-

len in ein einheitliches Datenformat überführt. Nach der Transformation der Events im Adapter-Framework erfolgt die Verarbeitung im Data-Processing-Framework. Das Data-Processing-Framework ist ein Complex Event Processing (CEP)-System, das aus verschiedenen Komponenten zur Datenverarbeitung besteht, unter anderem Pipes [KJ10b], eine Java-Bibliothek, die Operatoren zur Verarbeitung von Event-Strömen bereitstellt. Hierbei werden Daten aus verschiedenen Datenquellen gesammelt und verarbeitet um komplexe Zusammenhänge (z.B. Gaslecks) zu erkennen. Um die Performanz des Systems zu garantieren wird der Datenfluss und die Anwendungslogik durch die Monitoring- und Management-Komponente überwacht. Sensornetzwerke und andere Datenquellen können über das gesamte Betriebsgelände verteilt sein. Um die zeitnahe Datenverarbeitung zu garantieren und z.B. parallele Datenauswertung zu ermöglichen, sollte auch die Middleware auf mehrere Rechnerknoten verteilt werden. Zur automatischen Verteilung und Verwaltung der Operationen der Datenverarbeitung dient die Query-Processing- und Distribution-Komponente. Um eine möglichst effiziente Anfragebearbeitung zu ermöglichen, müssen Anfragen, die das System bearbeiten soll, optimiert werden.
Ziel der vorliegenden Arbeit ist ein System zur Optimierung einer dezentralen, verteilten Verarbeitung von Event-Strömen zu entwickeln und prototypisch zu implementieren. Ziel der Optimierung ist die Bearbeitungszeit (Latenz) einer Anfrage und die Netzwerklast zu minimieren. Dabei wird die Platzierung von Operatoren im System optimiert. Andere Optimierungsmöglichkeiten wie z.B. Multi-Query-Optimierung [XLTZ07] oder Query-Rewriting [SMMP09] werden nicht betrachtet, da diese den Umfang einer Bachelorarbeit sprengen würden. Bei der Multi-Query-Optimierung wird versucht Ergebnisse von Operatoren bestehender Anfragen bei neuen Anfragen wiederzuverwenden. Aufgabe des Query-Rewriting ist Anfragen, die das System bearbeiten sollen, zu einer effizienteren Struktur umzuformen. Um das Ziel der Arbeit zu erreichen müssen folgende Aufgaben durchgeführt werden.
Erste Aufgabe ist bestehende Ansätze zur Platzierung von Operatoren zu recherchieren. Die bestehenden Verfahren müssen miteinander verglichen werden, und ein geeignetes Verfahren ausgewählt werden.
Die zweite Aufgabe ist das ausgewählte Verfahren zu implementieren. Sowohl bei der Auswahl, als auch bei der Implementierung des Verfahrens müssen die Anforderungen des Systems beachtet werden. Abschließend muss das implementierte System bzgl. Skalierbarkeit evaluiert werden.
Im Folgenden werden zuerst in Kapitel 2 die Grundlagen der restlichen Arbeit beschrieben, so wird CEP und Pipes erläutert. Des Weiteren wird die Aufgabenstellung der Platzierung von Operatoren dargestellt, und einige vorhandene Verfahren zur Platzierung von Operatoren präsentiert. Danach werden die Anforderungen, die an das Verfahren gestellt werden, in Kapitel 3 analysiert. Es werden Anwendungsfälle, Problemstellung sowie Anforderungen des Systems dargestellt. Außerdem werden die in Kapitel 2 beschriebenen Verfahren zur Platzierung der Operatoren anhand der Anforderungen des Systems miteinander verglichen, und ein Verfahren für die Implementierung ausgewählt. Nach der Anforderungsanalyse wird das Design des Systems in Kapitel 4 beschrieben. Es wird die Systemarchitektur, externe Schnittstellen, sowie Aufgaben und Funktionsweise der einzelnen Komponenten des Systems erläutert. In Kapitel 5 wird die Implementierung des in Kapitel 3 ausgewählten Verfahrens zur Platzierung der Operatoren dargestellt. Hierzu wird sowohl das Verfahren, als auch die dazu benötigten Komponenten erläutert. In Kapitel 6 wird das System bzgl. der Skalierbarkeit evaluiert. In Kapitel 7 erfolgt eine

Zusammenfassung der Arbeit, sowie eine Beschreibung möglicher Erweiterungen des Systems.

Kapitel 2

Grundlagen

Im folgenden Kapitel werden die Grundlagen der Arbeit beschrieben. Es wird die Problemstellung des dezentralen Complex Event Processing (CEP) dargestellt. Es wird Pipes, eine Java-Bibliothek, die Operatoren zum CEP bereitstellt, erläutert und bestehende Algorithmen zur Platzierung von Operatoren vorgestellt, die während der Konzeptionierung der Arbeit betrachtet wurden.

2.1 Complex Event Processing

CEP [MC11] ist die Erkennung von vorgegebenen Mustern innerhalb einer Menge von unterschiedlichen Event-Strömen. Events sind Ereignisse, die für die weitere Verarbeitung von Bedeutung sind, z.B. eine Änderung des Drucks der durch ein Druck-Sensors erkannt wurde, eine Timer-Event, das signalisiert, das eine gewisse Zeit vergangen ist, bzw. allgemein jede erkennbare Zustandsänderung [MFP06]. Event-Ströme sind physisch unabhängig von einander, sie können jeweils unterschiedliche Datenformate und unterschiedliche Event-Raten besitzen. Event-Ströme können jedoch in einem logischen Zusammenhang stehen [LF98]. Jedes Event besitzt ein festes und bekanntes Datenformat. Anwendung findet CEP, falls die Verarbeitung der Events zeitnah erfolgen soll, oder die Event-Raten der Datenquellen zu groß ist, um die gesamte Datenmenge zu speichern.
Ein Beispiel für CEP-Systeme sind Branderkennung, Finanzanwendungen, z.B. Betrugserkennung, durch Analyse von Kreditkartentransaktionen und Verarbeitung von Daten aus Sensornetzwerk, z.B. in Umweltwissenschaften oder Fabriken ([SMMP09], [CcC+02], [WDR06]).
Im Anwendungsfall des Ginseng-Projekts, einer Ölraffinerie, werden 35000 Sensoren mit konstanten Datenraten verwaltet um allgemeine Aufgaben wie z.B. Bestimmung von Druck, Flüssigkeitsstand, Erkennung von Lecks und Erfassung allgemeiner Umweltdaten, wie z.B. Temperatur, zu erledigen. Zur Erkennung von Lecks, können z.B. der Flüssigkeitsstand und Drucksensor einer Leitung in einen logischen Zusammenhang gebracht werden. Sinkt der Flüssigkeitsstand und Druck einer Leitung unter eine bestimmte Grenze existiert gegebenenfalls ein Leck, und das Ventil der entsprechenden Leitung muss sofort geschlossen werden.
Zur Mustererkennung werden Operationen auf den Events zur Verarbeitung und Analyse ausgeführt. Falls notwendig werden Events zur weiteren Verarbeitung gespeichert, um zeitliche Zusammenhänge erkennen zu können. Operationen, die zur Mustererkennung

verwendet werden, können sowohl generische Operationen wie Filter, Join, Max, Min, als auch komplexere Operationen wie Top-k [CH08] sein.
Eine Musterkennung wird im Gegensatz zu Datenbankanfragen kontinuierlich durchgeführt. Bei einer Datenbankanfrage wird eine Anfrage formuliert. Die formulierte Anfrage wird auf einem bestehenden Datenbestand ausgeführt und liefert ein Ergebnis. Die Operatoren einer Anfrage werden bei der Ausführung einer Anfrage nicht gespeichert. Bei der Mustererkennung in CEP werden Operatoren einer Anfrage im System erstellt. Quellen einer Anfrage erzeugen kontinuierlich Events, diese werden von Operatoren der Anfrage verarbeitet. Operatoren einer Anfrage erzeugen bei der Bearbeitung von Events abhängig von den durchgeführten Operationen neue Events, die an die nachfolgenden Operatoren weitergeleitet werden, dadurch wird ein Netzwerk von Operatoren gebildet. Events werden von den einzelnen Operatoren falls nötig zwischengespeichert, jedoch nicht persistent. Werden Events zwischengespeichert geschieht dies nur mit Hilfe eines verschiebbaren Fensters [BBD+02], entweder werden Events zeitlich begrenzt, oder es wird eine gewisse Anzahl von Events gespeichert. Ein einzelnes Event hat hierbei eine geringere Bedeutung. Ist ein Knoten überlastet, d.h. müsste er mehr Events verarbeiten, als seine Kapazität ermöglicht, kann ein Event verworfen werden (load shedding) [BBD+02].

2.2 Pipes

Pipes [CHK+03] [CHKS04] (Public Infrastructure for Processing and Exploring Streams) ist eine Java-Bibliothek, die grundlegende Funktionalitäten zur kontinuierlichen Verarbeitung und Analyse von Datenströmen bereitstellt. Pipes ermöglicht, basierend auf dem Publish-Subscribe-Modell [EFGK03], die Erstellung komplexer Anfragegraphen über beliebige Datenquellen. Komplexe Anfragegraphen können zur Mustererkennung genutzt werden. Im Publish-Subscribe-Modell werden zwei wesentliche Rollen unterschieden, Publisher und Subscriber. Publisher erzeugen Events. Um Events von Publishern zu erhalten registriert sich ein Subscriber bei beliebig vielen Publishern. Erzeugt ein Publisher ein neues Event wird dieses an alle registrierten Subscriber weitergegeben.
Ein Anfragegraph besteht aus drei unterschiedlichen Arten von Operatoren: Sources (Datenquellen/ Publisher), Sinks (Datensenken/ Subscriber) und Pipes (Operationen zur Bearbeitung und Analyse des Datenstroms/ Subscriber für Dateneingang und Publisher der Ergebnisse). Eine Datenquelle (Source) kann beliebig viele Ausgangsverbindungen besitzen, besitzt jedoch keine Eingangsverbindung. Sie kann sowohl als Schnittstelle zu einer externen Datenquelle sowie als Datengenerator benutzt werden. Zum Testen und Evaluieren bietet Pipes bereits implementierte Datengeneratoren, die Events zur Bearbeitung im System erzeugen.
Eine Datensenke (Sink) hat mindestens eine Eingangsverbindung, aber besitzt keine Ausgangsverbindungen zu anderen Operatoren. Zur Bearbeitung der erhaltenen Daten wird jeder Datenquelle der Sink eine eindeutige ID zugeteilt. Die Sink dient zur Weiterleitung von Anfrageergebnissen an Anwendungen, die die Ergebnisse weiterverarbeiten. Es werden ebenfalls implementierte Senken, z.B. zur Ausgabe der Ergebnisse in einem Textfeld, bereitgestellt.
Der Pipes-Operator kombiniert die Eigenschaften einer Source und einer Sink. Er besitzt mindestens eine Eingangsverbindung und besitzt mindestens eine Ausgangsverbindung. In den Pipes-Operatoren erfolgt die eigentliche Verarbeitung und Analyse der Datenströme.

Pipes-Operatoren erhalten von ihren Datenquellen Daten-Events, und verarbeiten diese anhand der internen Logik, und leiten die Ergebnisse an ihre Datensenken weiter. Es werden eine Vielzahl von Pipes-Operatoren bereitgestellt, z.B Filter, Join, Difference, Group, und Sort. Diese sind zur besseren Wiederverwendung generisch implementiert, und benötigen zur Ausführung Funktionen bzw. Prädikate, die die interne Logik steuern. Ein Anfragegraph startet mit einer Menge von unterschiedlichen Datenquellen und endet mit einer Menge von unterschiedlichen Datensenken. Zwischen Datenquellen und Datensenken wird eine beliebige Anzahl von Pipes-Operatoren ausgeführt. Bei der kontinuierlichen Bearbeitung der Anfragen werden Daten-Events von den Datenquellen erzeugt und an die zugehörigen Datensenken (Sink/ Pipes-Operatoren) weitergeleitet.
Bei der Verarbeitung werden Daten nicht zwischen den einzelnen Operatoren gepuffert, sondern direkt verarbeitet und weitergeleitet. Die Operatoren erhalten von ihren Datenquellen Daten, abhängig vom Typ der Pipes-Operatoren werden diese Daten verändert und an die Senken der Operatoren weitergeleitet. Zur Effizienzsteigerung können zum einen Events eines Source/ Pipes-Operators an beliebig viele Pipes/ Sink-Operatoren weitergeleitet werden. Zum anderen können Verbindungen dynamisch zur Laufzeit eines Operators verändert werden, dies ermöglicht ein inkrementelles Hinzufügen neuer Anfragen, da Ergebnisse bestehender Operatoren genutzt werden können. Beide Eigenschaften können für eine Multi-Query-Optimierung [XLTZ07] genutzt werden. Bei einer Multi-Query-Optimierung wird versucht, die Ergebnisse eines Operators in mehreren Anfragen zu verwenden. Zur einfachen Erweiterung der bestehenden Funktionalitäten werden neben den bereits implementierten Operatoren abstrakte Klassen, der drei beschriebenen Operatoren (Source, Sink und Pipes) bereitgestellt, die die Grundfunktionalitäten implementieren und beliebig erweitert werden können. Diese abstrakten Klassen wurden in früheren Arbeiten genutzt, um spezifische Operatoren für das Ginseng-Projekt zu implementieren, siehe Tabelle 2.1. In früheren Arbeiten wurde ebenfalls eine graphische Oberfläche, der *QueryAgent*, zur benutzerfreundlichen Anfragedefinition implementiert, siehe Abbildung 2.1.

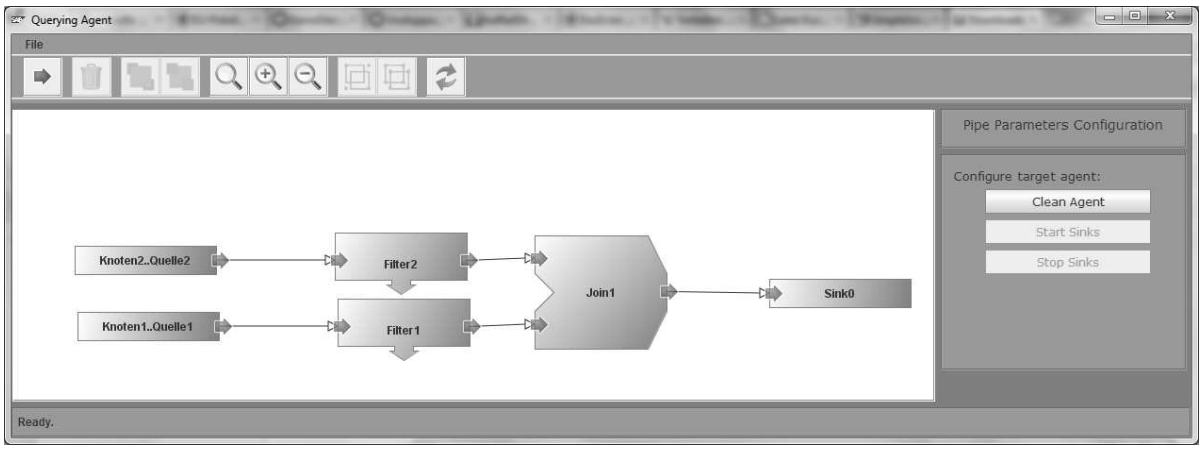


Abbildung 2.1: QueryAgent

Operator-Typ	**Parameter**	**Beschreibung**
DropperPipe	DropRatio (Integer)	Verwirft periodisch Events
BatteryLevelCommand-Trigger	Frequency (Long)	Erzeugt Warnung bei einen niedrigen Batteriestand eines Sensors
ValveCommandTrigger	Frequency (Long)	Erzeugt Befehle zur Steuerung von Ventilen
RegressionPipe	StartLevel (Integer) AlarmLevel (Integer)	Erhaltene Werte werden zu einem Wert aggregiert, z.B. Max oder Min.
AggregatorPipe	StepSize (Integer) NumberOfTuples (Integer) TimeBase (Boolean) AggregationFunction (String)	Aggregiert festgelegte Menge von Events, z.B. Average, Count usw.
ConstantMultiplication	Factor (Integer)	Multipliziert den Event-Wert mit Factor
ConstantDivisor	Factor (Integer)	Dividiert den Event-Wert mit Factor
ConstantAddition	Number (Integer)	Addiert den Event-Wert mit Number
ConstantSubstraction	Number (Integer)	Subtrahiert vom Event-Wert den Number-Wert
TemporalJoin	Operation (String) TupleAgeForRemoval (Integer) AllowedTimestampDeviation (Integer)	Verbindet Events zweier Event-Ströme abhängig von den Zeitstempeln der Events miteinander und führt die angegebene Operation (z.B. Addition) aus
SelectionPipe	ComparisonValue (Double) Operation (String)	Verwirft Events, die die definierte Bedingung nicht erfüllen.

Tabelle 2.1: Operatorentypen

2.3 Platzierung von Operatoren

Übersteigen die benötigten Ressourcen für das CEP die Kapazitäten eines Knotens, oder werden Datenquellen an unterschiedlichen Orten ausgeführt, können mehrere Knoten für das CEP verwendet werden. Besteht das CEP-System aus mehreren Knoten, muss anhand

Algorithmus	**Pietzuch** [PLS+06]	**Balazinksa** [BBS04]	**Bonfils** [BB03]	**Widom** [SMW05]	**Ahmad** [AceJ+05] [Ac04]
Architekturbeschreibung	unabhängiges Modul: Berechnung der Platzierung in einem übergeordneten Netzwerk	verteilte Platzierungslogik: Entscheidung der Platzierung treffen die einzelnen Knoten selbst	verteilte Platzierungslogik	verteilte Platzierungslogik	verteilte Platzierungslogik: Platzierung werden von zonenbasierenden Koordinatorknoten getroffen
Metrik	Produkt aus Latenz und Bandbreite	Auslastung und verfügbare Ressourcen	Bandbreite	Bandbreite und Berechnungskosten	Bandbreite oder Produkt aus Latenz und Bandbreite
Operationen auf Operator-Ebene	Wiederverwendung	Replikation	Wiederverwendung	Wiederverwendung	Replikation
Auslöser zur Änderung	Periodische Neubewertung	Periodische Neubewertung	Periodische Neubewertung	Periodische Neubewertung	Grenzwert
Komplexität	$O\left(i \times n^2\right)$	$O\left(N + C + N \times C\right)$	$O\left((m \times n)^3\right)$	$O\left((m+n)\right.$ $\left.\times\, log\,(m+n)\right)$	$O\left(d^h + d^{\frac{h(h-1)}{t}}\right)$
Probleme	keine optimale Platzierung garantiert	keine Berücksichtigung von Abhängigkeiten zwischen Operatoren	Kommunikations-Overhead, initiale Platzierung notwendig, keine optimale Platzierung garantiert	Beschränkung auf hierarchische Netzwerkstruktur	Anzahl und Auswahl der Zonen, die von Koordinatoren verwaltet werden, ist nicht geklärt

Tabelle 2.2: Erweiterte Klassifizierung nach [LLS08]

eines Verfahrens bestimmt werden, auf welchem Knoten des Systems ein Operator ausgeführt werden soll. Abhängig davon, wie ein System diese Zuordnung der Operatoren auf die Knoten des Systems bzw. allgemein die Ausführung der Mustererkennung organisiert, kann man CEP-Systeme unterschiedlich klassifizieren [MC11]. Grundsätzlich gibt es zwei unterschiedliche Ansätze für CEP-Systeme. Die Organisation der Musterkennung kann zentral oder dezentral erfolgen. Bei einem zentralen CEP-System gibt es einen Knoten, der die gesamte Mustererkennung organisiert. Er besitzt alle benötigten Kenntnisse, die zur Durchführung einer Mustererkennung notwendig sind, z.B. Netzwerktopologie, Event-Raten der Event-Ströme, ggf. auch die Latenzen zwischen den einzelnen Systemknoten. Da alle Informationen für eine korrekte Ausführung der Mustererkennung ständig auf einem Knoten aktualisiert werden müssen, kann deshalb am ausführenden Knoten ein Flaschenhals entstehen. Durch diesen Flaschenhals ist die Skalierbarkeit zentraler Systeme eingeschränkt. Dezentrale CEP-Systeme können Probleme bzgl. der Skalierbarkeit umgehen [LLS08]. Bei dezentralen CEP-Systemen verfügt kein Knoten über alle Informationen, die zur Organisation der Mustererkennung notwendig sind. Eine Mustererkennung wird in einem dezentralen System abhängig vom System von einem festen, oder beliebigen Knoten gestartet. Die Operatoren für eine neue Musterkennung werden abhängig vom benutzten Verfahren im Netzwerk verbreitet.
Im Folgenden werden unterschiedliche Ansätze zur Platzierung von Operatoren in einem System verteilter Knoten erläutert. Hierbei werden Teile einer bestehenden Klassifizierung [LLS08] verwendet und dargestellt, siehe Tabelle 2.2. Es gibt eine Vielzahl verschiedener Ansätze zur Platzierung von Operatoren, die für unterschiedliche Anforderungen entworfen wurden. Da die Arbeit ein dezentrales System implementieren soll, siehe Abschnitt 3.3, werden einige dezentrale Ansätze, die für das implementierte System näher betrachtet wurden, kurz beschrieben.
Das von Ahmad et. al. ([AceJ$^+$05], [Ac04]) vorgestellte Verfahren baut auf einer verteilten Hashtabelle auf. Das zugrunde liegende Netzwerk wird in Zonen eingeteilt. Für jede Zone wird ein entsprechender Koordinator zugeordnet, der zentral die Optimierung der Zone durchführt. Ein Koordinator wird hierbei genutzt, um Ergebnisse der Anfragen an weiterverarbeitende Anwendungen weiterzuleiten. Um eine globale Optimierung zu ermöglichen, übermitteln die einzelnen Koordinatoren untereinander Informationen, z.B. Kosteninformationen für neue Operatoren. Zur eigentlichen Optimierung können je nach Bedarf unterschiedliche Heuristiken verwendet werden, die die Bandbreite und/ oder die Latenz der Datenübertragungen zwischen den Knoten berücksichtigen. Es wird jedoch kein Verfahren erläutert um die Zonen, die von den Koordinatoren verwaltet werden, zu bilden. Hierbei müssten vermutlich Kenntnisse über das Netzwerk bestehen, um eine gute Zuordnung treffen zu können.
Der Algorithmus von Widom et. al. [SMW05] beschränkt sich bei seinen Überlegungen auf hierarchische Netzwerkstrukturen. Die Optimierung erfolgt anhand der Selektivität der Operatoren. Die Operatoren werden nach ihren Abhängigkeiten, Kosten und Selektivität geordnet, und entsprechend der Ordnung im Netzwerk platziert. Auf einer Ebene der Hierarchie werden hierbei die Kosten für die Ausführung mit den Kosten der Datenübertragung verglichen. Sind die Kosten für das Übermitteln der Daten geringer, als für die Ausführung wird der Operator auf eine höhere Ebene verschoben. Um eine optimale Ausführung zu gewährleisten, können zusätzlich weitere Kosten im Netzwerk berücksichtigt werden. Um die Kosten für die Übertragung zu berechnen, werden die Verbindungen, über die Daten übertragen werden müssen, ebenfalls als Operator modelliert.

Bei der Platzierung können ebenfalls begrenzte Ressourcen der Knoten beachtet werden. Der Algorithmus von Bonfils et. al. [BB03] orientiert sich bei der Optimierung an physikalischen Kräften. Abhängig von der Position eines Operators entstehen Kosten durch die verursachte Netzwerklast der Vorgänger und Nachfolger und die Latenz zwischen den Operatoren. Diese Kosten können als Kräfte in einem Raum modelliert werden. Ziel ist es, die Operatoren so im System zu verschieben, dass die Kräfte (Netzwerklast und Latenz) minimiert werden. Die Kosten einer Anfrage werden anhand der Kosten seiner Operatoren bestimmt, dabei werden die Kosten eines Operators rekursiv durch die Kosten der Vorgängeroperatoren und den Kosten zur Übertragen der Daten an den Operator berechnet. Da alle Kosten in einer Senke zusammengefasst werden, können die Kosten der Anfrage durch die Kosten der Senke ermittelt werden. Zur Ermittlung der optimalen Position, erstellt jeder Operator vorläufige Kopien auf den Nachbarknoten seines aktuellen Knotens. Sind die Kosten auf einem anderen Knoten niedriger als auf dem aktuellen Knoten wird der Operator verschoben.
Das von Balazinksa et. al. [BBS04] beschriebene Verfahren orientiert sich an der Zusammenarbeit von Internetdienstanbietern. Anhand von Kostenplänen der Operatoren wird die Platzierung der Operatoren optimiert. Für jeden Operator, der auf einen Knoten läuft, werden sowohl Kosten des Operators als auch Bezahlung für den Operator bestimmt. Die Kosten können je nach Bedarf durch unterschiedliche Metriken abgeschätzt werden z.B. Netzwerklast, CPU-Kosten usw. oder auch eine Kombination mehrerer Metriken. Unterschiedliche Knoten können verschiedene Metriken verwenden. Erzeugt ein Knoten einen Operator erhält er für diesen Operator keine Bezahlung. Jeder Knoten versucht seinen eigenen Gewinn (Auslastung) zu maximieren. Hierfür werden zwischen Knoten des Systems Kostenpläne von Operatoren ausgetauscht. Die Bezahlung des möglichen neuen Knoten, sind die aktuellen Kosten des Operators auf dem aktuellen Knoten. Sind die Kosten für einen Operatoren auf einem anderen Knoten geringer, als die Kosten auf dem aktuellen Knoten inklusive der Kosten einer Verschiebung, wird der Operator auf den anderen Knoten verschoben. Jeder Knoten hat eine bestimmte Kapazität von Ressourcen zur Ausführung von Operatoren zur Verfügung. Je höher die Auslastung eines Knotens ist, desto größer werden die Kosten zur Ausführung eines Operators. Es wird garantiert, dass kein Knoten seine Kapazität überschreitet, solange die Kapazität des Gesamtsystems nicht überschritten wird.
Der Algorithmus von Pietzuch et. al. [PLS+06] fügt zwischen dem System, das den Datenstrom analysiert und bearbeitet, und dem Netzwerk eine weitere Komponente ein, die zur dezentralen Platzierung der Operatoren verwendet wird. Zur Bestimmung der Platzierung der Operatoren wird ein dezentraler Kostenraum konstruiert, indem die Positionen der Knoten verwaltet werden. Durch die Latenzen zwischen den Knoten werden die Positionen der Knoten abgeschätzt, die zur Platzierung der Operatoren verwendet werden. Die Metrik des Verfahrens berücksichtigt sowohl die benötigte Latenz, als auch die Datenraten, die zur Bearbeitung einer Anfrage notwendig sind. Zur Platzierung der Operatoren werden für die Operatoren ebenfalls Positionen im virtuellen Kostenraum ermittelt. Hierbei wird eine Technik zur Entspannung von Federn verwendet. Operatoren werden als Objekte modelliert, die durch Federn miteinander verbunden sind. Es wird zwischen Operatoren, die nicht verschoben werden können, wie Senken und Datenquellen, und Operatoren, die verschoben werden können, unterschieden. Abhängig von der Ausdehnung der Feder (Latenz) und der Federkonstante (Datenrate) wirkt eine Kraft auf einen Operator. Ziel ist es durch Verschiebung von Operatoren die im System enthaltene Kraft zu minimieren,

und dadurch die Latenz und die Datenmenge, die über das Netzwerk gesendet werden muss, zu minimieren. Die ermittelten Positionen werden anschließend zur Erstellung der Operatoren im System verwendet. Zur Bestimmung der Platzierung der Operatoren einer Anfrage werden die Positionen der Senken und Datenquellen benötigt.

Kapitel 3

Anforderungsanaylse

Im folgende Abschnitt werden die Anwendungsfälle des Systems erläutert. Es wird die Problemstellung der dezentralen Platzierung von Operatoren dargestellt. Des Weiteren werden die Anforderungen des Systems dargestellt, und eine Analyse der bereits erläuterten Verfahren bzgl. der Anforderungen durchgeführt, siehe Abschnitt 2.3.

3.1 Anwendungsfälle

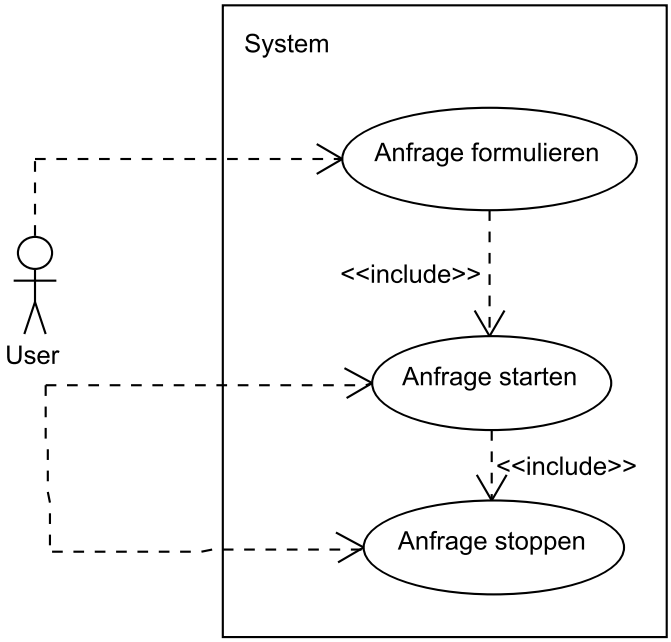


Abbildung 3.1: Anwendungsfälle

Da die gesamte Funktionalität der Anfrageoptimierung vor dem Nutzer verborgen wird, hat der Benutzer drei Möglichkeiten mit dem System zu interagieren, siehe Abbildung 3.1. Die erste Möglichkeit ist, dass der Benutzer Anfragen formulieren kann. Die zweite Möglichkeit ist, dass der Benutzer bereits formulierte Anfragen starten kann. Bereits gestartete Anfragen können als dritte Interaktionsmöglichkeit beendet werden. Um die

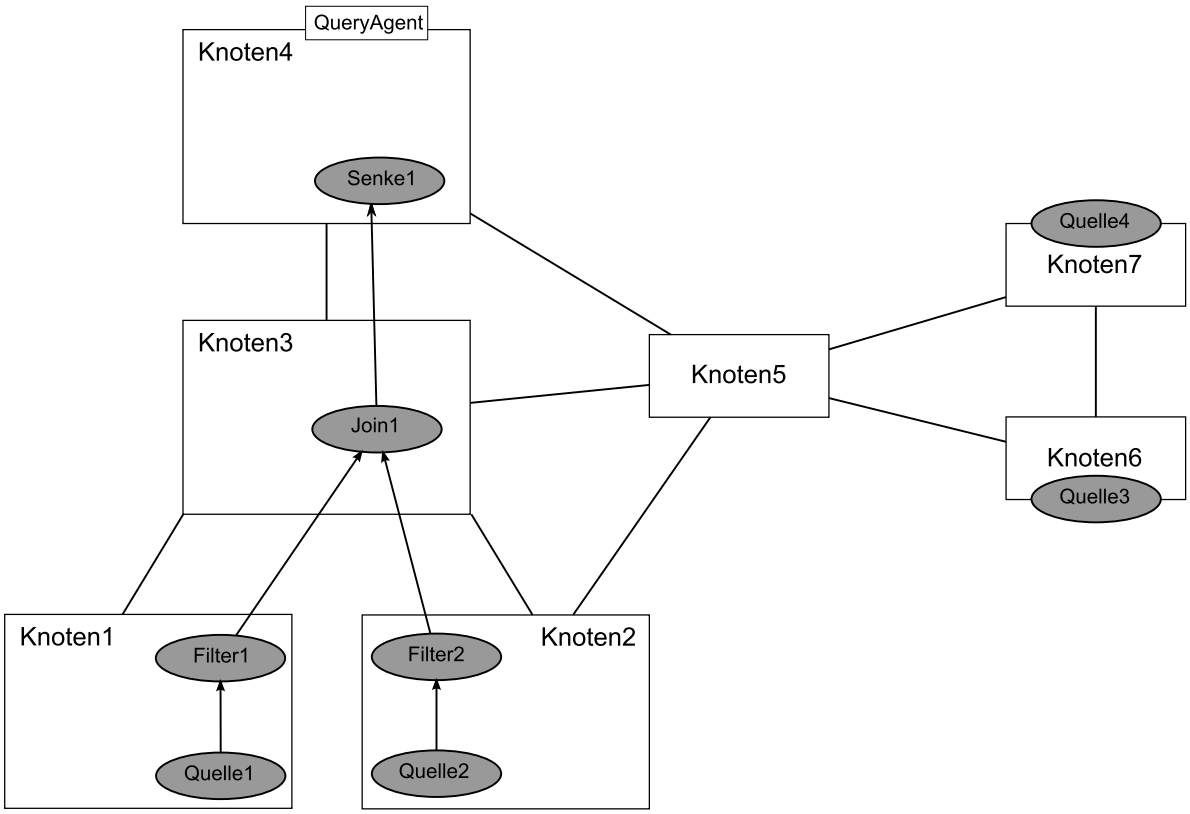
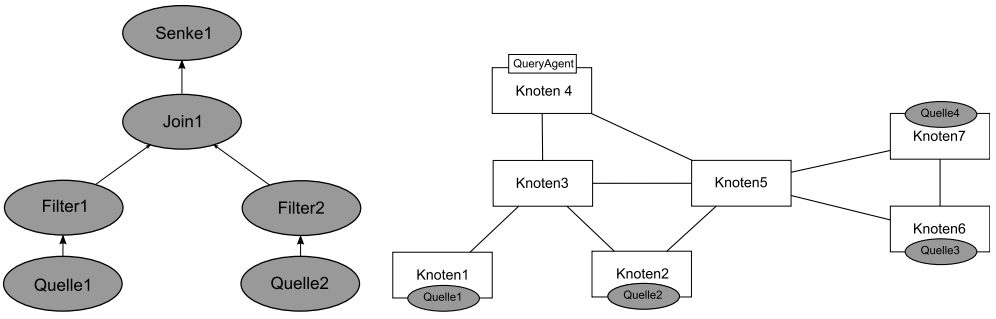


Abbildung 3.3: Beispielanfrage im Knotensystem

Anfragen im System verarbeiten zu können wird die graphische Repräsentation der Anfrage beim Start in ein generisches Anfrageformat (siehe Abschnitt 4.3) überführt und anschließend weiterverarbeitet.

3.2 Problemstellung

Im nachfolgenden Abschnitt wird die Problemstellung der Platzierung der Operatoren genauer beschrieben.



(a) Beispielanfrage (b) Knotensystem

Abbildung 3.2: Beispielkonfiguration

Ein Nutzer kann eine Anfrage, die vom System verarbeitet werden soll, mit Hilfe einer graphischen Oberfläche, dem *QueryAgent*, definieren, starten und beenden. Eine Anfragen hat eine Baumstruktur, siehe Abbildung 3.2(a). Wurzel einer Anfrage ist immer eine Senke, Blätter der Anfragen sind immer Datenquellen. Zwischen Senke und Datenquellen können beliebig viele Operatoren ausgeführt werden, z.B. Joins, Filter und Aggregationen.
Das System zur Bearbeitung der Anfrage ist ein verteiltes dezentrales System. Knoten des Systems besitzen kein Wissen über die globale Struktur des Netzwerkes, besitzen jedoch Kommunkationsverbindungen zu physischen Nachbarknoten. Ein Beispiel System ist in Abbildung 3.2(b) dargestellt, hierbei repräsentieren die Verbindungen zwischen den Knoten die entsprechende Kommunkationsverbindungen. Jeder Knoten kann Anfragen starten und Operatoren ausführen. Da die Senken einer Anfrage die Ergebnisse der Anfrage liefern, bleiben Senken auf dem Knoten fixiert, der die Anfrage gestartet hat. Mindestens ein Knoten muss eine Datenquelle besitzen. Datenquellen repräsentieren Sensoren des Sensornetzwerks. Da Sensoren physisch an Knoten gebunden sind, können die Datenquellen ebenfalls nicht verschoben werden. Beim Starten einer neuen Anfrage wird der Anfrageplan, siehe Abbildung 3.2(a), in ein generisches Anfrageformat überführt, siehe Abschnitt 4.3. Im generischen Anfrageformat sind alle benötigten Informationen zur Ausführung einer Anfrage vorhanden, z.B. eine Beschreibung der Operatoren einer Anfrage, welche Datenquellen verwendet wurden und wie die Operatoren miteinander verbunden sind. Anhand des generischen Anfrageformats und den Informationen, die in der Initialisierungsphase bestimmt wurden, werden die Knoten bestimmt, auf dem die Operatoren einer Anfrage ausgeführt werden. Bei der Erstellung der Operatoren registriert sich ein Operator bei seinen Datenquellen. Die Datenquellen leiten entsprechend die erzeugten Nachrichten an alle registrierten Operatoren weiter. Wurden alle Operatoren einer Anfrage im System erzeugt, werden die Ergebnisse der Anfragen an den Senken bereitgestellt. Wird eine Anfrage durch den Nutzer beendet, werden entsprechend alle Operatoren einer Anfrage aus dem Netzwerk gelöscht. Wird die Anfrage 3.2(a) von Knoten 4 des in Abbildung 3.2(b) dargestellten Systems gestartet, ist die in Abbildung 3.3 dargestellte Platzierung der Operatoren sinnvoll. Die zwei Filter-Operatoren sind auf den zwei Knoten der Quellen platziert, die Senke auf dem Knoten des *QueryAgents*, und der Join-Operator auf einem Knoten zwischen den zwei Filter-Operator und der Senke der Anfrage.
Da das System zur Bearbeitung der Anfragen ein verteiltes dezentrales System ist, verfügt kein Knoten über das gesamte Wissen des Systems, dementsprechend muss sowohl die Platzierung, als auch die Ausführung der Anfrage bzw. die Weiterleitung der Ergebnisse der Operatoren im System dezentral erfolgen.
Vor der Bearbeitung von Anfragen des Nutzers, wird zum Systemstart eine Initialisierungsphase durchgeführt, in der die notwendige Informationen, z.B. Routinginformationen, Latenzen zwischen Knoten und Informationen über Datenquellen bestimmt und zwischen den Knoten des Systems ausgetauscht werden.

3.3 Anforderungen

Im nachfolgenden Abschnitt werden die Anforderungen des Systems beschrieben.

Dezentrales Verfahren zur Platzierung der Operatoren im System Aufgabe ist ein dezentrales verteiltes CEP-System zu implementieren. Die Operatoren selbst werden nicht implementiert, sondern werden aus einem bestehenden *Pipes-System* übernommen. Da die initiale Platzierung der Operatoren bereits eine komplexe Aufgabe ist, wird auf verschiedene andere Aspekte eines CEP-Systems verzichtet, da diese den Umfang dieser Arbeit sprengen würden. Die Einschränkungen, die bei der Platzierung der Operatoren getroffen wurden, werden im Folgenden beschrieben.
Ziel ist es eine initiale Platzierung der Operatoren festzulegen, es soll jedoch keine dynamische Optimierung zur Laufzeit durchgeführt werden. Bei einer dynamischen Optimierung würden bestehende Operatoren eines Knotens auf andere Knoten verschoben werden. Eine dynamische Optimierung kann durchgeführt werden, falls neue Operatoren auf dem selben Knoten ausgeführt werden müssen, oder, falls sich Informationen ändern, die zur Platzierung der Operatoren verwendet wurden, z.B. Änderung der Selektivität eines Operators, Änderung der Netzwerkstruktur, o.Ä. Werden bestehende Operatoren verschoben, müsste das Routing der Nachrichten entsprechend angepasst werden. Des Weiteren müsste für die Verschiebung von zustandsorientierten Operatoren (z.B. Joins) ein Algorithmus implementiert werden, z.B. [WB10], der den Zustand des zu verschiebenden Operator ebenfalls überträgt.
Das System soll keine dynamische Netzwerkstruktur unterstützen. Bei einer Änderung der Netzwerkstruktur (Entfernen/ Hinzufügen/ Ausfall von Knoten), muss entsprechend ein weiterer Mechanismus implementiert werden, um zum einen das Routing-Verfahren zu aktualisieren. Zum anderen muss zum korrekten Ablauf des Systems sichergestellt werden, dass Informationen über Operatoren, die auf den Knoten ausgeführt werden, rekonstruiert werden können. Ändert sich die Netzwerkstruktur müsste ggf. eine dynamische Optimierung der bestehenden Operatoren erfolgen, um eine optimale Anfragebearbeitung zu ermöglichen. Sollten Knoten entfernt werden, die Datenquellen besitzen, muss diese Information ebenfalls im Netzwerk verbreitet werden.
Die Verbindungsdaten zwischen den Knoten sollen bei der Initialisierung des Systems bekannt sein. Sind keine Verbindungsinformationen zwischen den Knoten vorhanden, muss ein Verfahren implementiert werden, bei dem die Verbindungsdaten ausgetauscht, und die Verbindungen zwischen den Knoten bei Bedarf automatisch initialisiert werden.
Zwischen den Knoten des Systems gehen Nachrichten nicht verloren. Insbesondere für die Nachrichten des Kontrollflusses ist es wichtig, dass keine Nachrichten verloren gehen, da sonst das System inkonsistent wird. Ist ein Nachrichtenverlust möglich muss ein Mechanismus implementiert werden, um diesen zu erkennen. Nachrichten müssten ggf. gepuffert werden, um bei Eintreten eines Nachrichtenverlustes die verlorengegangene Nachricht erneut verschicken zu können.

Aufbau einer verteilten Ausführung von Anfragen mittels Pipes Die Operatoren der Anfrage werden hierbei von einem bestehenden *Pipes-System* bereitgestellt, das in früheren Arbeiten zur Verwendung im Ginseng-Projekt erweitert wurde. Die Operatoren, die für Anfragen verwendet werden können sind in Tabelle 2.1 dargestellt. Zur Benutzerfreundlichkeit, sollen Anfragen inkrementell hinzugefügt bzw. entfernt werden können. Eine Multi-Query-Optimierung erfolgt hierbei nicht, d.h. Ergebnisse bereits vorhandener Operatoren werden nicht zur Effizienzsteigerung genutzt.

Eingeschränkte Kapazitäten eines Knotens bei der Platzierung von Operatoren Ein Knoten des Systems verfügt über begrenzte Ressourcen (z.B. CPU, Arbeitsspeicher), diese sollen bei der Platzierung der Operatoren einer Anfrage berücksichtigt werden. Soll ein Operator auf einem Knoten ausgeführt werden, dem keine Ressourcen mehr zur Verfügung stehen, soll nach Möglichkeit ein neuer Knoten des Systems zur Ausführung bestimmt werden, wird kein Knoten mit genügen freien Ressourcen gefunden, wird der Operator trotzdem auf dem ursprünglichen Knoten ausgeführt.

Wiederverwendung des existierenden Anfrageformats und Knoteninformationen Anfragen, die das System verarbeiten soll, werden in einem bestehenden generischen Anfrageformat, siehe Abschnitt 4.3, an das System gestellt. In dem generischen Anfrageformat wird für eine Anfrage eine eindeutige ID vergeben. Neben der ID der Anfrage werden ebenfalls eine Beschreibung der Operatoren mit den jeweils notwendigen Parametern, und Verbindungsinformationen zwischen den Operatoren der Anfrage gespeichert.

Geringer Kommunikations-Overhead zwischen den Knoten Das Ziel des Systems ist zum einen die Netzwerklast zu minimieren. Deshalb soll das Verfahren sowohl bei der Initialisierung der Anfrage, als auch bei der Bearbeitung der Anfrage einen möglichst geringen Kommunikations-Overhead zwischen den Knoten besitzen.

Skalierbarkeit bzgl. Anzahl der Knoten und Anfragen Die Evaluation soll die Skalierbarkeit des Systems überprüfen. Zwei Kriterien der Skalierbarkeit müssen untersucht werden. Zum einen muss bei der Initialisierung des Systems überprüft werden, ob die Anzahl der Nachrichten, die zur Initialisierung gesendet werden müssen, als auch die benötigte Zeit zur Initialisierung skaliert. Zum anderen muss ermittelt werden, ob die Anzahl der Anfragen, die das System bearbeiten kann skaliert.

Netzwerktopologie unabhängig Das System soll auf unterschiedliche Topologien übertragbar sein, und dementsprechend sich nicht auf eine bestimmte Netzwerktopologie beschränken.

3.4 Vergleich von Verfahren zur Platzierung von Operatoren

Im Abschnitt 2.3 wurden bereits verschiedene Verfahren zur Platzierung von Operatoren erläutert, deren Eignung im Folgenden untersucht wird. Die Anforderungen, die das zu verwendende Verfahren erfüllen muss, sind in Abschnitt 3.3 dargestellt. Ein Überblick für den Vergleich zwischen den beschriebenen Anforderungen und den vorgestellten Verfahren ist in Tabelle 3.1 dargestellt.
Der Algorithmus von Ahmad et. al. [AceJ+05] [Ac04] erfüllt generell die Anforderungen. Durch die zentrale Optimierung einer Zone durch einen Koordinator, kann ein Koordinator die begrenzten Ressourcen der Knoten innerhalb einer Zone bei der Platzierung der Operatoren berücksichtigen. Da die Platzierung der Operatoren nur durch die Koordinatoren der Zonen durchgeführt wird, ist ebenfalls ein geringer Kommunikations-Overhead

Anforderung	**Pietzuch** [PLS+06]	**Balazinska** [BBS04]	**Bonfils** [BB03]	**Widom** [SMW05]	**Ahmad** [AceJ+05] [Ac04]
Berücksichtigung begrenzter Knotenkapazität	✓	✓	✓	✓	✓
geringer Kommunikations-Overhead	✓	(✓)	(×)	✓	✓
Skalierung bzgl. der Knotenanzahl	✓	✓	✓	✓	✓
Skalierbarkeit bzgl. Anfragenanzahl	✓	(✓)	(✓)	✓	✓
Topologie unabhängig	✓	✓	✓	(×)	✓
Vollständigkeit	(✓)	✓	×	✓	×

Tabelle 3.1: Anforderungen der Arbeit an das Verfahren zur Platzierung von Operatoren

des Verfahrens gewährleistet. Da die Zonen, die von den Koordinatoren verwaltet werden, frei gewählt werden können, kann garantiert werden, dass ein Koordinator nicht zum Flaschenhals einer Zone wird, somit ist die Skalierbarkeit des Systems sichergestellt. Da der Koordinator das gesamte Wissen der Zone besitzt, kann das Verfahren auf unterschiedliche Topologien übertragen werden. Ein Problem des Verfahrens ist, dass eine Kernfrage des Verfahrens, die Bildung der Zonen und Zuordnung der Zonen zu Koordinatoren, nicht abschließend geklärt wurde. Da kein Verfahren bekannt ist, dass dieses Problem dezentral löst, müsste bei der Einrichtung des Systems manuell Zonen gebildet werden und eine sinnvolle Zuordnung der Zonen zu Koordinatoren getroffen werden. Eine manuelle Zuordnung setzt jedoch Kenntnisse über die globale Struktur des Netzwerkes voraus, und stellt einen großen Aufwand bei der Initialisierung des Systems dar.
Das Verfahren von Widom et. al. [SMW05] kann ebenfalls die begrenzten Ressourcen eines Knotens bei der Platzierung der Operatoren berücksichtigen. Da jeder Knoten für sich entscheidet, ob ein Operator auf ihm oder auf einem anderen Knoten ausgeführt werden soll, ist ebenfalls ein geringer Kommunikations-Overhead garantiert. Durch den dezentralen Ansatz, die optimale Ausführung der Anfragen, und der geringe Kommunikations-Overhead, skaliert das Verfahren ebenfalls. Durch die Einschränkung auf eine hierarchische Netzwerktopologie, wird eine Anforderung (Unabhängigkeit der Netzwerktopologie) verletzt. Zwar könnte das Verfahren vermutlich trotzdem auch auf unterschiedliche Netzwerktopologien ausgeführt werden, jedoch müssten hierbei gesonderte Effizienzbetrachtungen durchgeführt werden, und eine zusätzliche Datenstruktur erstellt werden, die eine Abstraktion der physischen Netzwerktopologie auf ein hierarchisches Modell durchführt. Die Abstraktion der Netzwerktopologie, würde dazu führen, das eine optimale Platzierung der Operatoren nicht mehr gegeben ist.
Der Algorithmus von Bonfils et. al. [BB03] kann durch die vorläufigen Operatoren zum einen garantieren, dass die begrenzten Ressourcen bei der Platzierung der Operatoren berücksichtigt werden, zum anderen, dass das Verfahren auf unterschiedliche Topologien übertragbar ist. Da das Verfahren zur Optimierung nur Informationen der Operatoren berücksichtigt, sollte das Systems bzgl. der Knotenanzahl skalieren. In der Beschreibung des Verfahrens wird explizit daraufhin gewiesen, dass nicht überprüft wurde, inwieweit der Austausch der Kosteninformation zwischen aktiven und vorläufigen Operatoren einer Anfrage einen Kommunikations-Overhead darstellt, da der Austausch der Kosteninformationen insbesondere bei einer hohen Konnektivität des Netzwerkes einen starken Einfluss haben wird, wird vermutet, dass diese Anforderung nicht erfüllt wird. Bei einem großen Kommunikations-Overhead würde die Skalierbarkeit bzgl. der Anfragen nicht mehr gewährleistet sein, da der Kommunikations-Overhead, jedoch nicht untersucht wurde, kann nicht abgeschätzt werden, ob das Verfahren bzgl. der Anfragenanzahl skaliert. Außerdem wird eine funktionierende zufällige initiale Platzierung der Operatoren benötigt, für das ein weiteres Verfahren implementiert werden muss. Ein weiteres Problem ist, dass der Algorithmus keine optimale Platzierung der Operatoren gewährleistet.
Der Algorithmus von Balazinksa et. al. [BBS04] kann durch den Austausch der Kostenpläne ebenfalls die begrenzten Ressourcen der Knoten bei der Platzierung der Operatoren berücksichtigen. Da ebenfalls die Optimierung nur anhand der Informationen der Operatoren durchgeführt wird, sollte das Verfahren bzgl. der Knotenanzahl skalieren. Das Verfahren kann auf beliebige Netzwerktopologien übertragen werden. Da im Verfahren jedoch keine Abhängigkeiten zwischen den einzelnen Operatoren im System berücksichtigt werden, können Komplexitätsbetrachtungen und Platzierungsergebnisse nicht auf den

Anwendungsfall dieser Arbeit übertragen werden. Abhängigkeiten zwischen Operatoren können insbesondere in Netzwerken mit geringer Konnektivität, zu einer hohen Netzwerklast bzw. hohen Latenzen führen und zu einem hohen Kommunikations-Overhead, wenn abhängige Operatoren weit entfernt im Netzwerk platziert werden. Die Vernachlässigung der Abhängigkeiten der Operatoren kann ebenfalls dazu führen, dass das System nicht bzgl. der Anfragenanzahl skaliert. Außerdem müsste noch ein Verfahren implementiert werden, das die initialen Kosten eines Operators bzw. einer Anfrage abschätzt.
Der Algorithmus von Pietzuch et. al. [PLS+06] erfüllt generell alle Anforderung. Es kann die begrenzte Knotenkapazitäten berücksichtigen. Das Verfahren besitzt einen geringen Kommunikations-Overhead, es ist skalierbar, und kann auf unterschiedliche Netzwerktopologien übertragen werden. Der Algorithmus benötigt lediglich eine Abschätzung der Latenzen durch Positionen in einem metrischen Raum, dieser kann jedoch dezentral für beliebige Netzwerkstrukturen erstellt werden. Negativ ist jedoch, dass keine optimale Platzierung der Operatoren erfolgt. Problematisch ist ebenfalls, dass die Zuordnung der Positionen der Operatoren zu den Knoten des Systems im beschriebenen Verfahren zentral durchgeführt wird, zwar werden dezentrale Verfahren als Beispiel für eine Integration genannt, jedoch wurden diese Verfahren nicht mit dem Verfahren getestet.
Unter den betrachteten Verfahren erfüllen nur die Verfahren von Ahmad [AceJ+05] [Ac04], Balazinksa et. al. [BBS04] und Pietzuch et. al. [PLS+06] alle Anforderungen. Da im Verfahren Ahmad et. al. [AceJ+05] [Ac04] jedoch nicht alle Aspekte (Bildung der Zonen und Zuordnung der Zonen zu Koordinatoren) geklärt sind, wird dieses Verfahren nicht verwendet. Da beim Verfahren von Balazinksa et. al. [BBS04] die Annahme getroffen wird, das Operatoren unabhängig voneinander sind, und dies insbesondere bei Operatoren einer Anfrage nicht zutrifft, wird dieses Verfahren ebenfalls verworfen. Da das Verfahren von Pietzuch et. al. [PLS+06] alle Anforderung erfüllen und keine Annahmen trifft, die gegen die Anforderungen verstoßen, dient dieses Verfahren als Grundlage zur Platzierung der Operatoren im entwickelten System.

Kapitel 4

Design

In diesen Kapitel wird der grundlegende Aufbau der entwickelten Software dargestellt. Es wird beschrieben, welche bestehenden Komponenten integriert und welche Einschränkungen getroffen wurden. Des Weiteren werden die Schnittstellen sowie der Aufbau und Funktionsweise des Systems zur Bearbeitung der Anfrage beschrieben.

4.1 Systemarchitektur

Nach der Analyse der Systemanforderungen und -einschränkungen, siehe Abschnitt 3.3, wurde die Systemarchitektur zur dezentralen Optimierung der verteilten Anfrageverarbeitung entworfen.
Ein Knoten des Systems besteht aus den folgenden Komponenten (siehe Abbildung 4.1):

- PipesConnector
- ConnectionManager
- QueryManager
- Forwarding
- Distributed Placement
- QueryAgent
- Kommunikation (Netty)
- erweitertes Pipes-System

Innerhalb eines Knotens ist der *QueryAgent* optional, alle anderen Komponenten sind erforderlich um eine korrekte Ausführung des Systems zu gewährleisten.
Das *erweiterte Pipes-System*, die *Kommunikationskomponente (Netty)*, sowie der *QueryAgent*, konnten aus bestehenden Projekten übernommen und integriert werden.
Der Informationsaustausch zwischen Knoten des Systems benötigt, jeweils für den Kontrollfluss (QMEvent) und für den Datenfluss (DataEvent) unabhängige Datenformate. Der Kontrollfluss wird vom *Distributed Placement (DP)* verwaltet, der Datenfluss wird

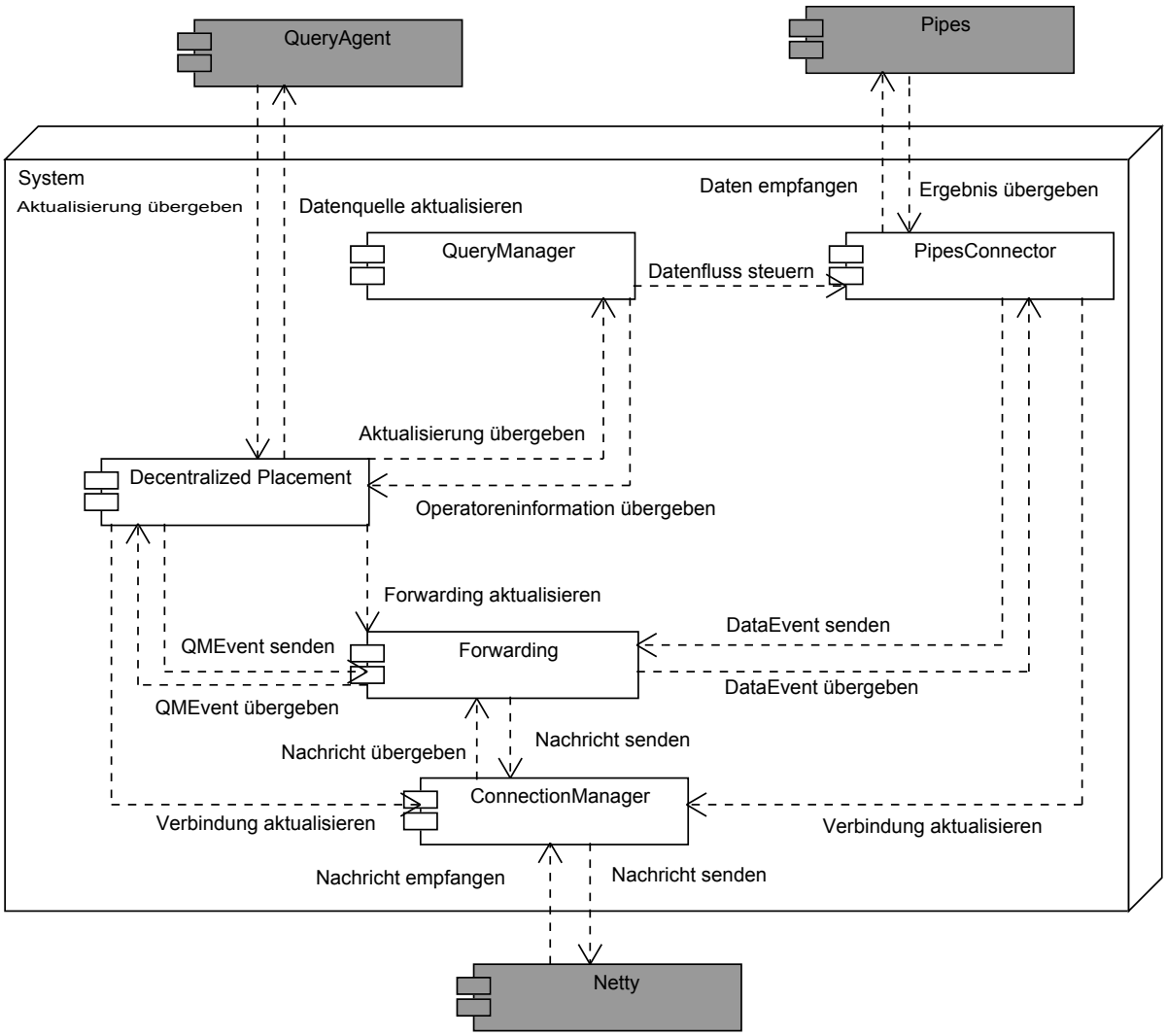


Abbildung 4.1: Design-Architektur

vom *PipesConnector* verwaltet. Beide Nachrichtenformaten werden über die *Kommunikationskomponente (Netty)* übertragen. Dabei baut jeder Knoten zu seinen physischen Nachbarknoten für jedes der zwei Datenformate eine eigene Verbindung auf, die im *ConnectionManager* verwaltet wird, siehe Abbildung 4.2. Die Verbindungsverwaltung im *ConnectionManager* erfolgt, um die konkrete Umsetzung der Kommunikation zu verbergen.

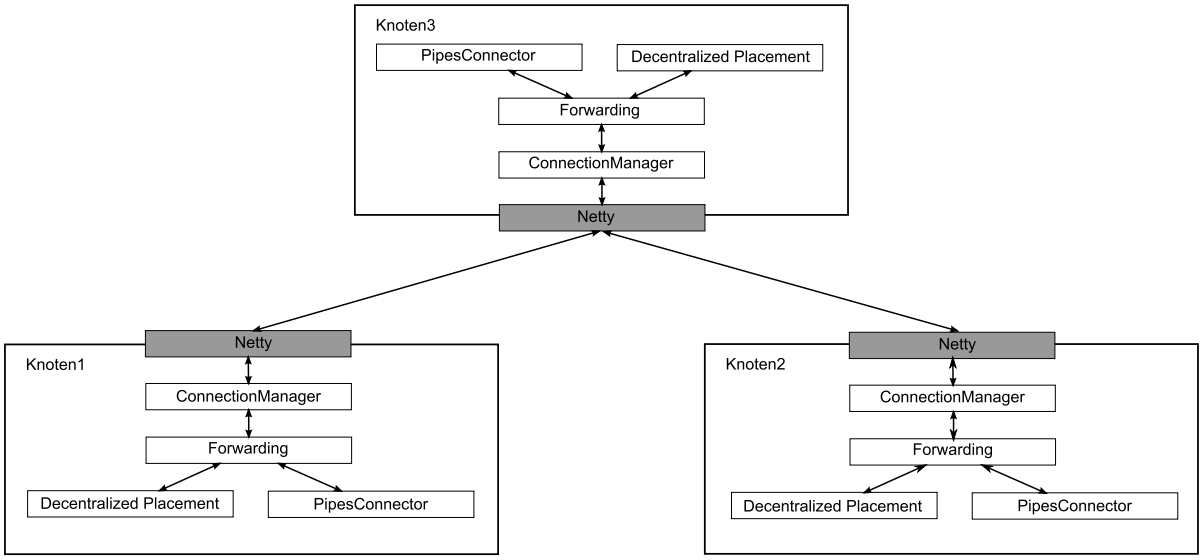


Abbildung 4.2: Kommunikation zwischen Knoten des Systems

4.1.1 Existierende Komponenten

Das *erweitertes Pipes-System*, der *QueryAgent* und die *Kommunikationskomponente (Netty)* konnten aus bestehenden Projekten übernommen werden.
Das *erweiterte Pipes-System* [CHK+03] bildet die Kernkomponente der Anfrageverarbeitung. Es stellt Operatoren zur Verarbeitung und Analyse von Datenströmen bereit. In der vorhandenen Version von Pipes sind bereits eine Vielzahl von Operatoren implementiert, z.B. Join, Filter und Aggregationen, die für die Verwendung im Ginseng Projekt angepasst bzw. durch neue Operatoren erweitert wurden.
Der *QueryAgent* bietet ein graphisches Nutzerinterface um Anfragen zu definieren, die über das System ausgeführt werden sollen. Eine Anfrage hat immer eine Baumstruktur. Blätter jeder Anfrage ist mindestens eine Quelle, und Wurzel einer Anfrage ist immer eine Senke, siehe Abbildung 3.2(a). Nach Formulierung der Anfrage wird die graphische Repräsentation in ein generisches Anfrageformat (siehe Abschnitt 4.3) überführt.
Die *Kommunikationskomponente* des Systems wird durch *Netty* [Net12], ein asynchrones, ereignisgesteuertes Netzwerk-Framework, implementiert. Über die *Kommunikationskomponente* werden Informationen zwischen den Knoten des Systems ausgetauscht, z.B. Events, die vom *erweiterte Pipes-System* erzeugt wurden. Die gesamte Kommunikation erfolgt asynchron. Da die Kommunikation asynchron erfolgt, muss das implementierte System zur Bearbeitung von Anfragen ggf. Informationen zwischenspeichern um eine korrekte Bearbeitung der Anfragen zu gewährleisten. Besteht eine Verbindung zu einem Knoten, können sowohl Daten empfangen, als auch gesendet werden dies ist notwendig, da zur

Initialisierung des Systems und zur Aktualisierung von Systeminformationen Informationen zwischen Nachbarknoten ausgetauscht werden müssen. Für jedes Nachrichtenformat muss eine getrennte Verbindung eingerichtet werden, wobei jedes Nachrichtenformat mit Google Protocol Buffers [Goo12] definiert wird. Da durch die getrennten Verbindungen für die Nachrichtenformate, die Anzahl an Verbindungen vergrößert wird, wird eine zusätzliche Komponente implementiert, die die Kommunkationsverbindungen verwaltet, der *ConnectionManager*.

4.1.2 Implementierte Komponenten

Im Folgenden werden die Komponenten beschrieben, die selbst entworfen wurden, der *ConnectionManager*, der *PipesConnector*, der *QueryManager*, das *DP* und die *Forwarding-Komponente*. Der *ConnectionManager* stellt grundlegende Eigenschaften zur Verwaltung der Verbindungen bereit, die für das *Distributed Placement (DP)* und den *PipesConnector* benötigt werden. Für jeden Knoten werden sowohl die Verbindungsdaten (eindeutige ID, Hostname, Port zum Datenempfang, Port um Daten zu Senden) gespeichert, als auch die bestehenden Verbindungen zu Nachbarknoten verwaltet. Zur Laufzeit können zu bekannten Knoten getrennte Schreib- und Leseverbindungen aufgebaut bzw. entfernt werden. Es werden hierbei jedoch unterschiedliche Verbindungen für den Kontrollfluss und den Datenfluss initialisiert.
Der *PipesConnector* ist ein Wrapper für das *erweiterte Pipes-System*. Hierbei wird ein Interface für das *erweiterte Pipes-System* bereitgestellt um alle wesentlichen Funktionalitäten von Pipes nutzen zu können. Mit Hilfe des *PipesConnectors* ist es somit möglich Operatoren auf einen Knoten zu erstellen bzw. zu entfernen und Verbindungen der Operatoren dynamisch anzupassen. Der *PipesConnector* kontrolliert die Verbindungen des Datenflusses über den *ConnectionManager*. Eine wesentliche Aufgabe ist, Datenstrom-Events von Operatoren des Knotens über die *Forwarding-Komponente* an andere Operatoren, die auf anderen Knoten des Systems ausgeführt werden, weiterzuleiten bzw. Events anderer Knoten an bestehende Operatoren des Knotens zu übergeben. Hierfür wird für jeden Operator des Knotens gespeichert, ob und ggf. an welche Operatoren die Ergebnisse eines Operators weitergeleitet werden müssen. Hierbei wird, wie in Pipes selbst auch, das Publish-Subscribe-Modell [EFGK03] verwendet, d.h. ein Operator kann sich für Events eines anderen Operators registrieren. Erzeugt ein Operator neue Events werden diese an alle registrierten Operatoren weitergeleitet. Registrierte Operatoren können dabei sowohl auf dem selben Knoten, als auch auf einem anderen Knoten des Systems, ausgeführt werden. Gesteuert wird der Datenfluss vom *QueryManager*.
Der *QueryManager* übernimmt die Verwaltung der Operatoren, die auf einem Knoten ausgeführt werden. Er steuert den Datenfluss über den *PipesConnector*. Hier werden ebenfalls die Repräsentation der Operatoren gespeichert, die im *erweiterten Pipes-System* ausgeführt werden. Im *QueryManager* können ggf. weitere Optimierungsschritte erfolgen, z.B. eine Multi-Query Optimierung. Bei einer Multi-Query-Optimierung wird versucht Ergebnisse von bestehenden Operatoren zur Effizienzsteigerung bei neuen Anfragen wiederzuverwenden. Verfahren die zur Multi-Query-Optimierung implementiert werden können sind [JC08] bzw. [Sel88].
Kern der Anfrageoptimierung ist das *DP*. Das *DP* entscheidet, wie die einzelnen Operatoren der Anfrage auf die Knoten des Systems verteilt werden. Zur Aufgabe der Komponente gehört die Bestimmung der Position des Knotens im Netzwerk. Dazu müssen Knoteninfor-

mationen bestimmt werden, und diese Informationen mit den Nachbarknoten ausgetauscht werden. Die einzigen Informationen die hierbei im gesamten Netzwerk verbreitet werden ist die Position der Datenquellen. Abhängig von der bestimmten Knotenposition und den Knoteninformationen, wird zum einen die Platzierung der Operatoren im System durchgeführt. Zum anderen aktualisiert das *DP* die *Forwarding-Komponente* durch Erstellung neuer Einträge in der Forwarding-Tabelle der *Forwarding-Komponente*, siehe Abschnitt 5.5.3, dies kann zum Beispiel erforderlich sein, falls sich Operatoren anderen Knoten für Operatoren auf dem Knoten registrieren. Das *DP* verteilt ebenfalls die Kontrollinformationen an die Komponenten innerhalb eines Knotens bzw. leitet diese ggf. an Nachbarknoten weiter. Das *DP* kontrolliert die Verbindungen des Kontrollflusses ebenfalls über den *ConnectionManager*. Hierbei werden jedoch die Verbindungen für Kontroll- und Datenfluss getrennt verwaltet.
Die *Forwarding-Komponente* übernimmt das Routing der Nachrichten des Kontroll- und Datenflusses. Aufgabe der *Forwarding-Komponente* ist zum einen für Nachrichten, die von anderen Knoten des Systems empfangen wurden, zu entscheiden ob eine Nachricht weitergeleitet werden muss, oder ob der Knoten die Nachricht verarbeiten soll. Zum anderen leitet die *Forwarding-Komponente* Nachrichten, die auf dem Knoten erzeugt wurde, an andere Knoten des Systems weiter. Falls eine Nachricht weitergeleitet werden soll, wird durch diese Komponente eine bestehende Verbindung des *ConnectionManagers* ausgewählt, die zur Weiterleitung der Nachricht verwendet wird. Empfangene Nachrichten, die der Knoten verarbeiten soll, werden falls notwendig durch das *DP* an andere Komponenten des Knotens weitergeleitet. Verarbeitet der Knoten z.B. eine Nachricht zur Erzeugung eines Operators, werden die benötigten Informationen an den *QueryManager* weitergeleitet. Nachrichten des Kontrollflusses werden an das *DP* weitergeleitet, Nachrichten des Datenflusses werden an den *PipesConnector* weitergeleitet. Die Aktualisierung der Forwarding-Information erfolgt durch das *DP*.

4.2 Funktionsweise

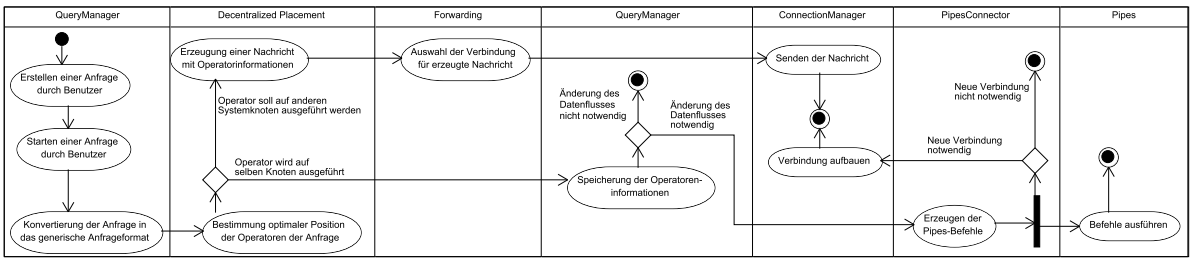


Abbildung 4.3: Ablauf beim Starten einer Anfrage

Der Ablauf beim Starten einer Anfrage ist in Abbildung 4.3 dargestellt. Der Benutzer erstellt mit Hilfe der graphischen Nutzeroberfläche, dem *QueryAgent*, eine Anfrage. Diese graphische Repräsentation wird beim Starten der Anfrage in ein generisches Anfrageformat (siehe Abschnitt 4.3) überführt, und an das *DP* übergeben. Das *DP* bestimmt nun die ideale Platzierung der Operatoren. Wird ein Operator auf dem eigenen Knoten verwaltet, werden die Informationen an den *QueryManager* des Knotens weitergeleitet. Der *QueryManager* speichert die Informationen über den Operator. Falls ein neuer Operator

erzeugt werden muss, gibt der *QueryManager* die benötigten Informationen zur Erstellung des Operators an den *PipesConnector* des Knotens weiter. Der *PipesConnector* erzeugt daraufhin Steuerbefehle für das *Pipes-System*, die an dieses übergeben und ausgeführt werden. Müssen Datenstrom-Events auf andere Knoten übertragen werden, wird überprüft ob eine neue Verbindung aufgebaut werden muss, oder ob die Daten über bestehende Verbindungen übertragen werden können. Soll ein Operator auf einem anderen Knoten des Systems ausgeführt wird eine Nachricht (QMEvent) mit den benötigten Informationen erzeugt und an die *Forwarding-Komponente* übergeben. Die *Forwarding-Komponente* wählt die Verbindung des *ConnectionManagers* aus, die zur Senden der Nachricht genutzt werden soll. Nach der Bestimmung der Verbindung wird die Nachricht übermittelt.

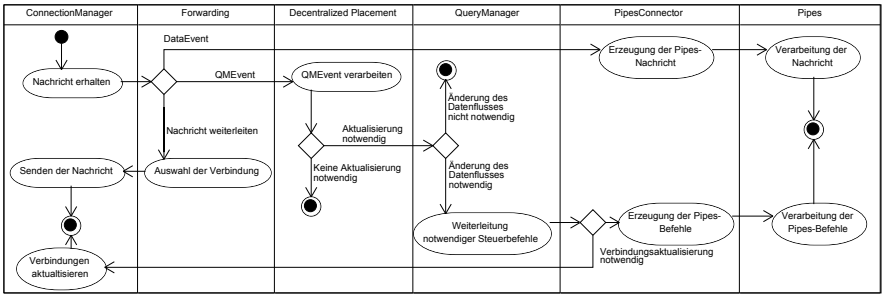


Abbildung 4.4: Ablauf beim Erhalten einer Nachricht

Der Ablauf beim Erhalten einer Nachricht ist in Abbildung 4.4 dargestellt. Wird eine Nachricht über eine Verbindung, die durch den *ConnectionManager* verwaltet wird, empfangen, leitet der *ConnectionManager* die Nachricht an die *Forwarding-Komponente* weiter. Die *Forwarding-Komponente* entscheidet nun, ob eine Nachricht an einen anderen Knoten des Systems weitergeleitet werden muss, oder ob die Nachricht vom eigenen Knoten verarbeitet wird. Muss die Nachricht weitergeleitet werden, wird eine bestehende Verbindung des *ConnectionManagers* ausgewählt und die Nachricht über diese Verbindung gesendet.
Sofern die Nachricht vom eigenen Knoten verarbeitet wird, wird die Nachricht entweder an den *PipesConnector* übertragen falls die Nachricht ein DataEvent ist, oder an das *DP* übertragen falls die Nachricht ein QMEvent ist. Das *DP* verarbeitet das QMEvent und leitet notwendige Informationen an den *QueryManager* weiter, der falls notwendig eine Änderung des Datenfluss mit Hilfe des *PipesConnectors* durchführt. Beispiele für die Änderung des Datenflusses ist das Erstellen oder Löschen von Operatoren. Erhält der *PipesConnector* Informationen zum Ändern des Datenflusses erzeugt dieser die notwendigen Befehle für das *Pipes-System* und übergibt diese an das *Pipes-System*. Das *Pipes-System* führt die erhaltenen Befehle aus. Müssen Verbindungen aktualisiert werden, z.B. Schließen einer nicht mehr benötigten Verbindung, wird diese Information an den *ConnectionManager* übergeben, der die Verbindungen entsprechend aktualisiert. Der *PipesConnector* konvertiert ein DataEvent in das interne Datenformat des *Pipes-Systems*, und leitet die konvertierte Nachricht an das *Pipes-System* weiter. Das *Pipes-System* verarbeitet die konvertierte Nachricht mit den bestehenden Operatoren, hierbei können neue Nachrichten entstehen.

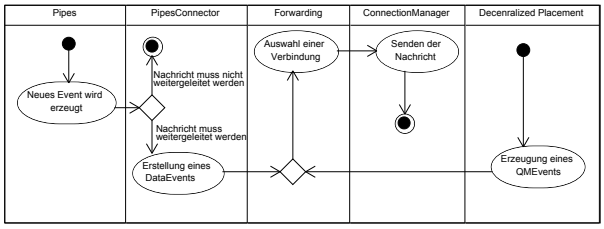


Abbildung 4.5: Ablauf beim Erstellen neuer Nachrichten

Der Ablauf beim Erstellen einer neuen Nachricht ist in Abbildung 4.5 dargestellt. Bei der Anfragebearbeitung werden Daten durch das *Pipes-System* erzeugt bzw. bearbeitet, diese werden an den *PipesConnector* weitergeleitet. Im *PipesConnector* werden erhaltene Daten in ein DataEvent konvertiert und an die *Forwarding-Komponente* übergeben, falls diese an andere Knoten des Systems weitergeleitet werden sollen.
Neben den erzeugten DataEvents erzeugt das *DP* zur Laufzeit periodisch QMEvents, um Informationen, z.B. Position des Knotens mit Nachbarknoten im System auszutauschen. Die QMEvents werden ebenfalls an die *Forwarding-Komponente* weitergeleitet. Die *Forwarding-Komponente* wählt die Verbindung des *ConnectionManagers* aus über die die Nachricht übermittelt wird. Nach der Bestimmung der Verbindung wird die Nachricht übermittelt.

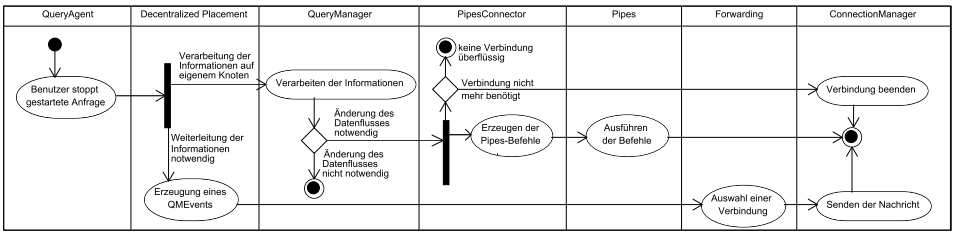


Abbildung 4.6: Ablauf beim Beenden einer Anfrage

Der Ablauf beim Beenden einer Anfrage ist in Abbildung 4.6 dargestellt. Wird eine Anfrage über den *QueryAgent* durch den Benutzer beendet, leitet der *QueryAgent* diese Information an das *DP* weiter. Das *DP* übergibt diese Informationen an den *QueryManager*, falls Operatoren der Anfrage auf dem selben Knoten ausgeführt werden, der diese Information verarbeitet. Werden durch Beenden einer Anfrage Operatoren nicht mehr benötigt, erzeugt der *QueryManager* mit Hilfe des *PipesConnectors* zum einen Pipes-Befehle, die an das *Pipes-System* übergeben und ausgeführt werden. Zum anderen aktualisiert der *PipesConnector* die Verbindungen des Datenflusses. Wird eine Datenfluss-Verbindung nicht mehr benötigt, schließt der *ConnectionManager* die Verbindung. Muss die Information über das Beenden der Anfrage an andere Knoten übertragen werden (weil Operatoren der Anfrage auf anderen Knoten ausgeführt werden), erzeugt das *DP* neue Nachrichten (QMEvents), und übergibt die erzeugten Nachrichten an die *Forwarding-Komponente*. Die *Forwarding-Komponente* wählt eine bestehende Verbindung des *ConnectionManagers* aus und sendet die Nachricht über diese Verbindung.

4.3 Externe Schnittstellen

Im folgenden Abschnitt wird erläutert welche Informationen zwischen den Knoten des Systems übertragen werden.
Es werden sowohl Daten der Datenströmen (DataEvents), als auch Steuernachrichten (QMEvents) zwischen Nachbarknoten gesendet. Anfragen, die vom System bearbeitet werden, werden in einem generischen Anfrageformat an das System übergeben.

QMEvent Durch **Q**uery**M**anagement**Event**s werden Informationen zur Steuerung des Datenflusses und zur Entscheidungsfindung übertragen. Folgende Informationen werden übertragen.

- Informationen für das Forwarding der Nachricht. Hierbei gibt es drei unterschiedliche Informationen
  - ID des Zielknotens
  - Zielkoordinaten
  - ID des Relay-Knotens
- Ursprung der Nachricht,
- Informationen zur Bestimmung der Position eines Knoten im System,
- Position der Datenquellen im System,
- Steuerungsbefehle für den Datenfluss.

DataEvent Mit **DataEvent**s werden die Daten der Operatoren bzw. Datenquellen zwischen den Knoten übertragen. Folgende Informationen werden übertragen.

- Informationen für das Forwarding der Nachricht. Hierbei gibt es drei unterschiedliche Informationen
  - ID des Zielknotens
  - Zielkoordinaten
  - ID des Relay-Knotens
- Ursprung der Nachricht,
- Daten der Datenströme.

Generisches Anfrageformat Das generische Anfrageformat beinhaltet Informationen, die zur Ausführung einer Anfrage im System benötigt werden, folgende Informationen werden gespeichert.

- Jeder Anfrage wird eine eindeutig ID zugeordnet, diese ID wird für jeden Operator gespeichert.
- Für jeden Operator wird ein String erzeugt, der alle Informationen enthält, um den Operator zu erzeugen. Folgenden Bestandteile sind enthalten:
  - Jedem Operator ist eine globale eindeutige ID zugeordnet, durch die ein Operator im System eindeutig identifiziert werden kann
  - Für jeden Operatoren wird der entsprechende Typ des Operators gespeichert.

  - Abhängig vom Typ des Operators sind ggf. weitere Parameter notwendig, die zur Erzeugung des Operatoren notwendig sind, z.B. für den Additionsoperator einen nummerischen Parameter.

- Für jeden Operator werden die eingehenden und ausgehenden Verbindungen gespeichert. Um Duplikate zu vermeiden werden alle Verbindungen in einer gemeinsamen Liste gespeichert. Hierbei unterscheiden sich die Anzahl der Verbindungen der einzelnen Operatoren. Quellen haben nur ausgehende Verbindungen, und keine eingehenden Verbindungen. Senken haben nur genau eine eingehende Verbindungen, und keine ausgehende Verbindungen. Pipes-Operatoren besitzen mindestens eine eingehende und mindestens eine ausgehende Verbindung.

Kapitel 5

Implementierung

Im folgenden Kapitel wird das implementierte System zur Platzierung von Operatoren beschrieben. Es werden die Aufgaben und verwendete Verfahren der einzelnen Komponenten genauer erläutert. Ziel des Systems ist eine initiale Platzierung der Operatoren einer Anfrage in einem verteilten System zu ermöglichen. Zur Platzierung der Operatoren wird das in Abschnitt 3.4 beschriebene Verfahren von Pietzuch et. al. [PLS+06] verwendet, siehe Abschnitt 5.2.1. Um dieses Verfahren verwenden zu können, wird ein dezentrales Verfahren zur Konstruktion eines virtuellen Kostenraums [QL11], siehe Abschnitt 5.3, implementiert um eine Abschätzung der Latenz zwischen den Knoten zu erhalten. Um Informationen sowohl zur Initialisierung des Systems, als auch zur Platzierung der Operatoren weiterleiten zu können wird ein Forwarding-Verfahren [QL11] implementiert. Das Forwarding-Verfahren nutzt Informationen einer verteilten Delaunay-Triangulation, deshalb wurde ein Verfahren [Led07] zur Konstruktion einer Delaunay-Triangulation (DT) implementiert, siehe Abschnitt 5.4.

5.1 Architektur des Prototypen

Aufgabe des *Distributed Placement (DP)* ist zum einen die Bestimmung der Koordinaten der Knoten, zum anderen muss das *DP* Informationen zum Routen der Nachrichten im Netzwerk bereitstellen. Um diese Funktionalität zu gewährleisten, besteht *DP* aus folgenden Komponenten (siehe Abbildung 5.1):

- VirtualPlacement
- CostSpaceManager
- Routing

Das *VirtualPlacement* dient zum einen als Dispatcher der Informationen, die in den QMEvents enthalten sind. Je nach Nachrichtentyp sind unterschiedliche Komponenten beteiligt. Jeder Nachricht ist z.B. die aktuelle Information des Knoten angehängt, der die Nachricht gesendet hat, diese Position wird dem *CostSpaceManager* übergeben.
Zum anderen wird in dieser Komponente das in Abschnitt 5.2.1 beschriebene Verfahren zur Bestimmung der Positionen der Operatoren einer Anfrage durchgeführt. Hierfür werden die übergebenen Informationen des *QueryAgent* verarbeitet, z.B. eine neue Anfrage

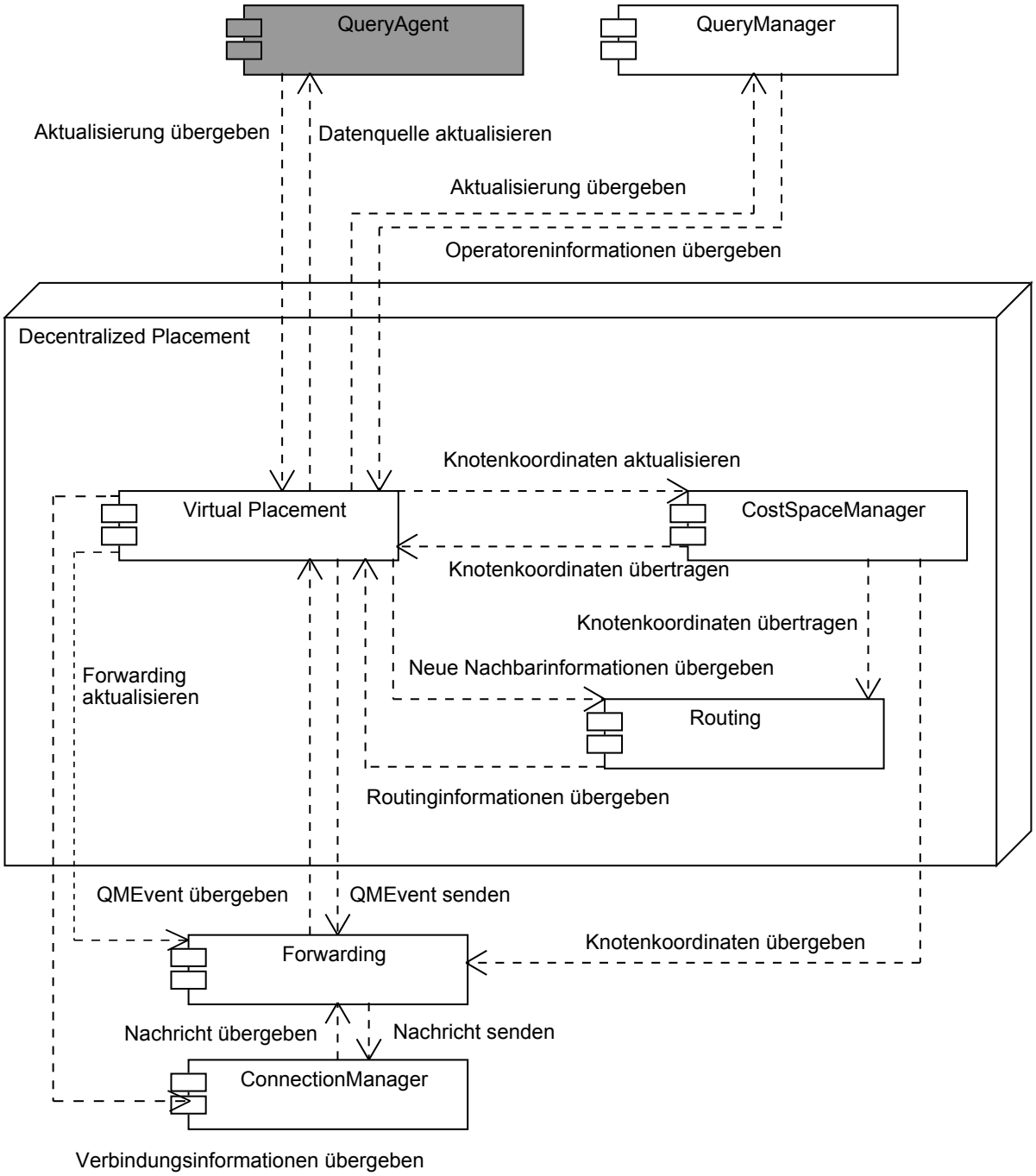


Abbildung 5.1: Architektur des Decentralized Placements

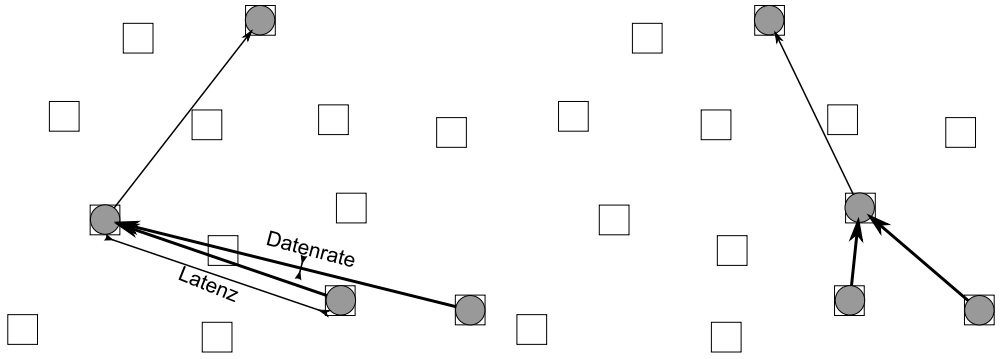
verarbeitet, und ggf. Nachrichten an andere Knoten weitergeleitet.
Aufgabe des *CostSpaceManager* ist zum einen die Positionen der Knoten zu verwalten, zum anderen anhand von Informationen, die durch das *VirtualPlacement* ermittelt werden, die Positionen der Knoten zu aktualisieren.
Aufgabe des Routings ist die nächsten Nachbarknoten eines Knotens in dem Koordinatenraum zu bestimmen. Zur Bestimmung der nächsten Nachbarknoten werden Informationen über Koordinaten der Knoten verwendet, die vom *CostSpaceManager* bereitgestellt werden. Nächste Knoten, zu denen keine direkte Kommunikationsverbindung vorhanden ist, werden als virtuelle Nachbarn bezeichnet.

5.2 Platzierung von Operatoren

Ziel des Systems ist, Anfragen, die an das System gestellt werden, möglichst effizient, d.h. mit möglichst geringer Kommunikation zwischen den einzelnen Knoten bei geringer Ausführungszeit (Latenz), zu bearbeiten. Um dieses Ziel zu erreichen müssen die Operatoren einer Anfrage möglichst optimal auf das Netzwerk verteilt werden. Da der Knoten, der eine Anfrage startet, auch das Ergebnis erhalten soll, werden die Senken einer Anfrage nicht verschoben, sondern verbleiben auf diesem Knoten. Die Datenquellen des Netzwerkes, z.B. Sensoren, sind physisch an einen Knoten gebunden, deshalb können Datenquellen nicht verschoben (siehe Abschnitt 3.2). Andere Operatoren können auf beliebige Knoten im System verteilt werden.

5.2.1 Beschreibung des Verfahrens

Zur Platzierung der Operatoren wird das Verfahren von Pietzuch et. al. [PLS+06] genutzt. Dieses Verfahren verwendet zur Platzierung der Operatoren eine Technik zur Entspannung von Federn, siehe Abbildung 5.2. Operatoren werden als Objekte modelliert, die durch Federn miteinander verbunden sind. Es wird zwischen Operatoren, die nicht verschoben werden können, Senken und Datenquellen, und Operatoren, die verschoben werden können unterschieden. Abhängig von der Ausdehnung der Feder (Latenz) und der Federkonstante (Datenrate) wirkt eine Kraft auf einen Operator. Ziel ist es durch Verschiebung von Operatoren die im System enthaltene Kraft zu minimieren, siehe Abbildung 5.2(b). Eine Anfrage wird also abhängig von der Datenrate und Latenz zwischen den Knoten optimiert, um die Netzwerklast und die Latenz einer Anfrage zu minimieren. Zur Bestimmung der Federausdehnung werden die Positionen der Senken und Datenquellen benötigt. Die Positionen, die zur Platzierung der Operatoren verwendet werden, werden vom *CostSpaceManager* bereitgestellt. Die Distanz zwischen den Operatoren ist eine Abschätzung der Latenzen zwischen den Positionen. Prinzipiell kann zur Angabe der Positionen der Knoten ein beliebiges n-dimensionales metrisches System verwendet werden. Bei der Umsetzung des Ansatzes wurde ein dreidimensionales kartesisches Koordinatensystem verwendet, da dieses auch in [PLS+06] als Beispiel dient.
Zur Durchführung der Platzierung der Operatoren müssen die Positionen der Datenquellen im Netzwerk verbreitet werden. Die Position einer Datenquelle ist hierbei die Position des Knotens auf dem die Datenquelle ausgeführt wird. Ein Knoten kann beliebig viele Datenquellen besitzen.



(a) Ausgangssituation (b) Nach Ausführung des Algorithmus

Abbildung 5.2: Beispiel für die Ausführung des Verfahrens [PLS+06]

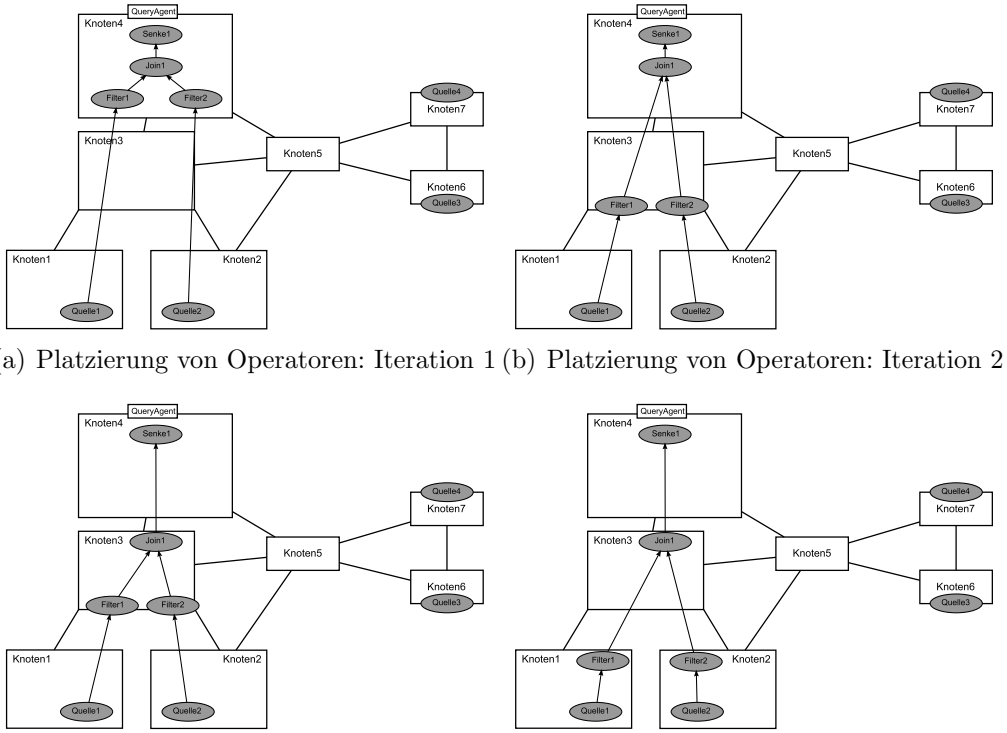
Algorithmus 1 Virtual-Place(S) [PLS+06]

1: **repeat**
2: $\vec{F} = \vec{0}$
3: **for all** S_{parent} **in** $Parents(S)$ **do**
4: $\vec{F_N} = \left(\vec{S} - \vec{S}_{parent}\right) \times DR\left(S, S_{parent}\right)$
5: $\vec{F} = \vec{F} + \vec{F_N}$
6: **end for**
7: **for all** S_{child} **in** $Children(S)$ **do**
8: $\vec{F_N} = \left(\vec{S} - \vec{S}_{child}\right) \times DR\left(S_{child}, S\right)$
9: $\vec{F} = \vec{F} + \vec{F_N}$
10: **end for**
11: $\vec{S} = \vec{S} + \vec{F} \times \delta$
12: **until** $\left|\left|\vec{F}\right|\right| < F_t$

Eine Anfrage, die vom System bearbeitet wird, wird an einen Knoten im generischen Anfrageformat (siehe Abschnitt 4.3) übergeben. Dieser Knoten bestimmt die initiale Position der verschiebbaren Operatoren der Anfrage anhand des Algorithmus 1, der im Folgenden beschrieben wird. Für die Platzierung der Operatoren, müssen sowohl die Vorgänger-Operatoren (parent) als auch die Nachfolger-Operatoren (child) bestimmt werden. Diese Informationen können dem generischen Anfrageformat entnommen werden. Die ideale Platzierung der verschiebbaren Operatoren wird iterativ bestimmt. Bei jeder Iteration wird für jeden verschiebbaren Operator S einer Anfrage, die Position durch einen Verschiebungsvektor F bestimmt, der die Abweichung zur optimale Position festlegt, dieser wird mit dem $\vec{0}$-Vektor initialisiert, siehe Zeile 2. Die Bestimmung von F erfolgt ebenfalls iterativ. Für jeden Vorgänger und Nachfolger wird ein Vektor F_N bestimmt, um die ein Operator verschoben werden muss, um die ideale Position bzgl. des Vorgänger-s/ Nachfolgers zu erreichen. Die Richtung und initiale Länge von F_N wird durch die Position des Operators und des zugehörigen Nachbar-Operators bestimmt, $\left(\vec{S} - \vec{S}_{child}\right)$ bzw. $\left(\vec{S} - \vec{S}_{parent}\right)$. Zusätzlich ist die Länge des Vektors F_N von der Datenrate der Übertragung zwischen zwei Operatoren, DR, abhängig, siehe Zeile 4 bzw. 8. Da im

Moment keine genaue Abschätzung der Datenraten möglich ist, werden die Datenrate der Nachbar-Operatoren durch Konstanten abgeschätzt. Da die Länge des Vektors um die ein Operator für seinen Nachbar-Operator verschoben wird, direkt abhängig von der Datenrate des Nachbar-Operators ist, führt eine höhere Datenrate dazu, dass Operatoren näher an den Nachbar-Operator positioniert werden.
Nach der Bestimmung von F_N für einen bestimmten Nachbarknoten wird dieser Vektor zum Verschiebungsvektor F hinzugefügt, siehe Zeile 5 bzw. 9. Um unnötige Oszillation bei der Verschiebung zu vermeiden, wird der bestimmte Verschiebungsvektor F mit einem konstanten Faktor δ multipliziert, $\vec{F} \times \delta$, siehe Zeile 11. Empirische Untersuchungen [PLS+06] zeigten, dass das Verfahren mit $\delta = 0,1$ funktioniert. Die Iteration zur Bestimmung der Position eines Operators endet, falls die Länge des Verschiebungsvektor den vorgegebene Grenzwert $\left|\left|\vec{F}\right|\right|$ unterschreitet. Wird der Grenzwert unterschritten, bedeutet dies, dass eine gewünschte Genauigkeit bzgl. der idealen Position eines Operators erreicht wurde. Um eine endlose Iteration zu vermeiden, kann eine maximale Anzahl an Iterationen definiert werden, nach der der Algorithmus abgebrochen wird.
Konvergieren alle Positionen der Operatoren einer Anfrage werden die ermittelten Position der Operatoren genutzt um die Operatoren auf den Knoten des Systems zu erstellen. Die Berechnung der Positionen der Operatoren erfolgt auf dem Knoten, der eine Anfrage verarbeitet.
Zur Erstellung der Operatoren werden Nachrichten erzeugt, der alle benötigten Informationen zur Erstellung des Operators angehängt werden, z.B. ID und Position der Vorgänger-Operatoren und weitere Parameter. Ziele der Nachrichten sind die ermittelten Positionen der Operatoren. Das verwendete Forwarding-Protokoll (siehe Abschnitt 5.5.1) garantiert, dass die Nachricht auf dem nächsten Knoten zu den Zielkoordinaten weitergeleitet wird. Erhält ein Knoten eine Nachricht zur Erzeugung eines Operators, registriert der Knoten die Vorgänger-Operatoren durch die an die Nachricht angehängten IDs und Positionen der Vorgänger-Operatoren. Sind alle Vorgänger-Operatoren registriert, wird der Operator erzeugt. Wurden alle Operatoren einer Anfrage erzeugt, startet die Verarbeitung der Anfrage. Wird eine Anfrage beendet, werden nicht mehr benötigte Operatoren gelöscht, hierbei werden ebenfalls die Positionen der Vorgänger-Operatoren genutzt, um das Löschen der Operatoren im System durchzuführen.

5.2.2 Beispiel



(a) Platzierung von Operatoren: Iteration 1 (b) Platzierung von Operatoren: Iteration 2

(c) Platzierung von Operatoren: Iteration 3 (d) Platzierung von Operatoren: Iteration 4

Abbildung 5.3: Möglicher Ablauf der Iteration der Beispielanfrage 3.2(a)

In Abbildung 5.3 sind mögliche Schritte der Berechnung des Algorithmus dargestellt. Abbildung 5.3(a) zeigt den Beginn des Algorihtmus. Die neuen Operatoren werden mit den Koordinaten des Knotens 4 initialisiert und die Position der Quellen wird bestimmt. In den ersten zwei Iterationen werden die Filter-Operationen zu den Quellen verschoben, siehe Abbildung 5.3(b). Aufgrund der geänderten Position der Filter-Operationen wird der Join-Operator ebenfalls verschoben, siehe Abbildung 5.3(c). Die Verschiebung der Operatoren wird solange durchgeführt, bis die Positionen der Operatoren konvergieren, siehe Abbildung 5.3(d). Im Normalfall sollten die Positionen der Operatoren jeder Anfrage konvergieren. Wie bereits in Abschnitt 5.2.1 beschrieben kann jedoch eine maximale Anzahl an Iterationen definiert werden, um endlose Iterationen zu vermeiden. Die bestimmten Operatoren-Position werden genutzt, um die Operatoren an den Knoten zu erstellen. Die Zuordnung zwischen den Positionen der Operatoren und Knoten erfolgt ebenfalls dezentral.

5.2.3 Verfahren zum Lastenausgleich

Ein Knoten besitzt nur begrenzte Ressourcen, z.B. CPU, Arbeitsspeicher usw. Falls Operatoren auf einem überlasteten Knoten ausgeführt werden, führt dies zu einer höheren Latenz und einem geringeren Durchsatz an Events. Es muss deshalb versucht werden die zur Verfügung stehenden Ressourcen bei der Platzierung der Operatoren zu berücksichtigen. Ziel des Lastenausgleich ist erneut ein Kompromiss zwischen Netzwerklast

Algorithmus 2 Verfahren zum Lastenausgleich [PLS+06]

1: Bestimme die k-nächsten Nachbarknoten
2: Kontaktiere einen kleinen Teil der bestimmten Nachbarknoten um ihre laufende Operatoren und Ressourcen zu erfahren
3: Sortiere die bestimmten Nachbarknoten anhand der Distanz zur bestimmten Position des Operators
4: Bestimme den Knoten, der den Operator bereits ausführt
5: Falls kein Knoten den Operator bereits ausführt, bestimme den nächsten Knoten, der genug Ressourcen zur Ausführen des Operators besitzt.

und Latenz, d.h. Operatoren werden nur in einem bestimmten Suchraums des Netzwerkes verschoben. Existiert in diesem Suchraum kein Knoten, der freie Ressourcen zur Ausführung des Operators besitzt, wird der Operator dennoch auf dem überlasteten Knoten erzeugt. Das Verfahren zum Lastenausgleich basiert auf dem Algorithmus 2. Der erste Schritt ist die Bestimmung der nächsten Nachbarknoten, siehe Zeile 1. Die Bestimmung der nächsten Nachbarknoten wird durch die *Routing-Komponente* durchgeführt. Die bestimmten nächsten Nachbarknoten werden kontaktiert, um zu erfragen, ob genug Ressourcen vorhanden sind, siehe Zeile 2. Da keine Multi-Query-Optimierung existiert, haben die bestehenden Operatoren keinen Einfluss auf den Lastenausgleich und werden deshalb nicht berücksichtigt. Entsprechend werden Schritt 3 und 4 übersprungen. Ist ein Nachbarknoten vorhanden, der freie Ressourcen besitzt, wird der Operator auf diesem Knoten erstellt.

5.3 Virtueller Kostenraum

Zur Platzierung der Operatoren im System werden Positionen der Knoten benötigt, die Datenquellen besitzen, und die Position des Knotens, der eine Anfrage bearbeitet. Bei den Positionen der Knoten, die vom System zur Platzierung von Operatoren genutzt wird, handelt es sich nicht um die physische geographische Position der Knoten. Durch die Latenz zwischen den Knoten, wird die Positionen von Knoten im virtuellen Kostenraum abgeschätzt. Der virtuelle Kostenraum ist im System ein dreidimensionales kartesisches Koordinatensystem, entsprechend dem Beispiel aus [PLS+06]. Der virtuelle Kostenraum könnte jedoch durch ein beliebiges n-dimensionales metrisches System umgesetzt werden. Der Abstand der Positionen zwischen zwei Knoten wird durch den euklidischen Abstand bestimmt, und gibt die Abschätzung der Latenz zwischen den zwei Knoten an. Zur Bestimmung der Koordinaten der Knoten wird ein iteratives Verfahren verwendet [QL11]. Hierbei wird anhand der aktuellen Positionen der physischen Nachbarknoten, der aktuellen Latenz zweier Knoten zueinander und einer Schätzung des aktuellen Fehlers der Positionen, die Position eines Knoten aktualisiert. Ähnlich dem Verfahren zur Platzierung der Operatoren, werden Knoten als Objekte modelliert, die durch Federn (physische Verbindung) miteinander verbunden sind. Abhängig von der Ausdehnung der Feder (aktuelle Latenz) wirkt eine Kraft auf einen Knoten. Ziel ist es durch Verschiebung der Positionen der Knoten die im System enthaltene Kraft zu minimieren, und dadurch die Fehler der Abschätzung zu minimieren.
Die Koordinaten werden beim Systemstart ausgehend von einem zufällig ausgewählten

Knoten initialisiert. Um eine möglichst genaue Abschätzung treffen zu können, müssen Knoten, die im virtuellen Kostenraum in direkter Nachbarschaft liegen, Informationen austauschen. Da nicht zu jedem Knoten eine physische Verbindung (virtuelle Nachbarknoten) besteht, müssen Nachrichten durch die *Forwarding-Komponente* über bestehende Verbindungen weitergeleitet werden. Die Informationen, zu welchen Knoten Informationen ausgetauscht werden müssen, wird von der *Routing-Komponente* bestimmt.

5.3.1 Initialisierung des virtuellen Kostenraums

Der virtuelle Kostenraum wird ausgehend von einem Knoten initialisiert [QL11]. Die Position des Knoten, der den virtuellen Kostenraum initialisiert, wird auf den Ursprung des dreidimensionalen kartesischen Koordinatensystems gesetzt. Dieser Knoten schickt seinen physischen Nachbarn eine Nachricht, dass diese ebenfalls ihre Knotenposition initialisieren sollen. Bei der Initialisierung der nicht initialisierten Knoten wird die Anzahl der bereits initialisierten Nachbarknoten bestimmt. Es ist garantiert, dass mindestens ein Knoten bereits initialisiert ist, der Sender der Nachricht zur Initialisierung. Wurde nur ein Nachbarknoten initialisiert, wird eine Kugel konstruiert. Mittelpunkt der Kugel ist der bereits initialisierte Nachbarknoten. Radius der Kugel ist die aktuelle Latenz zwischen dem initialisierten Nachbarknoten und den zu initialisierenden Knoten. Nun wird eine beliebige Position auf der Oberfläche der konstruierten Kugel ausgewählt. Die Position des zu initialisierenden Punktes wird gleich der ausgewählten Position gesetzt. Sind bereits zwei oder mehr Nachbarknoten initialisiert, werden die zwei initialisierten Nachbarknoten bestimmt, deren Abstand am größten ist. Von diesen beiden Knoten wird der Mittelpunkt der Strecke, die die beiden Knoten verbindet, bestimmt. Da bei der Berechnung von Informationen der Nachbarschaft in der Routing-Komponente Spezialfälle auftreten, siehe Abbildung A.3, falls drei Positionen von Nachbarknoten auf einer Linie liegen, wird die Position des Knoten nicht mit dem Mittelpunkt initialisiert, sondern wird zufällig um einen kleinen Betrag in zufälliger Richtung verschoben. Neben den Koordinaten wird der abgeschätzte Fehler der Koordinate mit einer festgelegten Konstante initialisiert.

5.3.2 Aktualisierung des virtuellen Kostenraums

Da die Positionen der Knoten durch die Latenz zwischen den Knoten abgeschätzt werden, und sich die Latenzen z.B. durch eine hohe Auslastung der Knoten ändern können, muss jeder Knoten seine Position periodisch aktualisieren. Zur Aktualisierung der Positionen wird das in Algorithmus 3 dargestellte Verfahren verwendet. Für jede Aktualisierung wird ein Fehlerwert e_{sum} bestimmt. Der Fehlerwert gibt an wie genau die aktuelle Abschätzung der Positionen ist. e_{sum} wird zu Beginn mit $e_{sum} = 0$ initialisiert, siehe Zeile 1. Für jeden Knoten werden seine Koordinaten anhand der Koordinaten und Latenzen seiner physischen und virtuellen Nachbarknoten aktualisiert. Dabei wird für jeden Nachbarknoten v eines Knoten u ein Wert f für das Vertrauen in die Aktualisierung berechnet. Der Vertrauenswert f wird anhand des Fehlers der eigenen Koordinaten e_u und der Fehler der Koordinaten des Nachbarknotens e_v (siehe Zeile 5) berechnet, $f = \frac{e_u}{(e_u+e_v)}$, siehe Zeile 6. Anhand der im virtuellen Kostenraum geschätzten Distanz $\tilde{D}(u, v)$ zwischen u und v und der aktuellen Latenz $D(u, v)$ wird der aktuelle absolute Fehler e der berechneten Koordinaten ermittelt, $e = D(u, v) - \tilde{D}(u, v)$. Abhängig vom absoluten Fehler e, dem

Algorithmus 3 Verfahren zur Aktualisierung des virtuellen Kostenraum [QL11]

1: $e_{sum} = 0$;
2: **for all** Knoten v in $P_u \cup N_u$ **do**
3: **if** ($v \in P_u$ **and** $\tilde{D}(u,v) > D(u,v)$) **or** $v \in N_u \setminus P_u$ **then**
4: t = Eintrag in F_u, das $t.dest = v$;
5: $e_v = t.error$;
6: $f = \frac{e_u}{(e_u+e_v)}$;
7: $e = D(u,v) - \tilde{D}(u,v)$
8: $x_u = x_u + c_c \times f \times e \times u(x_u - x_v)$;
9: $e_{sum} = e_{sum} + \frac{|e|}{\tilde{D}(u,v)}$;
10: **end if**
11: **end for**
12: $e_{new} = \frac{e_{sum}}{|P_u \cup N_u|}$;
13: $e_u = e_u \times (1 - c_e) + e_{new} \times c_e$;
14: Übertrage die aktualisierten x_u und e_u zu allen Knoten in $P_u \cup N_u$;

Vertrauenswert f und einer Konstanten c_c wird nun ein Verschiebungsvektor bestimmt. Dieser Verschiebungsvektorektor bestimmt die Änderung der aktuellen Position des Knotens $x_u = x_u + c_c \times f \times e \times u(x_u - x_v)$, siehe Zeile 8.
Der Term $u(x_u - x_v)$ berechnet den Einheitsvektor mit Richtung von Knoten v zu Knoten u mit Hilfe entsprechenden Koordinaten x_v bzw. x_u. Das Produkt aus Einheitsvektor $u(x_u - x_v)$ und absoluten Fehler e bestimmt die Richtung des Verschiebungsvektor. Ist die aktuelle Latenz $D(u,v)$ größer als die geschätzte Latenz $\tilde{D}(u,v)$, wird die Position des Knoten u von der Position des Knotens v weg, ansonsten zur Position des Knotens v hin bewegt. Um Oszillationen zu vermeiden wird die Länge des Verschiebungsvektor durch die Konstante c_c beeinflusst. Um die Genauigkeit der aktuellen Position bestimmen zu können wird ebenfalls der Fehlerwert der Position aktualisiert. Hierzu wird nach der Aktualisierung der relative Fehler der geänderten Position bestimmt $\frac{|e|}{\tilde{D}(u,v)}$. Dieser Wert wird nach Bestimmung zum gesamten Fehler e_{sum} addiert, siehe Zeile 7. Wurde die Position des Knotens anhand aller Nachbarknoten aktualisiert wird der Mittelwert e_{new} der summierten Fehlerwerte e_{sum} gebildet, $e_{new} = \frac{e_{sum}}{|P_u \cup N_u|}$, siehe Zeile 12. Nach Aktualisierung der Koordinaten anhand aller Nachbarknoten wird der Mittelwert der Summe der Fehler der einzelnen Anpassung bestimmt. Der Fehler der aktuellen Koordinate wird nun anhand des neuen Fehlers e_{new}, einer Konstante c_e und dem alten Fehler e_u bestimmt, $e_u = e_u \times (1 - c_e) + e_{new} \times c_e$, siehe Zeile 13. Die Konstante c_e bestimmt das Verhältnis zwischen dem mittleren Fehlerwert der Aktualisierung e_{new} und dem bisherigen Fehler e_u bei der Berechnung des neuen Fehlerwerts der aktualisierten Position. Wurde die Aktualisierung durchgeführt, wird die aktualisierte Position an die Nachbarn gesendet. Um keine zusätzlichen Nachrichten zu erzeugen wird die aktualisierten Information an Nachrichten, die gesendet werden, z.B. Nachrichten zur Initialisierung des Systems, angehängt.

5.3.3 Bestimmung der Latenz

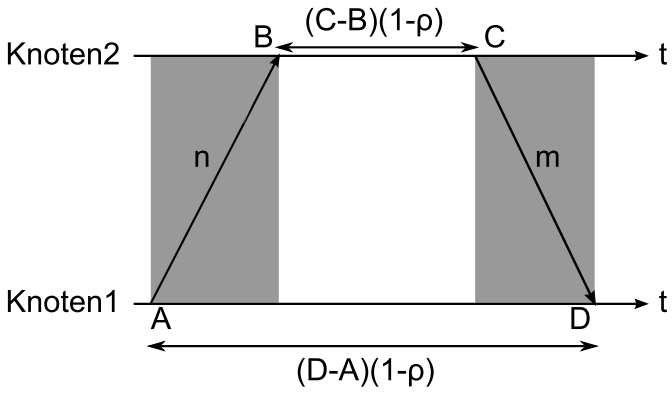


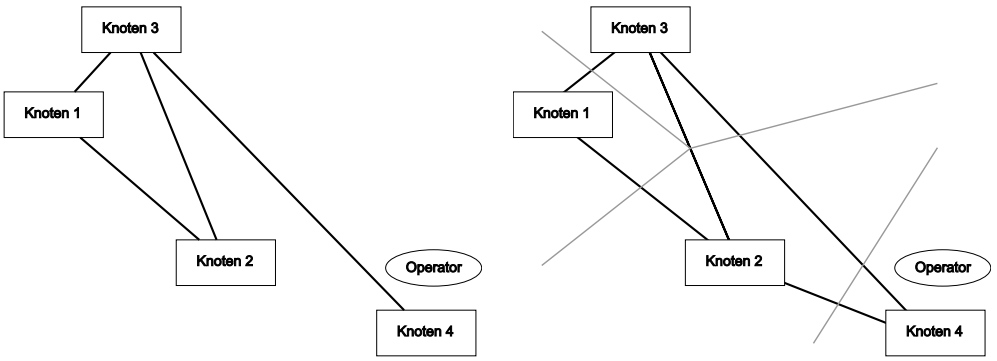
Abbildung 5.4: Nachrichtenaustausch zur Bestimmung der Latenz

Zur Aktualisierung der Position der Knoten im virtuellen Kostenraum werden Latenzen zwischen Nachbarknoten benötigt. Zur Bestimmung der Latenz wird das in [JFF07] beschriebene Verfahren verwendet. Vorteil dieses Verfahrens ist, dass keine Synchronisation der Systemuhren benötigt wird. Der Ablauf zur Bestimmung der Latenz ist in Abbildung 5.4 dargestellt. Es werden zwei Nachrichten n und m zur Bestimmung der Latenzen zweier Nachbarknoten gesendet. Knoten 1 schickt seinem Nachbarknoten 2 eine Nachricht, hierbei wird sowohl der Zeitpunkt A des Sendens der Nachricht in Knoten 1, als auch der Zeitpunkt B des Nachrichtenempfangs in Knoten 2 gespeichert. Knoten 2 bestimmt die Bearbeitungszeit $C - B$ der Nachricht n, hängt diese Information an Nachricht m an und sendet die Nachricht m an Knoten 1. Knoten 1 speichert den Zeitpunkt D des Nachrichtenerhalts der Nachricht m. Mit dem gespeichert Zeitpunkt A und den Informationen in der Nachricht m kann die Latenz, $(D - A)(1 - \rho) - (C - B)(1 - \rho)$, bestimmt werden. ρ ist hierbei ein konstanter Faktor, der den Drift der Systemuhren der zwei Knoten abschätzt. Um robust gegenüber temporären Extremwerten zu sein, werden vier Latenz-Werte mit dem beschriebenen Verfahren ermittelt, die zusammen aggregiert werden. Diese Berechnung liefert nicht die genaue Latenz zweier Knoten, sondern definiert eine obere Grenze, die ein korrekter Nachrichtenaustausch nicht überschreiten würde [JFF07].

5.4 Routing

Aufgabe des Systems ist die Platzierung von Operatoren durch das Verfahren aus Abschnitt 5.2.1, dazu werden Koordinaten innerhalb des virtuellen Kostenraums ermittelt, an dem Operatoren idealerweise ausgeführt werden sollen. Da das System zur Anfragebearbeitung dezentral ist, verfügt kein Knoten über die gesamte Netzwerkstruktur, dementsprechend muss auch die Zuordnung zwischen den idealen Operatorpositionen und Knoten dezentral erfolgen.
Um dies zu ermöglichen müssen die Informationen zur Platzierung eines Operators dezentral durch das Netzwerk geroutet werden. Zum Routen der Informationen kann ein Greedy-Routing-Algorithmus verwendet werden. Hierbei wird eine Nachricht immer an den Knoten weitergeleitet, der die Distanz zum Ziel minimiert.

Ein Problem, dass bei dem Greedy-Routing-Algorithmus auftreten kann, ist, dass eine Nachrichten in ein lokales Optimum geroutet werden kann, dies jedoch nicht erkannt wird. Ein Beispiel wird in Abbildung 5.5 dargestellt.



(a) Knotensystem

(b) Delaunay-Triangulation des Knotensystem

Abbildung 5.5: Anwendungsbeispiel

Angenommen Knoten 1 soll einen Operator im System platzieren. Die optimale Position des Operators wird ermittelt. Die Informationen zur Erzeugung des Operators muss nun zum nächsten Knoten weitergeleitet werden. Zur Weiterleitung der Information bestimmt Knoten 1 den nächsten physischen Nachbarknoten zur optimalen Position des Operators, Knoten 2, und leitet die Informationen an Knoten 2 weiter. Knoten 2 bestimmt erneut den nächsten physischen Nachbarknoten zur optimalen Position des Operators. Durch die vorhandenen Informationen erkennt Knoten 2 sich selbst als nächsten Knoten. Knoten 4 ist näher an der optimalen Position des Operators, da keine physische Verbindung zwischen Knoten 2 und 4 vorhanden ist wird dies von Knoten 2 jedoch nicht erkannt. Die Information befindet sich dementsprechend in einem lokalen Optimum. Zur Erkennung dieser Situation müssen Informationen zwischen Knoten 2 und 4 ausgetauscht werden.

5.4.1 Einführung der Delaunay-Triangulation

Um einen Nachrichtenaustausch zwischen den nächsten Nachbarknoten zu ermöglichen bestimmt die *Routing-Komponente* die nächsten Nachbarn des Knotens im virtuellen Kostenraum. Hierzu wird eine DT für das System konstruiert, siehe Abbildung 5.5(b). Hierbei werden bei Erstellung der DT Routing-Informationen ermittelt, um den Nachrichtenaustausch zwischen Nachbarknoten der DT zu ermöglichen, falls keine physische Verbindung zwischen den Knoten besteht. In dem Beispiel besitzt Knoten 2 nun die Knotenkoordinaten des Knoten 4 und kann die Nachrichten über den Knoten 3 anhand bestehender Routing-Informationen weiterleiten. Allgemein besitzt die DT zwei wesentliche Eigenschaften, die zur korrekten Ausführung der Platzierung der Operatoren benötigt werden. Zum einen garantiert die DT, dass Greedy-Routing immer erfolgreich ist, da in einer DT zwischen den nächsten Nachbarn eine Verbindung vorhanden ist. Zum anderen, dass zu einem gegebenen Punkt der nächste Knoten des System gefunden wird [LL06]. Zur Definition der DT wird zu erst der Begriff des Voronoi-Diagramms definiert [Aur91]. Das Voronoi-Diagramm einer Punktmenge S eines n-dimensionalen euklidischen Raums

ist eine Unterteilung des Raumes in $|S|$ Voronoi-Zellen. Voronoi-Zellen sind n-dimensionale Polytope, wobei gilt, dass jeder Punkt $u \in S$ der nächste Punkt von S zu allen Punkten innerhalb seiner Voronoi-Zelle $VC_s(u)$ ist. Folglich gilt für die Voronoi-Zelle $VC_s(u)$ eines Punktes $u \in S$:

$$VC_s(u) = \{p \,|D(p,u) \leq D(p,w), \forall w \in S\}$$

wobei $D(u,v)$ die Distanz zwischen den Punkten u und v definiert. Ein n-dimensionales Polytop wird hierbei von (n-1)-dimensionelen Facetten beschränkt. In dieser Arbeit wird ein dreidimensionaler euklidischer Raum betrachtet, daraus folgt, dass die Voronoi-Zellen Polyeder sind, die von Polygonen begrenzt werden. Die Punkte der Punktmenge $u, v \in S$ sind entsprechend die Knoten des Systems.
Ausgehend von der Definition des Voronoi-Diagramms wird die DT wie folgt definiert.
Eine DT [LL06] einer Punktmenge S eines n-dimensionalen euklidischen Raums ist ein Graph in dem zwei Punkte u und v der Punktmenge S eine gemeinsame Kante haben, genau dann wenn die zugehörigen Voronoi-Zellen der Punkte $VC_s(u)$ und $VC_s(v)$ benachbart sind. Im dreidimensionalen Raum bildet die DT Tetraeder aus den Punkten der DT. Ist ein Punkt in einem Tetraeder enthalten, bedeutet dies, dass die anderen Punkte des Tetraeders nächste Nachbarn zu diesem Punkt sind. In dem zweidimensionalen Beispiel, siehe Abbildung 5.5(b), werden keine Tetraeder, sondern Dreiecke konstruiert. Die Bedeutung ist jedoch gleich, Punkte in einem Dreieck sind nächste Nachbarknoten zueinander. Die Punkte u und v sind benachbart zueinander, falls die Voronoi-Zellen $VC_s(u)$ und $VC_s(v)$ benachbart sind. Die Begrenzungen der Voronoi-Zellen halbieren die Kanten der DT senkrecht. Entsprechend besteht eine Dualität zwischen einer DT und den zugehörigen Voronoi-Zellen.
Besitzen zwei Voronoi-Zellen $VC_s(u)$ und $VC_s(v)$, zweier Punkte u und v der Punktmenge S eine gemeinsame begrenzende (n-1)-dimensionale Facette, so besitzt die zugehörige DT eine gemeinsame Kante zwischen den Punkten u und v. Besitzen zwei Voronoi-Zellen dagegen nur eine d-Facette (d< n-1), so ist die zugehörige DT nicht mehr eindeutig [Led07], es kann eine Kante existieren, muss aber nicht.
Eine verteilte Delaunay-Triangulation (DDT) [LQ11] einer Menge von Knoten S ist festgelegt als $\{\langle u, N_u\rangle \,|u \in S\}$, wobei N_u die benachbarten Voronoi-Zellen des Knoten u darstellt, die von u lokal bestimmt wurden. Eine DDT ist korrekt, genau dann wenn die folgenden zwei Kriterien für jedes Paar von Knoten u und v der Knotenmenge S erfüllt werden. Falls in der globalen DT eine Kante zwischen den Knoten u und v existiert, dann gilt $v \in N_u$ und $u \in N_v$. Existiert keine Kante zwischen den Knoten u und v in der globalen DT, dann gilt $v \notin N_u$ und $u \notin N_v$, d.h. ist die DDT korrekt, sind in der lokalen DT eines Knotens nur die Verbindungen des Knotens enthalten, die auch in der globalen DT vorhanden sind. Zur korrekten Initialisierung der DDT ist es notwendig, dass ein Knoten seinen Nachbarknoten die Informationen übermittelt, ob der Knoten bereits Teil der DDT ist. Ähnlich der Übermittlung der aktuellen virtuellen Position, wird diese Information einer gesendeten Nachricht angehängt.
Da im System nicht zu jedem Knoten eine Verbindung vorhanden ist, muss zusätzlich entsprechend ein Forwarding-Pfad zwischen den einzelnen DDT-Nachbarn vorhanden sein [QL11].
Da sich die Positionen der Knoten bei Veränderung der Latenzen ebenfalls ändert, müssen die lokalen DT der Knoten der DDT periodisch neu berechnet werden.

5.4.2 Initialisierung der Delaunay-Triangulation

Der Algorithmus 4 zur Konstruktion der DT erstellt die DT inkrementell, aus einem künstlichen Tetraeder. Bedingung ist, dass alle Punkte, die man zur DT hinzufügt in einem künstlichen Tetraeder enthalten sind. Zur Konstruktion des benötigten initialen Tetraeders wird ein eigenes Verfahren verwendet. Um eine möglichst symmetrische Abdeckung des virtuellen Kostenraums zu erreichen wird eine Kugel um den Ursprung des virtuellen Kostenraums konstruiert. Die Eckpunkte des initialen Tetraeders werden nun so bestimmt, dass
Kugel wird abhängig von den zu erwartenden Positionen der Knoten gewählt. Ein kleiner Radius erhöht hierbei die Genauigkeit der durchgeführten Berechnungen. Ist keine Abschätzung bekannt, wird der Radius mit einem vorgegebenen Grenzwert initialisiert.

5.4.3 Berechnung der Delaunay-Triangulation

Algorithmus 4 InsertOnePoint(T,p) [Led07]

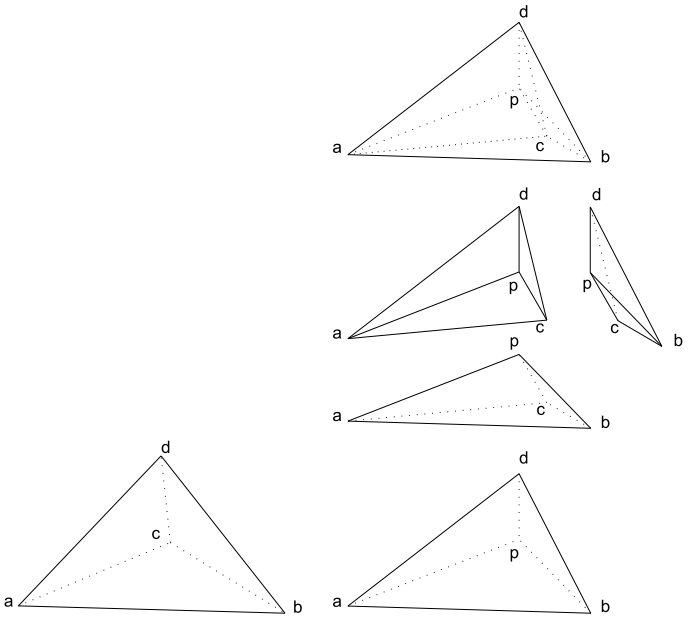
Eingabe: $DT(S)$ T in $\mathbb{R}^3$ und einen Punkt p der zu T hinzugefügt werden soll
Ausgabe: $T^p = T \cup \{p\}$

1: τ = Tetraeder, das den Punkt p enthält, bestimmen;
2: Führe eine Flip14-Operation mit dem Tetraeder τ und dem Punkt p aus;
3: Füge die vier entstehenden Tetraeder auf einen Stack hinzu;
4: **while** Stack ist nicht leer **do**
5: $\tau = [a, b, c, p]|$ Tetraeder τ vom Stack entfernen;
6: $\tau_a = [a, b, c, d]|$ Benachbartes Tetraeder τ_a von Tetraeder τ mit der gemeinsamen Fläche a, b, c bestimmen;
7: **if** Punkt d in der Umkugel von τ **then**
8: FLIP(τ, τ_a);
9: **end if**
10: **end while**

Ziel des Algorithmus 4 ist die DT einer Punktmenge P zu berechnen. Die Punktmenge P repräsentiert hierbei die Positionen der nächsten Nachbarknoten eines Knotens im System. Hierfür werden die einzelne Punkte $p \in P$ inkrementell durch die Ausführung des Algorithmus 4 in die Delaunay-Triangulation einer Punktmenge S ($DT(S)$) hinzugefügt. Vor der Ausführung wird durch die Initialisierung ein künstliches Tetraeder geschaffen, dies garantiert, dass die DT, die als Eingabe für den Algorithmus benötigt wird vorhanden ist. Ist ein $p \in P$ in dem künstlichen Tetraeder enthalten, so ist ebenfalls garantiert, dass ein Tetraeder existiert, das den Punkt p enthält. Liegt ein $p \in P$ jedoch außerhalb des künstlichen Tetraeders, so kann der Punkt p nicht verarbeitet werden, da kein Tetraeder der DT existiert, das den Punkt enthält. Zu Beginn der Ausführung wird das Tetraeder $\tau \in DT(S)$ bestimmt, das den Punkt p, der zur $DT(S)$ hinzugefügt werden soll, enthält, siehe Zeile 1.

Lemma 1. *Das Tetraeder τ ist ein variables Tetraeder, das bereits Teil der Delaunay-Triangulation ist.*

Anschließend wird mit dem Tetraeder τ und dem Punkt p eine Flip14-Operation durchgeführt, siehe Zeile 2. Die Flip14-Operation ist eine der vier Operationen, die zur inkrementellen Konstruktion der DT benötigt wird.



(a) Beliebiges Tetraeder (b) Ergebnis der Flip14-Operation

Abbildung 5.6: Beispiel für eine Flip14-Operation

Bei der Flip14-Operation werden aus einem bestehenden Tetraeder durch Hinzufügen eines weiteren Punkts vier neue Tetraeder konstruiert. Voraussetzung für die Flip14-Operation ist, dass sich der Punkt in dem Tetraeder befindet. Zur Erstellung der vier neuen Tetraeder wird jede der vier Seitenfläche des Tetraeders mit dem Punkt verbunden.

Lemma 2. *Die Punkte a, b, c und d sind variable Punkte, die bereits zur Delaunay-Triangulation hinzugefügt wurden.*

Ein Beispiel hierfür ist in Abbildung 5.6 dargestellt. Hierbei werden aus dem Tetraeder, $\tau = [a, b, c, d]$, siehe Abbildung 5.6(a), durch das Hinzufügen des Punktes p vier neuen Tetraeder konstruiert. Punkt p liegt hierbei im Tetraeder τ. Sowohl das Tetraeder τ, als auch die vier Punkte a, b, c und d, sind hierbei Platzhalter für ein Tetraeder bzw. Punkte der DT. Wie beschrieben werden hierfür die vier Flächen des Tetraeders $[a, b, c]$, $[b, c, d]$, $[b, c, d]$, $[a, b, d]$ mit dem Punkt p kombiniert. Die hierbei entstehenden Tetraeder $\tau_1 = [a, b, c, p]$, $\tau_2 = [b, c, d, p]$, $\tau_3 = [a, c, d, p]$, $\tau_4 = [a, b, d, p]$, werden in Abbildung 5.6(b) dargestellt.
Die vier entstehenden Tetraeder τ_1, τ_2, τ_3 und τ_4,werden zur weiteren Bearbeitung einem Stack hinzugefügt, siehe Zeile 3. Anschließend wird eine Schleife durchlaufen, solange Tetraeder auf dem Stack enthalten sind, siehe Zeile 4. Bei jedem Durchlauf der Schleife wird ein Tetraeder $\tau = [a, b, c, p]$ vom Stack entfernt, siehe Zeile 5. Zu dem Tetraeder τ wird nun das Tetraeder $\tau_a = [a, b, c, d]$ bestimmt, dass die Fläche $[a, b, c]$ mit τ teilt, siehe Zeile 6. Ist die Spitze d des Tetraeders τ_a in der Umkugel des Tetraeders τ enthalten, muss

ggf. eine weitere Flip-Operation an den beiden Tetraedern τ und τ_a durch Ausführung des Algorithmus 5 durchgeführt werden, siehe Zeilen 7 und 8. Umkugel ist hierbei eine Kugel, auf deren Oberfläche alle Eckpunkte eines Tetraeders liegen. Aufgabe des Algorithmus 5 ist Fehler in der DT, die durch Hinzufügen des Punktes p entstanden sind, zu beheben. Hierbei können neue Tetraeder entstehen, die ebenfalls dem Stack des Algorithmus hinzugefügt werden. Sind keine Tetraeder mehr auf dem Stack enthalten, endet die Ausführung des Algorithmus, und der Punkt p wurde der $DT(S)$ hinzugefügt.

5.4.4 Flip-Operationen

Algorithmus 5 Flip(T,p)[Led07]

Eingabe: Zwei benachbarte Tetraeder $\tau = [a, b, c, p]$ und $\tau_a = [a, b, c, d]$

1: **if** Fall1 **then**
2: Flip23(τ,τ_a);
3: Füge die entstehenden Tetraeder($\tau_1 = [a, b, d, p]$, $\tau_2 = [b, c, d, p]$, $\tau_3 = [a, c, d, p]$) dem Stack hinzu;
4: **else if** Fall2 und in T^p ist das Tetraeder $\tau_b = [a, b, d, p]$ enthalten **then**
5: Flip32(τ,τ_a, τ_b)
6: Füge die entstehenden Tetraeder($\tau_1 = [a, c, d, p]$, $\tau_2 = [b, c, d, p]$) dem Stack hinzu;
7: **else if** Fall3 und τ, τ_a, sind mit zwei weiteren Tetraeder $\tau_b = [b, c, d, e]$, $\tau_c = [b, c, e, d]$ in Konfiguration44 **then**
8: Flip44(τ, τ_a, τ_b, τ_c);
9: Füge die entstehenden Tetraeder($\tau_1 = [a, b, e, p]$, $\tau_2 = [a, b, e, d]$, $\tau_3 = [a, c, e, p]$, $\tau_4 = [a, c, e, d]$) dem Stack hinzu;
10: **else if** Fall4 **then**
11: Flip23(τ,τ_a);
12: Füge die entstehenden Tetraeder($\tau_1 = [a, b, d, p]$, $\tau_2 = [b, c, d, p]$, $\tau_3 = [a, c, d, p]$) dem Stack hinzu;
13: **end if**

Befindet sich bei der Ausführung des Algorithmus 4 die Spitze d eines Tetraeders $\tau_a = [a, b, c, d]$ in der Umkugel des Tetraeders $\tau = [a, b, c, p]$ muss überprüft werden, ob eine weitere Bearbeitung der Tetraeder notwendig ist, dabei können drei Operationen Flip23-, Flip32- und die Flip44-Operation, ausgeführt werden. An dieser Stelle wird auf eine genaue Beschreibung der Operationen verzichtet, diese erfolgt im Anhang A. Anhand der zwei Tetraeder τ und τ_a, die an den Algorithmus 5 übergeben werden, und den bestehenden Tetraedern in $DT(S)$, wird bestimmt, ob und ggf. welche Operation ausgeführt werden muss. Wird eine Flip-Operation ausgeführt, werden die neu entstehende Tetraeder dem Stack des Algorithmus 4 hinzugefügt.

5.5 Forwarding

Aufgabe der *Forwarding-Komponente* ist zum einen für Nachrichten des Kontroll- und Datenfluss, die zu anderen Knoten weitergeleitet werden sollen, die Verbindung auszuwählen

über die eine Nachricht geschickt werden soll. Zum anderen wird für empfangene Nachrichten bestimmt, ob eine Nachricht weitergeleitet werden muss oder verarbeitet werden soll. Wird eine Nachricht vom selben Knoten verarbeitet wird eine Nachrichten des Datenflusses an den *PipesConnector* übertragen, Nachrichten des Kontrollflusses werden an das *DP* übergeben, siehe Abbildung 4.2.

5.5.1 Forwarding-Protokoll

Zur Initialisierung des Systems und zur Platzierung der Operatoren müssen Nachrichten zwischen Knoten des Systems ausgetauscht werden. Erhält ein Knoten eine Nachricht muss entschieden werden, ob und wie ein Knoten die Nachrichten weiterleitet, oder die Nachricht selbst verarbeitet. Diese Entscheidung wird vom implementierten Forwarding-Protokoll [LQ11] (siehe Tabelle 5.1) getroffen. Für die Ausführung des Forwarding-Protokolls ist eine Forwarding-Tabelle notwendig, siehe Abschnitt 5.5.2.

	Bedingung	Aktion
1.	$u = m.ZielID$	Kein Forwarding der Nachricht (Knoten u ist der Zielknoten)
2.	Es gibt einen Knoten v in P_u und $v = m.ZielID$	Übertrage m an den Knoten v (Knoten v ist der Zielknoten)
3.	$m.RelayID \neq null$ und $m.RelayID \neq u$	Finde einen Eintrag t in der Forwarding-Tabelle F_u mit $t.Ziel = m.ZielID$, übertrage m zu $t.Nachfolger$.
4.	Es existiert ein Knoten v in $P_u \cup \{u\}$ am nächsten zu $m.Zielkoordinaten$, $v \neq u$	Übertrage m an v (Greedy 1)
5.	Es existiert ein Knoten v in $N_u \cup \{u\}$ am nächsten zu $m.Zielkoordinaten$, $v \neq u$	Finde einen Eintrag t in der Forwarding-Tabelle F_u mit $t.Ziel = v$, übertrage m zu $t.Nachfolger$. (Greedy 2)
6.	Trifft keine der Bedingungen 1-5 zu	Kein Forwarding der Nachricht(Knoten u ist am nächsten zum Ziel)

Tabelle 5.1: MDT-Forwarding-Protokoll [LQ11] auf einem Knoten u für eine Nachricht m

Wird eine Nachricht erstellt, wird neben den zu übertragenden Daten ebenfalls Informationen in die Nachricht integriert, die für das Forwarding der Nachricht genutzt werden. Eine Nachricht besitzt drei Einträge die zur Weiterleitung der Nachricht gesetzt werden können, ZielID, RelayID und die Zielkoordinaten. Dabei muss mindestens entweder die ZielID oder die Zielkoordinaten enthalten sein, die RelayID ist optional. Die RelayID wird zur Weiterleitung einer Nachricht an virtuelle Nachbarn benutzt.
Virtuelle Nachbarknoten sind Knoten, zu der eine Verbindung in der DDT vorhanden ist, jedoch keine physische Verbindung. Ist eine physische Verbindung zu eine Knoten vorhanden, ist der Knoten ein physischer Nachbar.
Ein Beispiel bei dem keine ZielID vorhanden ist, ist die Platzierung von Operatoren,

man besitzt zwar eine Position an der der Operator ausgeführt werden soll, da man aber keine genauen Informationen über das Netzwerk besitzt, kann das Ziel nur anhand der Koordinaten weitergeleitet werden.
Das Forwarding einer Nachricht wird anhand des in Tabelle 5.1 dargestellten Verfahrens durchgeführt. Ist in der Nachricht m eine ZielID vorhanden wird überprüft, ob der aktuelle Knoten der Zielknoten ist. Falls dies zutrifft, wird die Nachricht von dem Knoten selbst verarbeitet (1.Zeile).
Trifft diese Bedingung nicht zu, wird überprüft, ob ein physischer Nachbarknoten der Zielknoten ist. Falls ein physischer Nachbarknoten der Zielknoten ist, wird die Nachricht direkt an diesen Knoten weitergeleitet (2.Zeile).
Falls kein physischer Nachbarknoten Zielknoten der Nachricht ist, wird überprüft, ob die Nachricht eine RelayID besitzt, und diese nicht mit der ID des aktuellen Knotens übereinstimmt, trifft dies zu wird versucht die Nachricht anhand der RelayID weiterzuleiten. Hierbei wird anhand der Informationen der Forwarding-Tabelle bestimmt an welchen Knoten die Nachricht weitergeleitet werden soll (3.Zeile). Falls die ID des Knotens identisch mit der RelayID ist, wird diese gelöscht, da der virtuelle Nachbar erreicht wurde.
Konnte die Nachricht nicht weitergeleitet werden, werden die Zielkoordinaten zur Bestimmung der Koordinaten verwendet. Ähnlich eines Greedy Routing-Algorithmus wird versucht, die Distanz zwischen Ursprung und Ziel der Nachricht bei jeder Weiterleitung zu minimieren. Es wird bestimmt, ob ein physischer Nachbarknoten näher zu den Zielkoordinaten ist, als der aktuelle Knoten. Ist dies der Fall wird die Nachricht an diesen Punkt weitergeleitet (4.Zeile). Ist der aktuelle Knoten näher an den Zielkoordinaten als seine physischen Nachbarknoten wird der gleiche Vorgang für die virtuellen Nachbarn durchgeführt (5.Zeile). Erfolgt eine Weiterleitung an einen virtuellen Nachbarn wird die RelayID der Nachricht m mit der ID des virtuellen Nachbars gleichgesetzt, um Zyklen bei der Weiterleitung der Nachrichten zu vermeiden. Erfolgt keine Weiterleitung bedeutet dies, dass der Knoten der nächste Knoten zu den Zielkoordinaten ist, dementsprechend wird die Nachricht vom Knoten selbst verarbeitet (6.Zeile).

5.5.2 Informationen der Forwarding-Tabelle

Zur Weiterleitung der erhaltenen Nachrichten benötigt das Forwarding-Protokoll, siehe Abschnitt 5.5, Informationen wie die Weiterleitung erfolgen soll. Diese Informationen werden in der Forwarding-Tabelle gespeichert. In der Forwarding-Tabelle werden Pfade gespeichert, die in der DDT, siehe Abschnitt 5.4, vorhanden sind. Hierbei wird nicht der gesamte Pfad gespeichert, sondern lediglich der Ursprungsknoten und Zielknoten des Pfads, sowie direkter Vorgängerknoten und direkter Nachfolgerknoten. Aktualisiert wird die Forwarding-Tabelle durch Informationsaustausch mit anderen Knoten, insbesondere falls sich die DDT ändert, wird ebenfalls die Forwarding-Tabelle aktualisiert.

Quelle	Vorgänger	Nachfolger	Ziel	Zeitstempel
		Knoten1	Knoten7	17
		Knoten1	Knoten1	12
		Knoten5	Knoten5	13
			Knoten8	0
		Knoten1	Knoten3	1
Knoten4	Knoten5	Knoten1	Knoten2	28

Tabelle 5.2: Beispiel für Einträge der Forwarding-Tabelle

In der Forwarding-Tabelle werden in einem Eintrag Informationen über Quelle (Ursprungsquelle einer gesendeten Nachricht), Vorgänger (Nachbarknoten, über den eine Nachricht einen Knoten erreicht), Nachfolger (Knoten, an den eine Nachricht weitergesendet wird), Ziel (Knoten, für den eine Nachricht bestimmt ist), sowie einen Zeitstempel gespeichert. Quellen-, Vorgänger- und Nachfolger-Eintrag sind optional. Ist für einen Eintrag keine Quelle und Vorgänger vorhanden, bedeutet dies, dass dieser Eintrag für Nachrichten, die auf dem Knoten erzeugt wurden, verwendet wird. Falls Quellen- und Vorgänger-Eintrag eines Eintrags in der Forwarding-Tabelle, vorhanden sind, wird der Eintrag zur Weiterleitung von Nachrichten genutzt, die auf einen anderen Knoten erstellt wurden.
Aufgabe der Forwarding-Tabelle ist es Informationen zu speichern, um den Knoten zu bestimmen, an den eine Nachricht weitergeleitet werden soll. Um diese Informationen zu erhalten muss ein Ziel angegeben werden, dementsprechend ist der Zieleintrag notwendig. Der Nachfolger-Eintrag ist jedoch optional. Ein nicht gesetzter Nachfolger-Eintrag bedeutet, dass der Ziel-Knoten ein physischer Nachbar ist, der noch nicht der DDT beigetreten ist, siehe Abschnitt 5.5.3. Da eine physische Verbindung vorhanden ist, kann dennoch eine Nachricht weitergeleitet werden.
Der Zeitstempel-Eintrag ist notwendig, da dieser zur Entfernung ungenutzter Einträge genutzt wird. Wird ein Eintrag zur Weiterleitung genutzt, wird der Zeitstempel aktualisiert. Für jeden Eintrag wird periodisch überprüft, ob ein Eintrag benutzt wurde oder nicht. Wurde ein Eintrag in einem vorgegebenen Zeitraum nicht genutzt wird dieser entfernt. Ungenutzte Einträge können bei der Initialisierung des Systems entstehen. Bei der Initialisierung kann sich durch Aktualisierung der Knotenkoordinaten oder durch das bekannt werden neuer virtueller Nachbarn das Forwarding ändern und bestehende Einträge des eigenen Knotens oder anderen Knoten im System überflüssig machen, siehe Abschnitt 5.5.3.

5.5.3 Verwaltung der Forwarding-Tabelle

Neben der Berechnung der DT, siehe Abschnitt 5.4.3, ist zur Konstruktion einer DDT die Erstellung von Forwarding-Informationen in der Forwarding-Tabelle notwendig. Um ein korrektes Forwarding der Nachrichten durch das Forwarding-Protokoll zu gewährleisten, siehe Abschnitt 5.5, muss jeder Knoten des Systems der DDT beitreten. Informationen der Forwarding-Tabelle sind nicht statisch, sondern werden zur Laufzeit des Systems geändert,

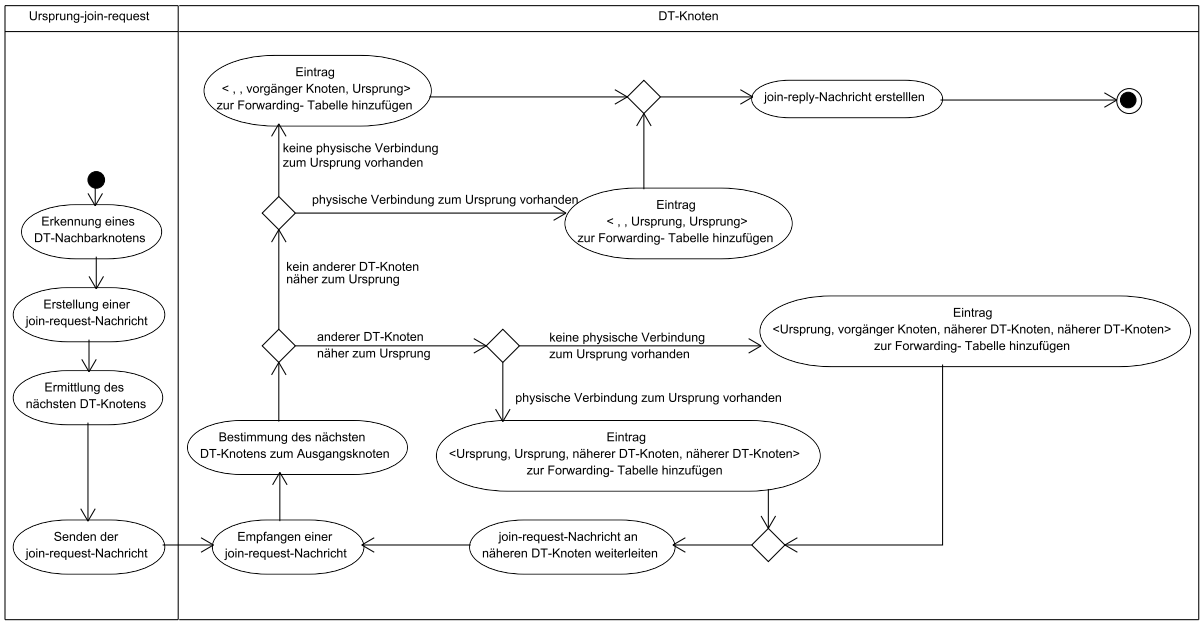
insbesondere falls sich die DDT ändert, ändert sich ebenfalls die Forwarding-Tabelle. In Abschnitt 5.5.2 wurden bereits die enthaltenen Informationen der Forwarding-Tabelle beschrieben. Werden Einträge in der Forwarding-Tabelle erstellt oder aktualisiert, wird der Zeitstempel-Eintrag immer auf 0 gesetzt, dementsprechend wäre bei jedem Eintrag im folgenden Abschnitt der gleiche Zeitstempel vorhanden, aus diesem Grund wird dieser Eintrag zur Vereinfachung im folgenden Abschnitt weggelassen. Die Struktur der Einträge ist also ⟨ *Quelle, Vorgänger, Nachfolger, Ziel*⟩.

Initialisierung der Forwarding-Tabelle

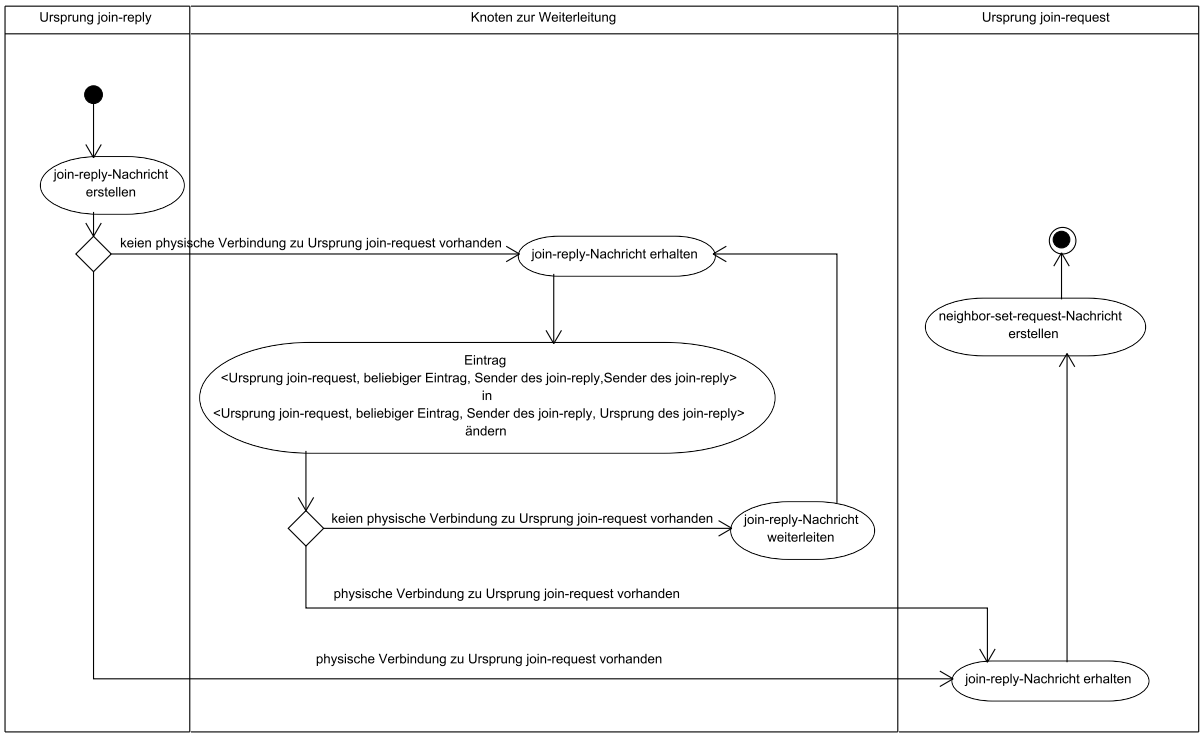
Wird die Forwarding-Tabelle initialisiert, werden für jeden physischen Nachbarn abhängig davon, ob ein Nachbarknoten bereits Teil der DDT ist, unterschiedliche Einträge zu der Forwarding-Tabelle hinzugefügt. Ist ein Nachbarknoten Teil der DDT wird der Eintrag ⟨ , , *Nachbarknoten, Nachbarknoten*⟩ zur Forwarding-Tabelle hinzugefügt, andernfalls ⟨ , , , *Nachbarknoten*⟩.

Aktualisierung der Forwarding-Tabelle

Zum Systemstart wird ein zufällig ausgewählter Knoten festgelegt, der per Definition ein Knoten der DDT ist. Alle anderen Knoten aktualisieren ihre Forwarding-Tabelle sobald ein Nachbarknoten die Information sendet, dass er Teil der DDT ist. Diese Wahl könnte ebenfalls durch ein beliebiges Wahlverfahren anhand eines einfachen Kriteriums, z.B. größte ID, getroffen werden [QL11].
Sobald ein Knoten, der noch nicht Teil der DDT ist, einen Nachbarknoten, der Teil der DDT besitzt, schickt dieser Knoten eine join-request-Nachricht an diesen Nachbarknoten. Sind mehrere Nachbarknoten vorhanden, die bereits Teil der DDT sind, wird der Nachbarknoten mit der geringsten Distanz ausgewählt. Mit dem Senden der join-request-Nachricht signalisiert der Knoten, dass dieser der DDT beitreten möchte, wobei die join-request-Nachricht zur dezentralen Bestimmung des nächsten Knoten der DDT verwendet wird. Um den nächsten Knoten der DDT zu bestimmen werden der join-request-Nachricht als Zielkoordinaten, die virtuellen Koordinaten des Ursprungs der join-request-Nachricht angehängt. Die Nachricht wird innerhalb der DDT durch Minimierung der Distanz vom Ziel zum aktuellen Knoten weitergeleitet. Da der Ursprung der join-request-Nachricht noch kein Knoten der DDT ist, wird die Nachricht nicht direkt an diesen Knoten weitergeleitet. Erhält ein Knoten eine join-request-Nachricht bestimmt der Knoten den Ursprung der Nachricht, von welchem Knoten er die Nachricht erhalten hat, und anhand der Zielkoordinaten an welchen Knoten er die Nachricht ggf. weiterleiten muss. Weitergeleitet werden muss die join-request-Nachricht falls ein anderer Knoten der DDT näher zum Ziel der Nachricht ist. Wird eine join-request-Nachricht an einen anderen Knoten weitergeleitet wird der Eintrag ⟨ *Ursprung der Nachricht, Sender der Nachricht, Knoten zur Weiterleitung, Knoten zur Weiterleitung*⟩ zur Forwarding-Tabelle hinzugefügt. Ist der Ursprung einer Nachricht ein physischer Nachbar wird zur besseren Effizienz die direkte Verbindung also ⟨ *Ursprung der Nachricht, Ursprung der Nachricht, Knoten zur Weiterleitung, Knoten zur Weiterleitung*⟩ zur Forwarding-Tabelle hinzugefügt. Hierbei ist zu beachten, dass beim Forwarding der join-request-Nachricht lediglich Einträge in der Forwarding-Tabelle zwischengespeichert werden. Eine endgültige Zuordnung des Ziel-Eintrags, der zwischengespeicherten Einträge erfolgt erst bei Erhalt



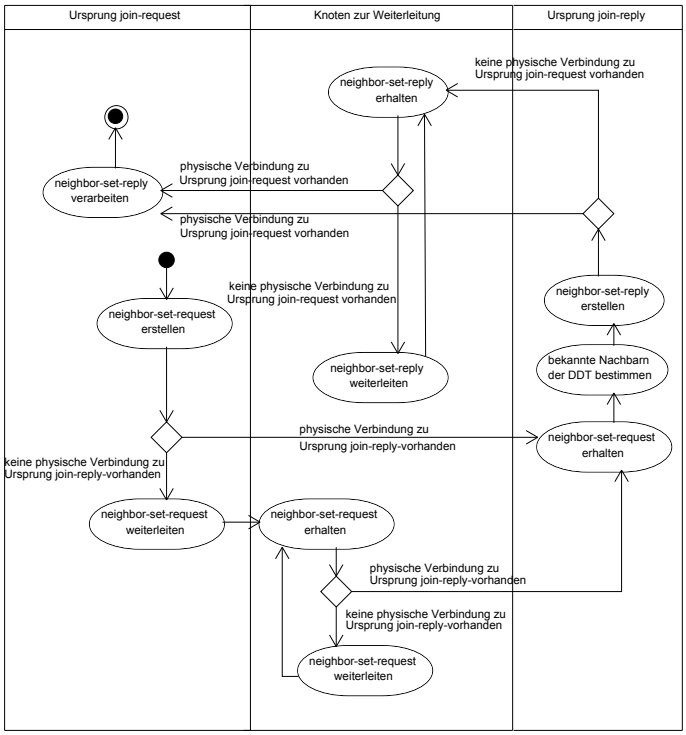
(a) Weiterleitung einer join-request-Nachricht



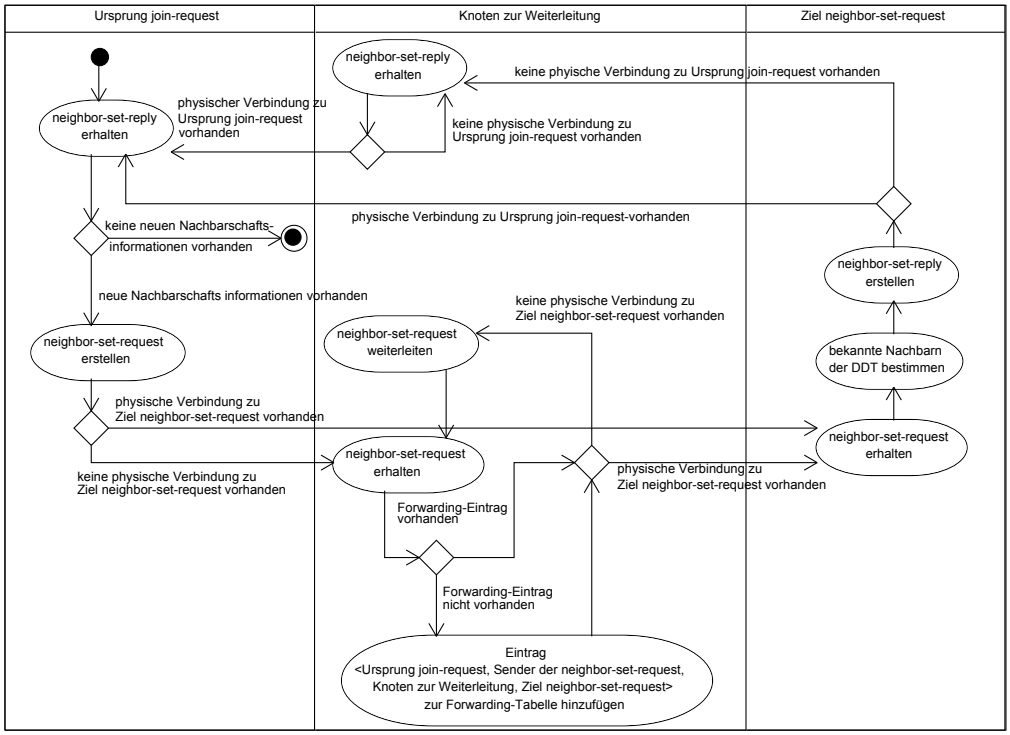
(b) Weiterleitung einer join-reply-Nachricht

Abbildung 5.7: Weiterleitung der join-request-Nachrichten

einer join-reply-Nachricht.
Kann eine Nachricht nicht mehr weitergeleitet, da der nächste Nachbarknoten in der DDT gefunden wurde, wird entsprechend der Eintrag ⟨ , , *Sender der Nachricht, Ursprung der Nachricht*⟩ eingetragen und eine join-reply-Nachricht wird an den Nachrichtenursprung gesendet. Die join-reply-Nachricht wird dazu verwendet, die virtuellen Koordinaten bzw. die ID des nächsten Knoten dem Sender der join-request-Nachricht zu übermitteln. Die Informationen werden von der *Routing-Komponente*, siehe Abschnitt 5.4, bereitgestellt. Hierzu wird der Ursprungsknoten der join-request-Nachricht der lokalen DT des Zielknotens der join-request-Nachricht hinzugefügt, siehe Abschnitt 5.4.3. Zur Weiterleitung der join-reply-Nachricht werden Informationen genutzt, die bei der Weiterleitung der join-request-Nachricht erzeugt wurden. Ein Knoten erhält dadurch nur dann eine join-reply-Nachricht, falls dieser auch die zugehörige join-request-Nachricht weitergeleitet hat. Erhält ein Knoten eine join-reply-Nachrichten werden die Einträge ⟨ *Ursprung der Nachricht, Sender der Nachricht, Knoten zur Weiterleitung, Knoten zur Weiterleitung* ⟩, die in der Forwarding-Tabelle bei der Weiterleitung der join-request-Nachricht erstellt wurden, aktualisiert. Hierbei wird die Zielangabe (Knoten zur Weiterleitung) durch den Ursprung der join-reply-Nachrichten ersetzt ⟨ *Ursprung der Nachricht, beliebiger Eintrag, Knoten zur Weiterleitung, Ursprung der join-reply-Nachrichten*⟩. Erhält der Ursprung der join-request-Nachricht die join-reply-Nachricht wird der Eintrag ⟨ , , *Sender der join-reply-Nachricht , Ursprung der join-reply-Nachricht*⟩ zur Forwarding-Tabelle hinzugefügt, sofern der Ursprung der join-reply-Nachricht kein physischer Nachbar des Knotens ist. Ist der Ursprung der join-reply-Nachricht ein physischer Nachbar, muss kein neuer Eintrag zur Forwarding-Tabelle hinzugefügt werden, da dieser bereits bei der Initialisierung der Forwarding-Tabelle erstellt wurde, ⟨ , , *Nachbarknoten, Nachbarknoten*⟩.
Nach Erhalt der join-reply-Nachricht wird eine neighbor-set-request-Nachricht an den Ursprung der join-reply-Nachricht geschickt. Eine neighbor-set-request-Nachricht signalisiert, dass der Sender der Nachricht, Informationen über Nachbarknoten erhalten möchte, die dem Zielknoten bekannt sind. Erhält ein Knoten eine neighbor-set-request-Nachricht, werden folgende Einträge der Forwarding-Tabelle hinzugefügt, falls der Knoten nicht der Zielknoten ist und der Eintrag noch nicht vorhanden ist, ⟨ *Ursprung der Nachricht, Sender der Nachricht, Knoten zur Weiterleitung, Ziel der Nachricht*⟩. Hierbei kann erneut, falls eine physische Verbindungen vorhanden ist, der Sender der Nachricht, direkt durch den Ursprung der Nachricht ersetzt werden, ⟨ *Ursprung der Nachricht, Ursprung der Nachricht, Knoten zur Weiterleitung, Ziel der Nachricht*⟩. Erreicht eine neighbor-set-request-Nachricht ihr Ziel wird der Eintrag ⟨ , , *Sender der Nachricht, Ursprung der Nachricht*⟩ in der Forwarding-Tabelle des Zielknoten hinzugefügt, falls der Eintrag noch nicht vorhanden ist, und die entsprechende neighbor-set-reply-Nachricht wird versendet. Vor dem Sender der neighbor-set-reply-Nachricht bestimmt der Knoten anhand der vorhandenen Informationen, die Nachbarknoten der DDT des Senders der neighbor-set-request-Nachricht. Die bestimmten Nachbarn werden der neighbor-set-reply-Nachricht angehängt.
Bei der Weiterleitung der neighbor-set-reply-Nachricht werden bestehende Informationen genutzt, um die Nachricht zum Zielknoten weiterzuleiten, hierbei werden die Forwarding-Tabellen der Knoten jedoch nicht geändert. Erhält der Ursprung der neighbor-set-request-Nachricht die neighbor-set-reply-Nachricht, überprüft dieser, ob neue Nachbarschaftsinformationen vorhanden sind. Zu neuen Nachbarn werden ebenfalls neighbor-set-request-Nachrichten geschickt, um weitere Informationen der DDT zu erhalten. Solange über



(a) Weiterleitung der neighbor-set-request-Nachricht zum Ursprung der join-reply-Nachricht



(b) Weiterleitung einer neighbor-set-request-Nachricht

Abbildung 5.8: Weiterleitung der neighbor-set-request-Nachrichten

neighbor-set-reply-Nachricht neue Nachbarschafts-Verbindungen erhalten werden, wird die Forwarding-Tabelle geändert. Ein ausführliches Beispiel für die Aktualisierung der Forwarding-Tabelle ist in Anhang B dargestellt.
Da sich die Positionen der Knoten bei Veränderung der Latenzen ebenfalls ändert, müssen das Forwarding der Knoten ebenfalls angepasst werden dazu werden periodisch neighbor-set-request-Nachricht an die Nachbarknoten der DDT gesendet.

Kapitel 6

Evaluation

Im folgenden Kapitel wird das implementierte Verfahren evaluiert. Um eine vollständige Evaluation des implementierten Systems zu erreichen müssten verschiedene Netzwerktopologien (Stern-, Ring-, Baum- und vollvermaschte Netzwerktopologie) untersucht werden. Für jede Topologie sollten verschiedene Aspekte untersucht werden (Skalierbarkeit bzgl. Anfragen und Initialisierung/ Abweichung der durchgeführten Platzierung der Operatoren von der optimalen Platzierung der Operatoren). Neben der Untersuchung allgemeiner Aspekte, wie Skalierbarkeit, müssten ebenfalls die verwendeten Verfahren bei den speziellen Netzwerktopologien evaluiert werden. Sowohl bei der Aktualisierung des virtuellen Kostenraums (siehe Abschnitt 5.3.2) als auch im Verfahren zur Platzierung der Operatoren (siehe Abschnitt 5.2.1) sind Konstanten vorhanden, die die Ausführung der Verfahren beeinflussen. Im Moment werden Werte für die Konstanten verwendet, die in den entsprechenden Arbeiten angegeben wurden, jedoch könnten ebenfalls unterschiedliche Werte ggf. Vorteile bei unterschiedlichen Topologien haben. Da eine vollständige Evaluation des Systems den Umfang der Arbeit sprengen würde, wird sich in den folgenden Untersuchungen auf einige Aspekte beschränkt.

6.1 Funktionale Korrektheit

Vor der Evaluation des Systems wurde die Korrektheit der einzelnen Komponenten unabhängig voneinander getestet. Hierfür wurden einzelne Bestandteile der Komponenten ebenfalls unabhängig voneinander untersucht. Zur Überprüfung der Korrektheit der Konstruktion einer DT, siehe Algorithmus 4, wurden z.B. zuerst die einzelnen Flip-Operationen, siehe Algorithmus 5, und im Anschluss die gesamte Konstruktion überprüft. Zum Testen der Korrektheit wurden für die getesteten Komponenten verschiedene Testfälle konstruiert, die in drei Testklassen Falsch-, Extrem- und Normalwerte eingeteilt wurden. Für die Konstruktion der DT wurden z.B. zweidimensionale Punkte und Punkte, die im unendlichen liegen (Falschwerte), Positionen mit sehr großer Distanz (Extremwerte), und Position mit kleiner Distanz (Normalwerte) getestet. Da das System in Java implementiert wurde, wurden die Tests mit Hilfe von JUnit [jun12] durchgeführt. Alle Tests wurden erfolgreich durchgeführt. Bei den durchgeführten JUnit-Tests wurde ein Code-Coverage von mindestens 95% bei den entsprechenden Komponenten erreicht.

6.2 Zielstellung der Evaluation

Ziel der Evaluation ist die Skalierbarkeit des implementierten Systems zu überprüfen, dazu müssen drei Aspekte untersucht werden. Als erster Punkt wird überprüft, ob die Anzahl der Anfragen, die das System bearbeiten kann mit der Anzahl der Knoten des Systems steigt. Als zweiter Punkt wird bei der Initialisierung untersucht, wie viele Nachrichten zur Initialisierung des Systems gesendet werden, und wie lange das System zur Initialisierung benötigt. Der dritte Aspekt, der überprüft wird, ist die Berechnung der Delaunay-Triangulation (DT) als Teil der Initialisierung. Bei der Berechnung der DT muss die Zeit zur Konstruktion der DT untersucht werden. Bei den Untersuchungen der Skalierbarkeit wird sich auf ein vollvermaschtes Netzwerk beschränkt, da sonst der Umfang einer Bachelorarbeit gesprengt würde. Andere Netzwerktopologien (z.B. Stern-, Ring- und Baum-Topologie) sollten jedoch ebenfalls in zukünftigen Arbeiten untersucht werden, um ggf. Auswirkungen der verschiedenen Topologien auf das implementierte System zu überprüfen. Der Vorteil in einem vollvermaschten Netzwerk ist, dass sofern die Last der Knoten gleichverteilt ist kein Knoten zum Flaschenhals des Systems wird, d.h. müssen Informationen zwischen Knoten ausgetauscht werden, können diese direkt zu dem Zielknoten gesendet werden, und müssen nicht indirekt über einen anderen Knoten gesendet werden.

6.3 Evaluationssetup

Zur Überprüfung der Skalierbarkeit des System wurde ein Anfragengenerator implementiert. Dieser erzeugt Anfragen, die die Struktur der Anfrage, die in Abbildung 3.2(a) gezeigt wird, um einen weiteren Operator (AggregatorPipe) erweitert. Jede Anfrage verwendet zwei Quellen, die periodisch Events erzeugen. Die Events besitzen einen numerischen Wert, einen Zeitstempel an dem das Event erzeugt wurde, und eine Zeitraum in dem ein Event gültig ist. Der numerische Wert wird durch einen Generator für Zufallszahlen bestimmt. Die erzeugten Events werden von Selektionen (SelectionPipe, siehe Tabelle 2.1) bearbeitet, die einen festen Grenzwert besitzen. Unterschreitet der nummerische Wert eines Events den festgelegten Grenzwert der Selektion, wird das Event verworfen. Der Grenzwert der Selektion wird bei der Erstellung durch einen Zufallswert festgelegt. Events, die nicht verworfen werden, werden im Join-Operator (TemporalJoin, siehe Tabelle 2.1) verarbeitet. Im Join-Operator werden Events für einen festgelegten Zeitraum abhängig vom Gültigkeitszeitraum zwischengespeichert. Die Zwischenspeicherung der Events der unterschiedlichen Event-Ströme erfolgt hierbei unabhängig voneinander. Erreicht ein Event eines Datenstroms den Join-Operator wird dieser mit den zwischengespeicherten Events des anderen Datenstroms verbunden. Zur Verbindung unterschiedlicher Events wird eine maximale Abweichung der Gültigkeitszeiträume definiert. Unterschreitet ein Paar von Events die festgelegte Abweichung bedeutet dies, dass die Events in einem zeitlichen Zusammenhang stehen, deshalb werden diese Events miteinander kombiniert, indem die Zeiträume und Zeitstempel der Events aggregiert werden, und der nummerische Wert der Events addiert wird. Überschreitet ein Paar von Events die festgelegte Abweichung bedeutet dies, dass die Events keinen zeitlichen Bezug zueinander haben, dementsprechend werden diese Events nicht miteinander kombiniert. Zusätzlich zu den in Abbildung 3.2(a) dargestellten Operatoren wird nach dem Join-Operator ein Aggregator-Operator

(AggregatorPipe, siehe Tabelle 2.1) ausgeführt, der die Anzahl der erhaltenen Events zählt. Erhält der Aggregator-Operator eine Nachricht aktualisiert der Aggregator-Operator seinen internen Zähler und erzeugt eine neue Nachricht mit dem aktualisierten Zählerwert. Der Aggregator-Operator leitet die erzeugte Nachricht an die Senke der Anfrage weiter. Die Überprüfung der Skalierbarkeit der DT und der Initialisierung des Systems wurde lokal auf einem Rechner mit 4-Kern CPU (2,94 GHz) und 16 GB Arbeitsspeicher, da so die gemessenen Werte exakt ausgelesen werden können, und keine Synchronisation der einzelnen Knoten notwendig ist. Die Skalierbarkeit bzgl. der Anfragen wurde jedoch auf bis zu sechs Rechnern mit 2-Kern CPU (2,9 GHz) und 4 GB Arbeitsspeicher durchgeführt, um die Untersuchung der Skalierbarkeit des Systems in einem realen Netzwerk durchzuführen.

6.4 Skalierbarkeit bzgl. Anfragen

Wird ein Knoten zu einem bestehenden Netzwerk hinzugefügt, sollte die Anzahl an Anfragen, die das System verarbeiten kann, durch die Ausnutzung der Ressourcen des hinzugefügten Knotens, steigen. Ob das entwickelte System skaliert, wird im folgenden Abschnitt untersucht.

6.4.1 Lokale Ausführung

Die Anzahl an Anfragen, die das System bearbeiten muss, werden periodisch erhöht. Dabei wird die in Abschnitt 6.3 beschriebene Anfragenstruktur verwendet. Um die Skalierbarkeit des Systems zu überprüfen werden in den Senken der Anfragen der Zeitstempel der Events verwendet, um zu berechnen wie lange die Bearbeitung des Events dauerte. Skaliert ein System, ändert sich die Latenz einer Anfrage (Zeit, die für die Bearbeitung der Events notwendig ist) beim Hinzufügen neuer Anfragen kaum.

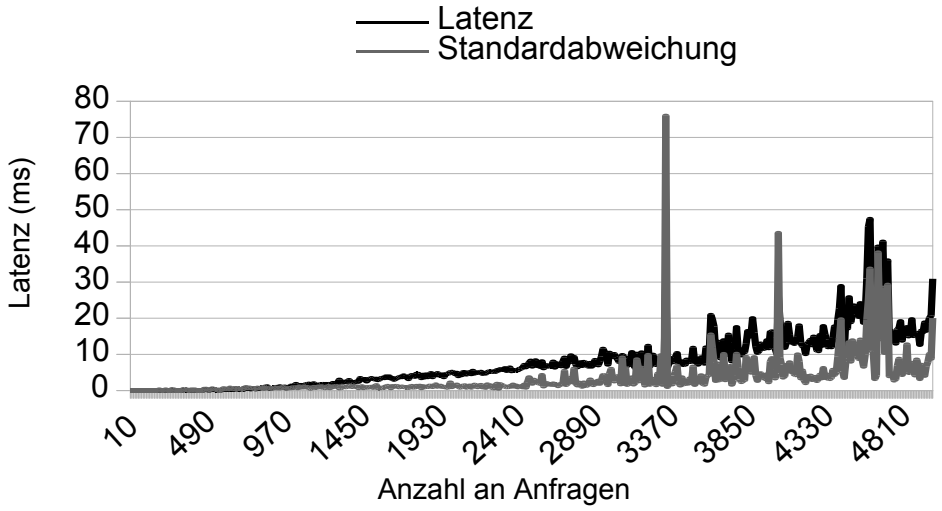


Abbildung 6.1: Latenzen der Anfragen bei lokaler Ausführung

Die Latenzen bei der lokalen Ausführung der Anfragen wird in Abbildung 6.1 dargestellt. Die Latenz steigt nur leicht mit der Zahl der Anfragen. Selbst bei 5000 Anfragen steigt

die durchschnittliche Ausführungszeit (Latenz) auf nur circa 30 Millisekunden. Die Verarbeitung der Events, die durch die Datenquellen erzeugt wurden, ist synchronisiert, d.h. dass die einzelnen Anfragen, die bereits vom System bearbeitet werden, nacheinander durchgeführt werden. Aus diesem Grund können Peaks entstehen, falls in einem gemessenen Abstand verhältnismäßige viele Events erzeugt wurden, die den nummerischen Grenzwert der Selektion überschreiten, da hierbei die Operatoren der Anfragen durch die Synchronisation länger warten müssen. Je höher das Verhältnis zwischen Events, die von der Selektion weitergeleitet werden, und Events, die von der Selektion verworfen werden, ist desto höher ist der Peak. Da die nummerischen Werte der Events gleichverteilte Zufallszahlen in einem festen Intervall sind, kommen diese Peaks nur in unregelmäßigen Abständen vor. In realen Anwendungen kommen solche Peaks häufig vor. Zum Beispiel können in kritische Situationen mehr Messwerte (mit höherer Datenrate) gemessen werden. Bei der Verkehrsanalyse können z.B. während des Berufsverkehrs mehr Daten erfasst werden, da mehr Fahrzeuge unterwegs sind. Um solche Peaks in realen Anwendungen abzufangen, werden oft mehr Ressourcen (z.B. Rechnerknoten) als im normalen Betrieb der Anwendungen verwendet um eine gleichbleibende Performanz zu gewährleisten. Diese Performanz kann jedoch nur erreicht werden, wenn alle Datenquellen ebenfalls auf dem selben Rechner vorhanden sind. Müssen Daten bei einer verteilten Ausführung einer Anfrage über ein Netzwerk übertragen werden, steigt die Ausführungszeit durch die Übertragungszeit im Netzwerk.

6.4.2 Verteilte Ausführung

Falls Datenquellen auf unterschiedlichen Knoten verteilt sind, stellt das Netzwerk eine begrenzende Ressource dar. Werden zu viele Daten über das Netzwerk gesendet, wird das Netzwerk überlastet, und die Daten benötigen mehr Zeit um übertragen zu werden, dadurch erhöht sich die gemessene Latenz in den Senken einer Anfrage. Abbildung 6.2 zeigt ein Beispiel für ein überlastetes Netzwerk.

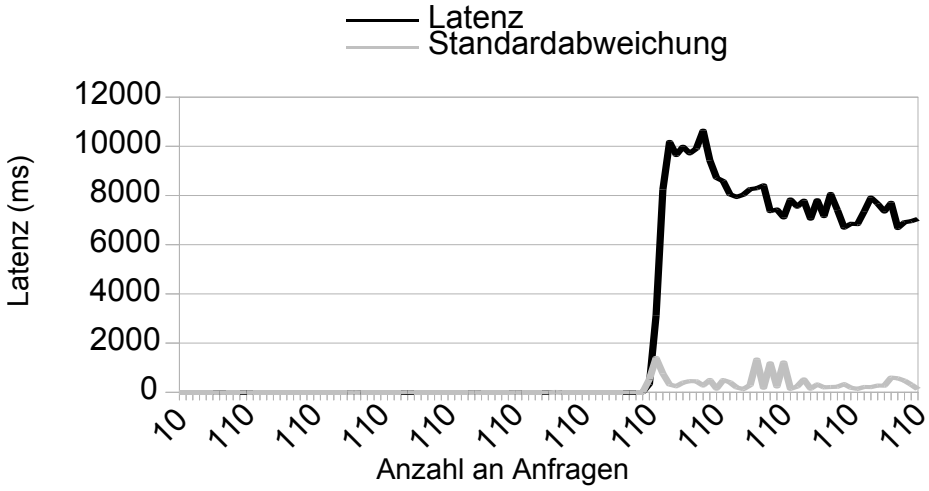


Abbildung 6.2: Überlastung des Netzwerks bei drei Knoten

Bei der Überprüfung besitzt jeder Knoten eine eigene Datenquelle und startet bis zur Überlastung des Systems periodisch Anfragen, die vom bereits beschriebenen Anfragengenerator (siehe Abschnitt 6.3) erzeugt werden. Jede Anfrage verwendet die Datenquelle

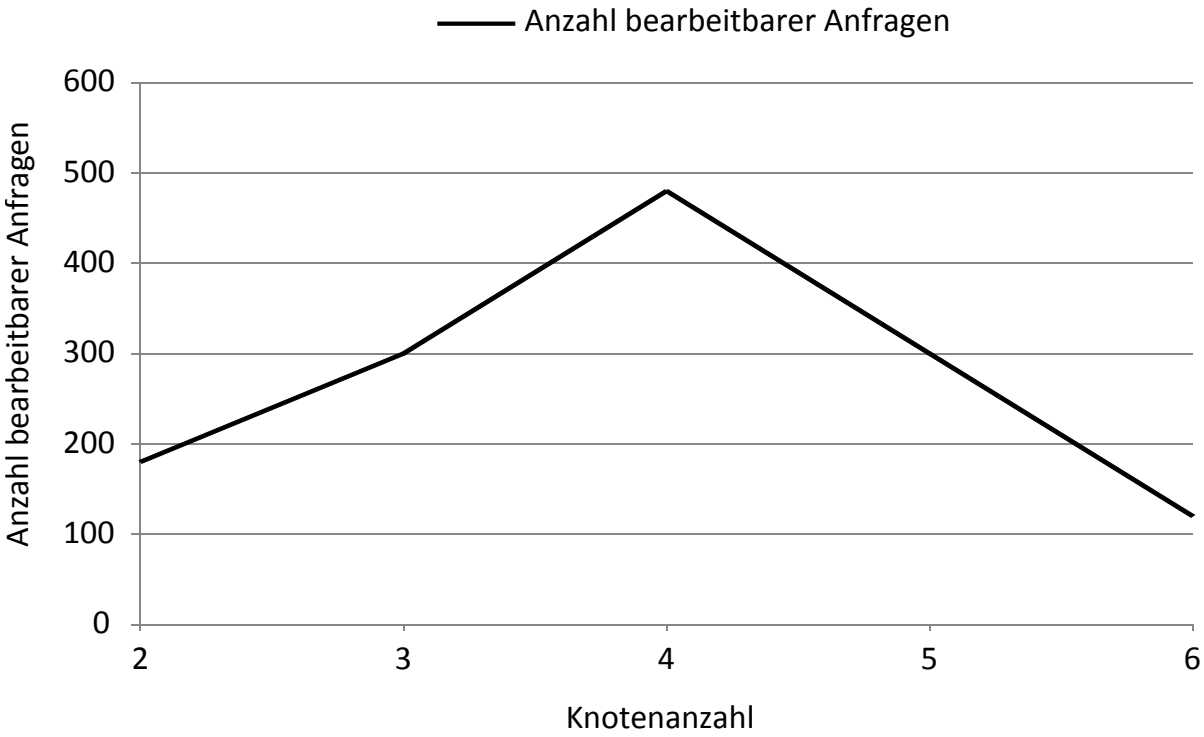


Abbildung 6.3: Skalierbarkeit bzgl. der Anfragen

des Knotens, der die Anfrage gestartet hat, und eine weitere Datenquelle eines beliebigen anderen Knotens des Systems. Da eine Anfrage Datenquellen unterschiedlicher Knoten nutzt, müssen zur Bearbeitung der Anfrage Daten über das Netzwerk übertragen werden, damit ist eine verteilte Ausführung der Anfrage garantiert. Zur Evaluierung der Skalierbarkeit wird die Anzahl der Anfragen gemessen, die das System bearbeitet, bis eine Überlastung eines Knotens des Systems eintritt. Die Ergebnisse werden in Abbildung 6.3 dargestellt. Die Anzahl der Anfragen kann bis zu einer Knotenzahl von vier gesteigert werden, danach fällt die Anzahl der Anfragen, die vom System bearbeitet werden können, ab. In der derzeitigen Form skaliert das System also nicht. In den Untersuchungen wurde ein vollvermaschtes Netzwerk verwendet. Jeder Knoten hatte zu jedem Punkt die gleiche Latenz, dementsprechend müssten ab vier Knoten im virtuellen Kostenraum die gleiche Anzahl an Punkte äquidistant in einem dreidimensional Raum angeordnet werden. Bei bis zu vier Knoten tritt bei den Abschätzungen der Positionen der Knoten im virtuellen Kostenraum keine Ungenauigkeit ein, dementsprechend kann die Anzahl der Anfragen gesteigert werden. Da nur vier Punkte äquidistant zueinander in einem dreidimensionalen Raum (reguläres Tetraeder) angeordnet werden können, wird die Abschätzung der Positionen im virtuellen Kostenraum ungenau. Da die Platzierung der Operatoren mit Hilfe der Positionen der Knoten im virtuellen Kostenraum bestimmt wird, hat die Ungenauigkeit im virtuellen Kostenraum zur Folge, dass ab fünf Knoten ein Knoten überlastet wird, da durch die Ungenauigkeit der Positionen, ein Knoten mehr Operatoren ausführen muss. Durch das implementierte Verfahren zum Lastenausgleich kann zwar die Anzahl an Operatoren der Knoten angeglichen werden, da jedoch die Operatoren auf beliebige Nachbarn verteilt werden, führt dies ebenfalls zu einer höheren Netzwerklast. Obwohl das System nicht bzgl. der Anfragen skaliert, kann das System dennoch eine initiale Platzierung der Operatoren durchführen. Um eine Skalierbarkeit zu gewährleisten

gibt es zwei Möglichkeiten. Eine Möglichkeit ist ein intelligenteres Verfahren zum Lastenausgleich zu implementieren, das eine genaue Abschätzung der Netzwerklast bestimmen kann und Aufgrund dieser Abschätzung eine neue Möglichkeit sucht, die Operatoren zu platzieren. Eine weitere Möglichkeit ist sowohl den virtuellen Kostenraum, als auch die Delaunay-Triangulation in einer höheren Dimension zu berechnen, da dies jedoch vermutlich die Skalierbarkeit nur begrenzt erhöht, dafür aber die Berechnungszeit vor allem bei der DT erhöhen würde, ist ein intelligenteres Verfahren zum Lastenausgleich vorzuziehen.

6.5 Initialisierung

Nach dem Systemstart müssen verschiedene Komponenten initialisiert werden, dazu senden die verschiedenen Knoten des Systems Nachrichten zum Informationsaustausch. Hierbei werden keine eigenen Nachrichten für die einzelnen Informationen z.B. aktuelle Positionen des Knotens gesendet. Die Informationen werden den Nachrichten die gesendet werden müssen, z.B. um die aktuelle Latenz zu bestimmen, siehe Abschnitt 5.3.3, angehängt.
Zur Evaluierung der Initialisierung müssen zwei Faktoren überprüft werden. Zum einen muss gemessen werden, wie viele Nachrichten zur Initialisierung gesendet werden, und wie viel Zeit die Initialisierung benötigt. Die Knoten bilden bei den Messungen ein vollvermaschtes Netzwerk, d.h. jeder Knoten hat zu jedem anderen Knoten des Knotens eine direkte physische Verbindung. Da das System ausgehend von einem Knoten initialisiert wird, siehe Abschnitt 5.3.1, wird zur Messung der Initialisierungszeit die Zeit zwischen dem Start dieses Knoten und dem Zeitpunkt an dem der Fehlerwert der Knotenposition (siehe Abschnitt 5.3.2) einen festgelegten Grenzwert unterschreitet gemessen. Neben der Unterschreitung des festgelegten Grenzwertes des Fehlerwertes müssen dem Knoten, auf dem die Initialisierungszeit gemessen wird, ebenfalls alle Quellen des Systems bekannt sein, die zu Beginn zufällig auf die Knoten des Systems verteilt wurden. Sind beide Bedingungen erfüllt, kann eine Platzierung von Anfragen durchgeführt werden. Ist eine der beiden Bedingungen nicht erfüllt, ist die Initialisierung des Systems noch nicht abgeschlossen, dadurch könnten sich Informationen ändern, die zur Platzierung der Operatoren genutzt werden.
Abbildung 6.4 zeigt die Anzahl der gesendeten Events zur Initialisierung. Es wird ein nahezu linearer Anstieg der gesendeter Events mit steigender Knotenanzahl erreicht. Abbildung 6.5 zeigt die benötigte Zeit zur Initialisierung. Die Zeit zur Initialisierung der Knoten sinkt mit steigender Anzahl der Systemknoten. Der Grund hierfür ist, dass die Informationen über aktualisierte Knotenposition schneller verbreitet werden, da mehr Nachrichten im Netzwerk verbreitet werden, siehe Abbildung 6.4.
Die Initialisierungszeit insbesondere bei einer geringen Anzahl von Knoten kann weiter reduziert werden, indem Knoten die Informationen über eine geänderte Position direkt nach einer Aktualisierung der Knotenposition, siehe Abschnitt 5.3.2, übermitteln.

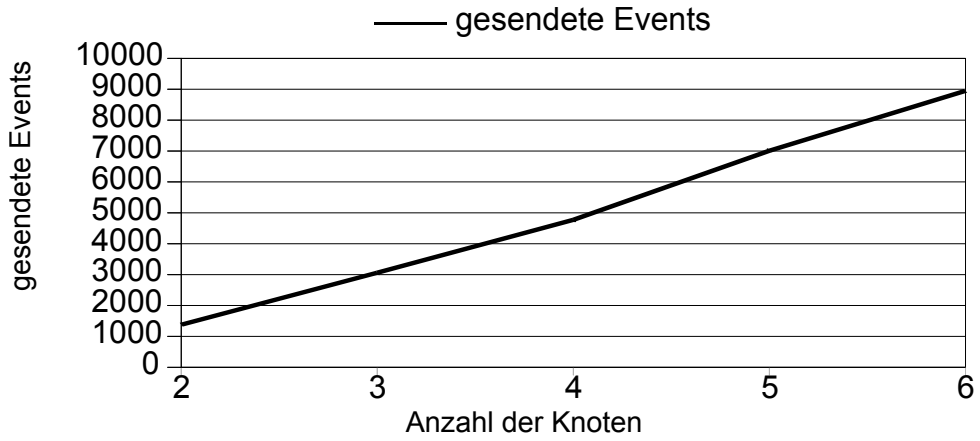


Abbildung 6.4: Gesendete Events zur Initialisierung

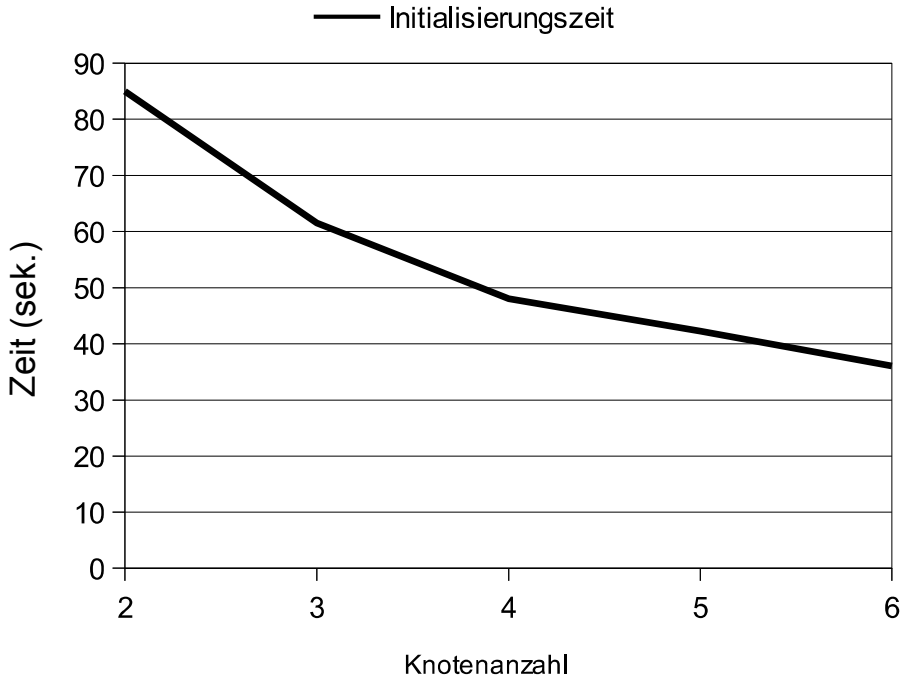


Abbildung 6.5: Initialisierungszeit

6.6 Skalierbarkeit der Delaunay-Triangulation

Das Verfahren zur Berechnung der Triangulation, siehe Algorithmus 4, hat im schlechtesten Fall eine Komplexität von $O(n^3)$, jedoch wird daraufhin gewiesen, dass meistens eine Komplexität kleiner $O(n^2)$ erreicht wird [Led07].
Zur Evaluierung der eigenen Implementation wurden die Zeit zur Konstruktion der DT überprüft. Die Evaluierung der DT erfolgt iterativ. Bei jeder Iteration wird die Anzahl an Knoten bis zu einer maximalen Knotenanzahl erhöht. Da bei der maximalen Knotenanzahl von sechs Knoten, die in den bisherigen Überprüfungen verwendet wurden, keine eindeutige Trendlinie der Berechnungszeit erkennbar war wurde die maximale Knotenanzahl bei der Evaluation der DT auf 500 Knoten festgelegt. Bei jeder Iteration wird die Zeit gemessen, die zur Neuberechnung der Knoten notwendig ist. Die Positionen der Knoten werden durch einen Generator zufällig erzeugt. Die Iteration bis zu einer Knotenanzahl von 500 Knoten wurde 20 Mal wiederholt, und aus den erhaltenen Werte ein Mittelwert gebildet.

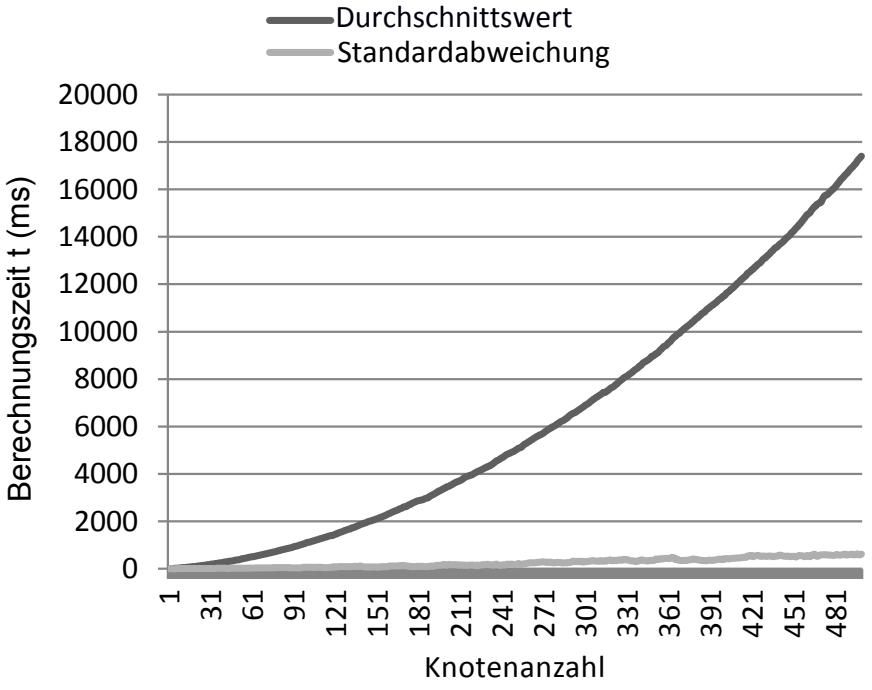


Abbildung 6.6: Konstruktionszeit der Delaunay Triangulation

Wie Abbildung 6.6 zeigt, wird eine Komplexität von $O(n^2)$ erreicht. Die Berechnung der DT liegt damit weit unter dem schlechtesten Fall, kann aber noch optimiert werden. Eine Effizienzsteigerung kann erreicht werden, indem z.B. bei der Konstruktion der DT (siehe Abschnitt 5.4.3) eine effizientere Suche [DPT01] zur Bestimmung des Tetraeders, in dem sich ein Punkt befindet, verwendet wird. Durch eine verbesserte Suche sollte sich die Anzahl an notwendigen Berechnungen reduzieren, und dadurch eine schnellere Konstruktion der DT ermöglicht werden. Eine weitere Steigerung der Effizienz könnte gegebenenfalls durch Verwendung einer anderen Speicherstruktur [ES92] der DT erreicht werden. Bei den Berechnungen wurden jeweils die komplette DT neu berechnet. Während der Ausführung ist die komplette Neuberechnung periodisch notwendig, um ggf. Fehler die sich durch die Änderung der Knotenpositionen ergeben zu beheben, jedoch kann die Zeit

zwischen den Neuberechnungen sehr hoch gesetzt werden. Insbesondere wenn die Latenzen zwischen den Knoten, die zur Bestimmung der Position ermittelt werden, stabil sind kann nach der Initialisierung des Systems auf die komplette Neuberechnung der DT verzichtet werden, da sich bei stabilen Latenzen die Positionen der Knoten nicht verändern. Bei der Initialisierung des Systems muss jedoch nicht die gesamte DT neu berechnet werden, da das Verfahren zur Konstruktion der DT inkrementell ist, siehe Abschnitt 5.4.3. Ein Knoten berücksichtigt bei der Konstruktion der DT nur seine physischen und nächsten virtuellen Nachbarn in der DT, deshalb sollte die Anzahl der zu betrachtenden Nachbarknoten eines Knoten selbst bei einem System mit einer großen Knotenanzahl gering sein. Um die Anzahl der Knoten, die ein Knoten des Systems bei der Konstruktion der DT berücksichtigen muss, weiter zu reduzieren, insbesondere bei einer hohen Konnektivität der Knoten im System, könnten nur die nächsten Nachbarn, unabhängig davon ob eine physische Verbindung der Nachbarn besteht, bei der Konstruktion der DT berücksichtigt werden.

6.7 Zusammenfassung

Wie die Evaluierung zeigt, skaliert das System bzgl. der Initialisierung. Die Anzahl der gesendeten Events steigt nahezu linear. Da bei der Initialisierung des System mit steigender Knotenanzahl ebenfalls die Anzahl der gesendeten Nachricht steigt, reduziert sich dadurch die benötigte Zeit zur Initialisierung des Systems. Die Berechnung der DT, die während der Initialisierung konstruiert wird, liegt mit einer Komplexität von $O(n^2)$ weit unter dem schlechtesten Fall, kann aber noch optimiert werden, z.B. durch eine effizientere Suche [DPT01] zur Bestimmung des Tetraeders in dem sich ein Punkt befindet. Obwohl erfolgreich eine initiale Platzierung für Anfragen erreicht wird, zeigte die Evaluierung, dass das System in der derzeitigen Form nicht bzgl. der Anzahl der Anfragen skaliert. Um eine Skalierbarkeit des Systems bzgl. der Anfragen zu erreichen, muss ein besseres Verfahren zum Lastenausgleich (siehe Abschnitt 6.4) implementiert werden. Das Verfahren zum Lastenausgleich muss erkennen können, falls ein Knoten des Systems überlastet ist. Überlastete Knoten dürfen nicht mehr bei der Platzierung der Operatoren verwendet werden, dementsprechend muss das Verfahren zum Lastenausgleich Operatoren neu im Netzwerk verteilen, falls ein Knoten, der einen Operator einer neuen Anfrage ausführen soll, überlastet ist.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Complex Event Processing (CEP) (siehe Abschnitt 2.1), die zeitnahe Verarbeitung von Ereignissen/ Events einer potentiell unendlichen Datenmenge eines Eventstroms, gewann in den letzten Jahren immer mehr an Bedeutung [CJ09]. Operationen, die für CEP (siehe Tabelle 2.1) genutzt werden, werden kontinuierlich ausgeführt um stetig einströmende Ereignisdaten zu analysieren. Zur Ausführung einer Operation wird ein Operator erstellt. Wird ein neues Ereignis durch einen Eventstrom erzeugt, wird dieses von den bestehenden Operatoren verarbeitet. Abhängig von der internen Logik der Operatoren, werden dabei neue Ereignisse erzeugt. Das Ginseng-Projekt beschäftigt sich unter anderem mit der Entwicklung eines effizienten CEP-Systems zur Verarbeitung von Daten aus einem Sensornetzwerk. Besteht ein CEP-System aus einem Netzwerk von mehreren Knoten, um mehr Ressourcen oder Datenquellen unterschiedlicher Knoten zu nutzen, muss ein Verfahren verwendet werden, dass entscheidet auf welchen Knoten des Systems die Operatoren erstellt und ausgeführt werden sollen. In dieser Arbeit wurde ein Prototyp zur Platzierung von Operatoren in einem System, das aus mehreren Knoten besteht, implementiert. Um eine effiziente Ausführung einer Anfrage (geringe Latenz und Netzwerklast) zu garantieren, muss die verwendete Metrik des Verfahrens zur Platzierung von Operatoren die Latenzen zwischen den Knoten berücksichtigen. Hierzu wurden mehrere verschiedene Ansätze recherchiert, siehe Abschnitt 2.3. Die recherchierten Ansätze wurden mit den Anforderungen des Ginseng-Projektes verglichen (siehe Abschnitt 3.4), dabei wurde das Verfahren von Pietzuch et. al. [PLS+06] für den implementierten Prototyp ausgewählt. Neben der Auswahl eines bestehenden Verfahrens, wurde die Architektur des Systems entwickelt (siehe Abschnitt 4.1). Hierbei konnten mehrere bestehende Komponenten (siehe Abschnitt 4.1.1) zur Ausführung grundlegender Funktionen in den implementierten Prototypen integriert werden. Zur Kommunikation zwischen Knoten des Systems wird eine *Kommunikationskomponente (Netty)* verwendet. Die Operatoren, die für das CEP verwendet werden, werden von einem *erweiterten Pipes-System* (siehe Abschnitt 2.2) bereitgestellt. Anfragen, die vom *erweiterten Pipes-System* bearbeitet werden sollen, können durch eine graphische Oberfläche, dem *QueryAgent*, definiert werden. Neben den bestehenden Komponenten wurden weitere Komponenten implementiert (siehe Abschnitt 4.1.2). Um das *erweiterte Pipes-System* in das implementierte System zu integrieren wurde ein Wrapper, der

PipesConnector, implementiert. Zur Verwaltung der lokalen Operatoren und späteren Integration von weiteren Optimierungsverfahren, z.B. Multi-Query-Optimierung, wurde der *QueryManager* implementiert. Um die Kommunikation zwischen den Knoten des Systems zu ermöglichen wurde neben der Integration der *Kommunikationskomponente (Netty)* der *Connectionmanager*, der die Verbindungen zwischen den Knoten des Systems verwaltet, und eine *Forwarding-Komponente* implementiert, die zum Routing zwischen den Knoten dient. Kernkomponente des implementierten System ist das Distributed Placement (DP) (siehe Kapitel 5). Nach der ersten Konzeptionierung des Systems (siehe Kapitel 4) wurde das ausgewählte Verfahren [PLS+06] (siehe Abschnitt 5.2.1) zur Verwendung in einem verteilten Knotennetzwerk angepasst. Hierfür wurde ein bestehendes Forwarding-Verfahren [LQ11] (siehe Abschnitt 5.5) implementiert, das auf der mathematischen Struktur der Delaunay-Triangulation (DT) basiert. Zur Berechnung der DT (siehe Abschnitt 5.4) wurde ebenfalls ein bestehendes Verfahren [Led07] implementiert. Mit Hilfe des implementierten Forwarding-Verfahrens können Knoten eines verteilten Systems Informationen austauschen, die zum einen zur Konstruktion eines virtuellen Kostenraums (siehe Abschnitt 5.3) genutzt werden. Im virtuellen Kostenraum werden Positionen der Knoten durch bekannte Latenzen zwischen Knoten abgeschätzt. Die abgeschätzten Positionen werden vom Verfahren zur Platzierung von Operatoren genutzt, um eine initiale Platzierung der Operatoren einer Anfrage durchzuführen (siehe Abschnitt 5.2.1). In der Evaluation wurden verschiedene Aspekte des Systems überprüft. Die Evaluation zeigte, dass das implementierte System nicht bzgl. der Anfragenanzahl skaliert (siehe Abschnitt 6.4), jedoch bzgl. der Initialisierung des Systems (siehe Abschnitt 6.5). Mögliche Lösungen um eine Skalierung bzgl. der Anfragenanzahl zu erreichen wurden aufgezeigt.

7.2 Ausblick

Der implementierte Algorithmus kann an vielen Stellen erweitert werden. Bisher wurde in der Arbeit nur die Optimierung anhand der Platzierung der Operatoren betrachtet. Es gibt jedoch weitere Optimierungsverfahren [Sch03] z.B. Multi-Query-Optimierung [XLTZ07], oder Query-Rewriting [SMMP09], die ebenfalls integriert werden könnten. Bei der Multi-Query-Optimierung wird versucht Ergebnisse von Operatoren bestehender Anfragen bei neuen Anfragen wiederzuverwenden. Aufgabe des Query-Rewriting ist Anfragen, die das System bearbeiten sollen, zu einer effizienteren Struktur umzuformen. Eine weitere Möglichkeit ist, die für die Arbeit getroffene Einschränkungen (keine dynamische Optimierung, kein Nachrichtenverlust, statische Netzwerkstruktur, Verbindungsdaten zwischen den Knoten bekannt, siehe Abschnitt 3.3) aufzuheben. Zur weiteren Verbesserung der Effizienz kann eine dynamische Optimierung integriert werden. Bei einer dynamischen Optimierung würden bestehende Operatoren eines Knotens auf andere Knoten verschoben werden. Eine dynamische Optimierung kann durchgeführt werden, falls neue Operatoren auf dem selben Knoten ausgeführt werden müssen, oder, falls sich Informationen ändern, die zur Platzierung der Operatoren verwendet wurden, z.B. Änderung der Selektivität eines Operators, Änderung der Netzwerkstruktur, o.Ä. Eine dynamische Optimierung des Systems ist ebenfalls sinnvoll falls ein Knoten dem System hinzugefügt wird, um eine effiziente Ausführung des Systems zu garantieren. Werden bestehende Operatoren verschoben, müsste das Routing der Nachrichten entsprechend

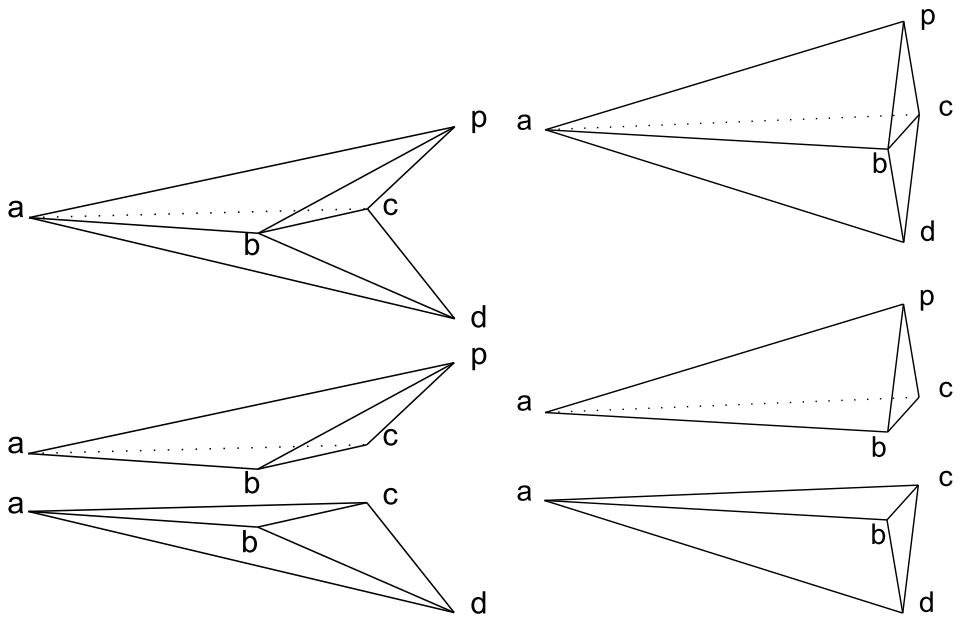
angepasst werden. Des Weiteren müsste für die Verschiebung von zustandsorientierten Operatoren (z.B. Joins) ein Algorithmus implementiert werden, z.B. [ZRH04], der den Zustand des zu verschiebenden Operator ebenfalls überträgt.
Neben der dynamischen Optimierung der Anfragen könnte das System eine dynamische Netzwerkstruktur unterstützen. Bei einer Änderung der Netzwerkstruktur (Ausfall/ Entfernen/ Hinzufügen von Knoten), muss entsprechend ein weiterer Mechanismus implementiert werden, um die Information zu aktualisieren, die zum Routing der Nachrichten im Netzwerk verwendet werden. Können Knoten des Systems ausfallen, muss sicher gestellt werden, dass die Informationen über Operatoren, die auf den Knoten ausgeführt werden, rekonstruiert werden können um einen korrekten Ablauf des Systems sicherzustellen. Fällt ein Knoten des Systems aus, der Datenquellen besitzt, muss die Information verbreitet werden, dass die Datenquellen des Knotens nicht mehr für Anfragen genutzt werden können. Wird ein Knoten entfernt, muss ebenfalls die Information über nicht mehr nutzbare Quellen im Netzwerk verbreitet werden. Es müssen zwar keine Informationen der Operatoren rekonstruiert werden, wenn ein Knoten aus dem Netzwerk entfernt wird, dennoch müssen die Operatoren auf andere Knoten des Systems übertragen werden. Das Hinzufügen eines Knotens zum System kann genutzt werden um einem überlasteten System zusätzliche Ressourcen zur Verfügung zu stellen.
Um eine weitere Automatisierung des Systems zu erreichen, könnte ein Verfahren implementiert werden, um Verbindungsinformationen automatisch zwischen den Knoten des Systems auszutauschen, um den Aufwand bei der Initialisierung des Systems zu senken und eine dynamische Netzwerkstruktur einfacher umzusetzen zu können.
In einem Sensorennetzwerk können gesendete Nachrichten verloren gehen, z.B. durch Störungen bei der Übertragung. Dementsprechend ist es sinnvoll einen Mechanismus zu implementieren, der den Nachrichtenverlust erkennt. Zusätzlich zur Erkennung eines Nachrichtenverlust, muss bei einem Nachrichtenverlust die gesendete Nachricht rekonstruiert und erneut verschickt werden können, z.B. durch eine Zwischenspeicherung in einem Puffer oder einem Log.
Zur Steigerung der Performanz des System könnte eine weitere Überprüfung der Konstruktion der DT durchgeführt werden. So kann untersucht werden wie sich eine andere Speicherstruktur (z.B. [ES92]) der DT auf die Performanz auswirkt und ob dadurch eine Effizienzsteigung erreicht werden kann. Des Weiteren kann eine effizientere Suche [DPT01] zur Bestimmung des Tetraeders, in dem sich ein Punkt befindet, bei der Konstruktion der DT implementiert werden. Durch eine verbesserte Suche sollte sich die Anzahl an notwendigen Berechnungen reduzieren, und dadurch eine schnellere Konstruktion der DT ermöglicht werden. Die Evaluierung beschränkte sich bisher nur auf die Untersuchung der Skalierbarkeit und eine vollvermaschte Netzwerkstruktur, es müssten noch weitere Untersuchungen durchgeführt werden, um weitere Eigenschaften des Systems z.B. Abweichung der Platzierung der Operatoren zur optimalen Position zu bestimmen. Außerdem müsste die Evaluierung noch auf weitere Topologien, wie z.B. Stern-, Ring- und Baumtopologie, erweitert werden.

Anhang A

Flip-Operationen

Im nachfolgenden Abschnitt werden sowohl die Voraussetzungen für die Ausführung, als auch die korrekte Ausführung der einzelnen Flip-Operationen beschrieben. Die einzelnen Flip-Operationen werden von Algorithmus 5 verwendet. Algorithmus 5 ist Bestandteil des Algorithmus 4, der zur inkrementellen Berechnung einer Delaunay-Triangulation (DT) genutzt wird. Hierbei werden die lokalen DT der einzelnen Knoten genutzt, um eine verteilte Delaunay-Triangulation (DDT) zu konstruieren. Diese wird für das Forwarding von Nachrichten das Systems benötigt.

A.1 Notwendige Bedingungen



(a) Flip32-Operation (b) Flip44-Operation

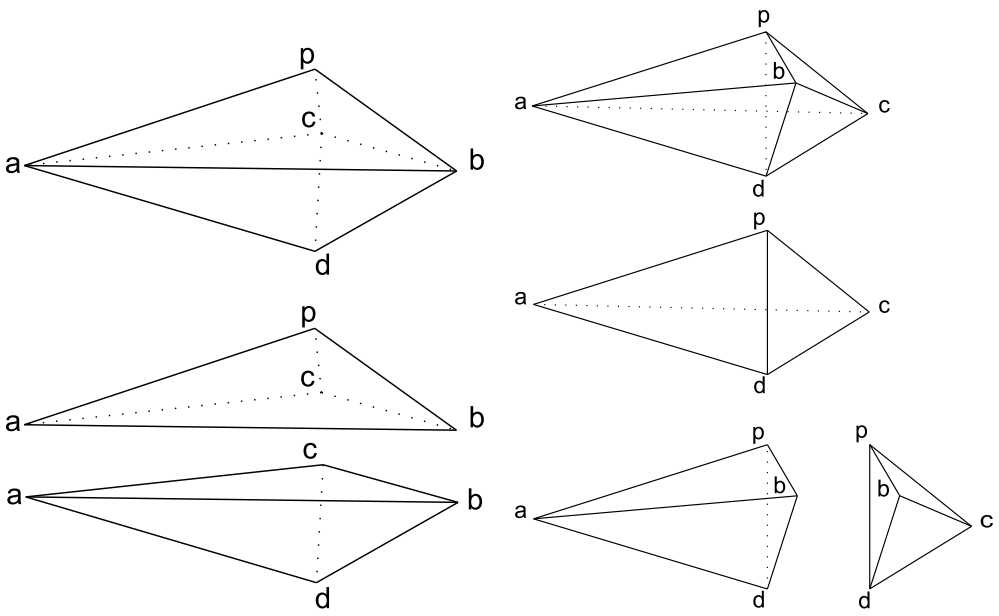
Abbildung A.1: Notwendige Bedingung für Flip-Operationen

Für die Flip-Operationen des Algorithmus 5 werden zwei Tetraeder miteinander verglichen. Für die Flip23-Operation reichen die Information der zwei Tetraeder aus, um zu

entscheiden, ob die Flip-Operation ausgeführt werden muss. Für die zwei anderen Flip-Operationen, Flip32 und Flip44, können hierbei nur notwendige Bedingungen überprüft werden.
Zur Ausführung der Flip32-Operation ist es notwendig, dass sich die zwei Tetraeder eine gemeinsame Fläche teilen, und dass das Polyeder, das durch die Kombination der zwei Tetraeder entsteht, konkav ist. In Abbildung A.1(a) werden zwei Tetraeder dargestellt, die diese notwendigen Bedingungen erfüllen. Die Tetraeder, $\tau_0 = [a, b, c, d]$ und $\tau_1 = [a, b, c, p]$, teilen sich die gemeinsame Fläche $[a, b, c]$, und bilden zusätzlich ein konkaves Polyeder $[a, b, c, d, p]$.
Zur Ausführung der Flip44-Operation ist es notwendig, dass sich die zwei Tetraeder eine gemeinsame Fläche teilen und die Spitzen der zwei Tetraeder und zwei der Eckpunkte, der gemeinsame Fläche, in einer Ebene liegen. In Abbildung A.1(b) werden zwei Tetraeder dargestellt, die diese notwendigen Bedingungen erfüllen. Die Tetraeder, $\tau_0 = [a, b, c, d]$ und $\tau_1 = [a, b, c, p]$, teilen sich die gemeinsame Fläche $[a, b, c]$, zwei Eckpunkt der gemeinsamen Fläche, (b, c) und die zwei Spitzen der Tetraeder (d, p) liegen in einer Ebene.

A.2 Flip23-Operation

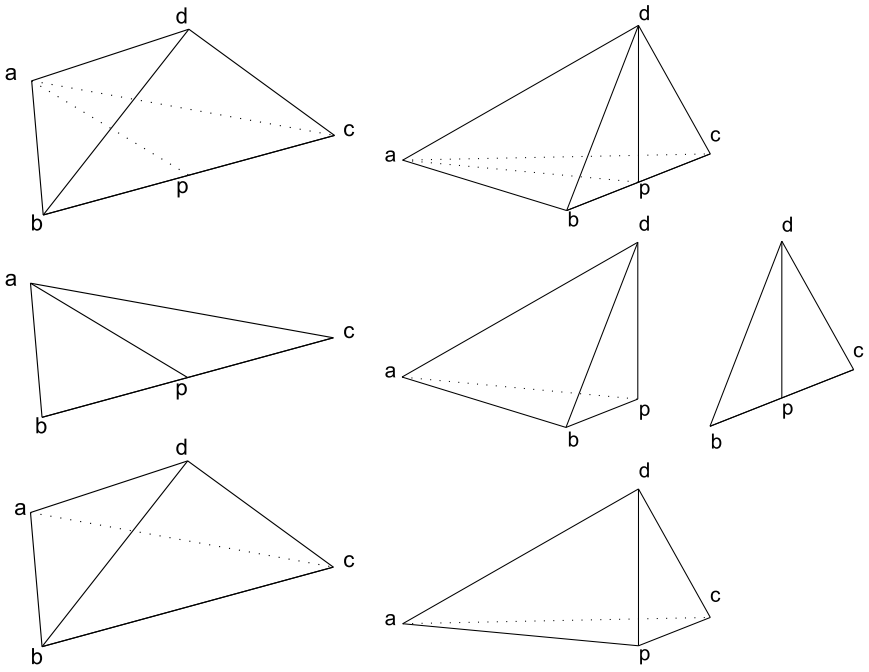
Bei der Flip23-Operation werden aus zwei bestehenden Tetraedern drei neue Tetraeder konstruiert. Voraussetzung für die Flip23-Operation ist, dass sich die zwei bestehenden Tetraeder eine gemeinsame Fläche teilen, und dass das Polyeder, das durch die Kombination der zwei Tetraeder gebildet wird, konvex ist. Zur Konstruktion der drei neuen Tetraeder werden die zwei Spitzen der bestehenden Tetraeder mit jeweils zwei Punkten der gemeinsamen Grundfläche kombiniert. Ein Beispiel hierfür ist in Abbildung A.2 dargestellt. Hierbei werden aus den zwei Tetraeder, $\tau_0 = [a, b, c, d]$ und $\tau_1 = [a, b, c, p]$,



(a) Fall1: Konvexe Kombination zweier Tetraeder

(b) Ergebnis der Flip23-Operation

Abbildung A.2: Beispiel für eine Flip23-Operation



(a) Beispiel von zwei Tetraeder im Fall4

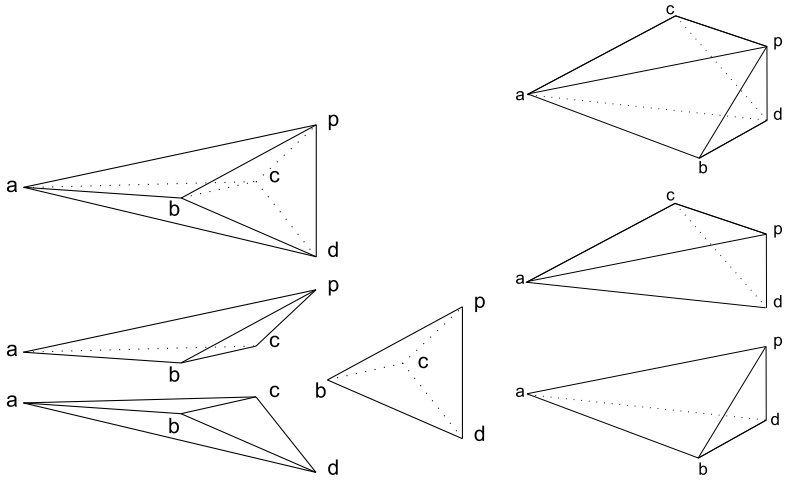
(b) Ergebnis der Flip23-Operation

Abbildung A.3: Spezialfall für die Flip23-Operation

siehe Abbildung A.2(a), drei neue Tetraeder konstruiert. Die Tetraeder τ_0 und τ_1 teilen sich die Fläche $[a, b, c, d]$, und das Polyeder $[a, b, c, d, p]$ ist konvex. Es werden die zwei Spitzen der Tetraeder bestimmt, (d, p), und mit den möglichen Paaren der Eckpunkte kombiniert (a, b), (b, c) und (b, d). Die hierbei entstehenden Tetraeder $\tau_2 = [a, b, d, p]$ $\tau_3 = [a, c, d, p]$ und $\tau_4 = [b, c, d, p]$ sind in Abbildung A.2(b) dargestellt. Ein Spezialfall (siehe Abbildung A.3), der bei der Konstruktion der DT eintreten kann, ist, dass alle Eckpunkte eines Tetraeder in einer Ebene liegen. Da sich dadurch die Ausführung der Flip23-Operation nicht ändert wird dieser nicht weiter betrachtet.

A.3 Flip32-Operation

Bei der Flip32-Operation werden aus drei bestehenden Tetraedern zwei neue Tetraeder konstruiert. Voraussetzung ist hierfür, dass sich alle drei bestehenden Tetraeder eine gemeinsame Kante teilen. Jeweils zwei der drei Tetraeder teilen sich eine gemeinsame Fläche. Die Spitzen jeweils zweier Tetraeder bilden mit der gemeinsame Kante das dritte Tetraeder. Zur Erstellung der zwei neuen Tetraeder wird zuerst die gemeinsame Kante der drei bestehenden Tetraeder bestimmt. Des Weiteren wird die gemeinsame Fläche zweier bestehender Tetraeder bestimmt. Es wird der Eckpunkt der Fläche bestimmt, der nicht in der gemeinsamen Kante enthalten ist, sowie die zwei Spitzen der zwei Tetraeder ermittelt, für die die gemeinsame Fläche bestimmt wurde. Zur Bildung der zwei neuen Tetraeder werden die zwei Spitzen mit dem bestimmten Eckpunkt und jeweils einem der Punkte, der gemeinsamen Kante, kombiniert. Ein Beispiel hierfür ist in Abbildung A.4 dargestellt. Hierbei werden aus den bestehenden Tetraedern $\tau_0 = [a, b, c, d]$, $\tau_1 = [a, b, c, p]$



(a) Kombination von Tetraedern an denen eine Flip32-Operation ausgeführt wird

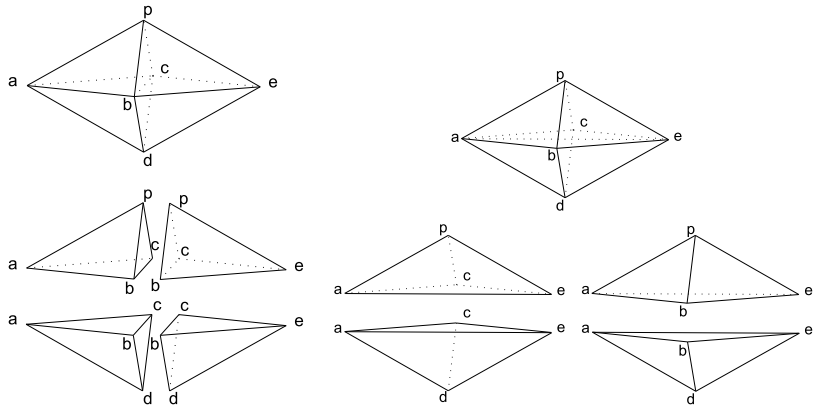
(b) Ergebnis der Flip32-Operation

Abbildung A.4: Beispiel für eine Flip32-Operation

und $\tau_2 = [b, c, d, p]$ zwei neue Tetraeder konstruiert. Die gemeinsame Kante und die jeweiligen Spitzen zweier Tetraeder bilden das jeweilige dritte Tetraeder. Zur Bildung der zwei neuen Tetraeder wird von zwei Tetraedern τ_0 und τ_1 die gemeinsame Fläche bestimmt $[a, b, c]$ bzw. der Eckpunkt a der gemeinsamen Fläche $[a, b, c]$, der nicht Teil der gemeinsamen Kante $\vec{bc}$ ist. Des Weiteren werden die zwei Spitzen d und p der Tetraeder τ_0 und τ_1 bestimmt. Die Fläche $[a, d, p]$ wird nun mit jeweils einem Punkt der gemeinsamen Kante kombiniert. Die hierbei entstehenden Tetraeder $\tau_3 = [a, b, d, p]$ und $\tau_4 = [a, b, c, p]$ werden in Abbildung A.4(b) dargestellt.

A.4 Flip44-Operation

Bei der Flip44-Operation werden aus vier bestehenden Tetraedern vier neue Tetraeder konstruiert. Voraussetzung hierfür ist, dass jedes Tetraeder mit zwei anderen Tetraedern jeweils eine gemeinsame Fläche besitzt. Des Weiteren besitzen alle vier Tetraeder eine gemeinsame Kante, und die Spitzen zweier Tetraeder, die sich eine gemeinsame Fläche teilen, liegen in einer Ebene mit der gemeinsamen Kante der vier Tetraeder. Zur Erstellung der vier neuen Tetraeder wird die gemeinsame Kante der vier Tetraeder bestimmt, sowie die Spitzen für jeweils zwei Tetraeder, die sich eine gemeinsame Fläche teilen. Die ermittelten Spitzen werden so gepaart, dass die Paare der Spitzen und die gemeinsame Kante jeweils in einer Ebene liegt. Eines der Paare der Spitzen wird mit jeweils einem Punkt der gemeinsame Kante und jeweils einem Punkt des anderen Paares der Spitzen kombiniert. Ein Beispiel ist in Abbildung A.5 dargestellt. Die vier Tetraeder, $\tau_0 = [a, b, c, d]$, $\tau_1 = [a, b, c, d]$, $\tau_2 = [b, c, d, e]$ und $\tau_3 = [b, c, e, p]$ (siehe Abbildung A.5(a)), besitzen eine gemeinsame Kante $\vec{bc}$, und die Spitzen der benachbarten Tetraeder (a, e) und (d, p) liegen jeweils in einer Ebene mit der gemeinsamen Kante. Zur Konstruktion der vier neuen Tetraeder wird ein Paar von Spitzen ausgewählt hier (a, e), und mit jeweils einem Punkt der gemeinsamen Kante $\vec{bc}$, und einem Punkt des anderen Spitzen-Paares (d, p), kombiniert. Folglich werden die Punkte (a, e) mit den Punkten (b, d), (b, p), (c, d) und



(a) Ausgangskonfiguration für Flip44-Operation

(b) Ergebnis der Flip44-Operation

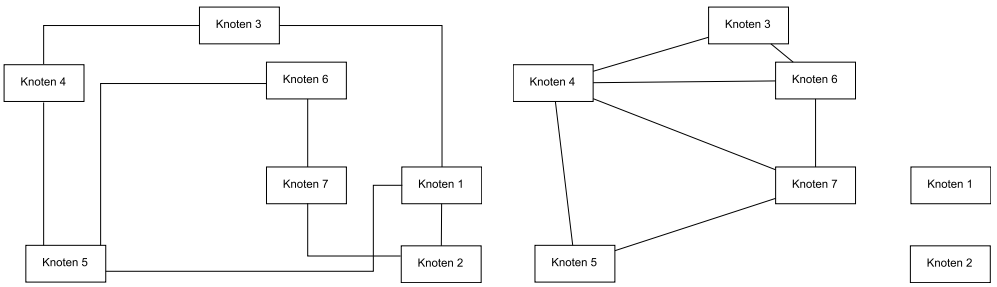
Abbildung A.5: Beispiel einer Flip44-Operation

(c, p) kombiniert. Die hierbei entstehenden Tetraeder $\tau_4 = [a, b, d, e]$, $\tau_5 = [a, b, e, p]$, $\tau_6 = [a, c, e, d]$ und $\tau_7 = [a, c, e, p]$ werden in Abbildung A.5(b) dargestellt.

Anhang B

Beispiel für den Aufbau einer Forwarding-Tabelle

Im Folgenden wird der Aufbau der Forwarding-Tabelle anhand eines Beispiels beschrieben. Hierbei werden die Verbindungsinformationen sowie die Delaunay-Triangulation (DT)-Informationen genutzt, die in Abbildung B.1(a) bzw. in Abbildung B.1(b) dargestellt werden. Im System selbst wird eine verteilte Delaunay-Triangulation (DDT) verwendet. Durch die inkrementelle Konstruktion der DT der Knoten $3-7$ wurden ebenfalls, die entsprechenden Einträge der Forwarding-Tabelle konstruiert, siehe Tabelle B.1. Knoten 1 und 2 sind in diesem Beispiel noch nicht initialisiert, und haben dementsprechend noch keine Einträge in der Forwarding-Tabelle, und wurden deshalb aus Gründen der Übersichtlichkeit nicht betrachtet. Die DT wird im implementierten System in $\mathbb{R}^3$ berechnet, zur besseren Darstellungen wird dieses Beispiel im $\mathbb{R}^2$ dargestellt. Betrachtet wird hierbei die Konstruktion der Forwarding-Tabelle des Knotens 1. Der Nachrichtenaustausch zwischen den einzelnen Knoten ist in Abbildung B.2 dargestellt.



(a) Verbindungen zwischen Knoten (b) Delaunay-Triangulation

Abbildung B.1: Forwarding

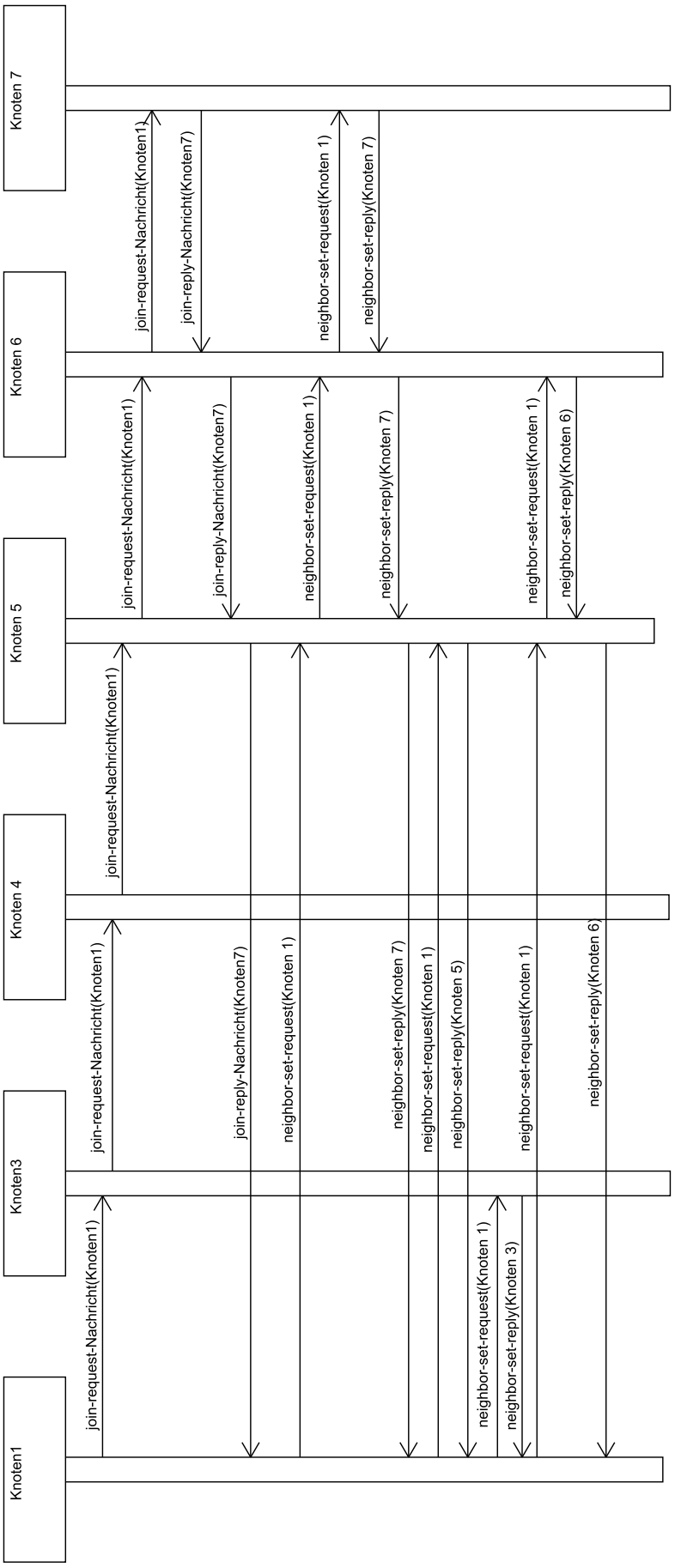


Abbildung B.2: Zeitlicher Ablauf des Nachrichtenaustausch des Beispiels 5.2

Knoten 3	Knoten 4	Knoten 5	Knoten 6	Knoten 7
, , , 1	, , 3, 3	, , 6, 7	, , 5, 5	, , , 2
, , 4, 4	, , 5, 5	4, 4, 6, 7	, , 7, 7	, , 6, 6
, , 4, 6	, , 5, 7	3, 3, 6, 6	, , 5, 3	, , 6, 5
	, , 5, 6	4, 4, 6, 6	4, 5, 7, 7	, , 6, 4
	3, 3, 5, 6	, , 4, 4	, , 5, 4	
		, , 6, 6	5, 5, 7, 7	
		, , , 1		

Tabelle B.1: Forwarding-Tabelle der Knoten, siehe Abbildung B.1(b)

, , 5, 5
, , 3, 3
, , , 2

Tabelle B.2: Forwarding-Tabelle von Knoten 1 nach der Initialisierung

Beim Start des Knotens 1 werden für seine physischen Nachbarknoten 2, 3 und 5 die Einträge in der Forwarding-Tabelle erzeugt. Da die Knoten 3 und 5 bereits Teil der DT sind werden die Einträge $\langle , , 3, 3\rangle$, und $\langle , , 5, 5\rangle$ hinzugefügt. Da der Knoten 2 noch nicht in der DT ist, wird der Eintrag $\langle , , , 2\rangle$ zur Forwarding-Tabelle hinzugefügt, siehe Tabelle B.2.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
, , , 1	, , , 1
, , 4, 4	, , 4, 4
, , 4, 6	, , 4, 6
	1, 1, 4, 4

Tabelle B.3: Aktualisierung der Forwarding-Tabelle von Knoten 3

Nach der Initialisierung des Knotens 1, sendet Knoten 1 an den Knoten 3 eine join-request-Nachricht, da Knoten 3 der nächste Nachbarnoten der DT von Knoten 1 ist, zu dem Knoten 1 eine direkte Verbindung besitzt. Bei Erhalt der join-request-Nachricht von Knoten 1 berechnet Knoten 3 den nächsten Nachbarknoten der DT, Knoten 6, der ihn bekannt ist, diese Information wird der Nachricht angehängt, anschließend leitet Knoten 3 die Nachricht über den Knoten 4 weiter. Hierbei speichert Knoten 3 den Eintrag $\langle 1, 1, 4, 4\rangle$. Die Auswahl des Knotens 4 erfolgt anhand bestehender

Forwarding-Informationen, mit dem Zielknoten 6, siehe Tabelle B.3.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
| | | 3 | 3 |	| | | 3 | 3 |
| | | 5 | 5 |	| | | 5 | 5 |
| | | 5 | 7 |	| | | 5 | 7 |
| | | 5 | 6 |	| | | 5 | 6 |
| 3 | 3 | 5 | 6 |	| 3 | 3 | 5 | 6 |
	| 1 | 3 | 5 | 5 |

Tabelle B.4: Aktualisierung der Forwarding-Tabelle von Knoten 4

Erhält Knoten 4 die join-request-Nachricht wird anhand der von Knoten 3 an die Nachricht angehängten Informationen der Knoten 5 bestimmt, an den die Nachricht weitergeleitet werden muss. Knoten 4 leitet die Nachricht an Knoten 5 weiter und speichert den Eintrag $\langle 1, 3, 5, 5 \rangle$.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
| | | 6 | 7 |	| | | 6 | 7 |
| 4 | 4 | 6 | 7 |	| 4 | 4 | 6 | 7 |
| 3 | 3 | 6 | 6 |	| 3 | 3 | 6 | 6 |
| 4 | 4 | 6 | 6 |	| 4 | 4 | 6 | 6 |
| | | 4 | 4 |	| | | 4 | 4 |
| | | 6 | 6 |	| | | 6 | 6 |
| | | | 1 |	| | | | 1 |
	| 1 | 1 | 6 | 6 |

Tabelle B.5: Aktualisierung der Forwarding-Tabelle von Knoten 5

Erhält der Knoten 5 die join-request-Nachricht leitet dieser die Nachricht an seinen physischen Nachbarknoten 6 weiter. Da Knoten 1, der Ursprung der join-request-Nachricht, ebenfalls ein physischer Nachbarknoten ist wird hierbei der Eintrag $\langle 1, 1, 6, 6 \rangle$ in der Forwarding-Tabelle gespeichert. Knoten 5 nützt hierbei die physische Verbindung zu Knoten 1 um einen effizienteren Weg zu ermöglichen. Alle Nachrichten, die zwischen Knoten 1 und Knoten 6 verschickt werden, werden direkt durch Knoten 5 weitergeleitet. Da die Einträge, die in Knoten 2 und 3 erstellt wurden, nicht mehr genutzt werden, wird der Zeitstempel der Einträge nicht mehr aktualisiert. Bei der periodischen Überprüfung

der Einträge, werden die Einträge nach Ablauf der Zeitstempel gelöscht.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
| | | 5 | 5 |	| | | 5 | 5 |
| | | 7 | 7 |	| | | 7 | 7 |
| | | 5 | 3 |	| | | 5 | 3 |
| 4 | 5 | 7 | 7 |	| 4 | 5 | 7 | 7 |
| | | 5 | 4 |	| | | 5 | 4 |
| 5 | 5 | 7 | 7 |	| 5 | 5 | 7 | 7 |
	| 1 | 5 | 7 | 7 |

Tabelle B.6: Aktualisierung der Forwarding-Tabelle von Knoten 6

Knoten 6 erhält die join-Request-Nachricht von Knoten 1 und berechnet erneut den nächsten DT-Nachbarknoten zu Knoten 1. Der nächste DT-Nachbarknoten der Knoten 6 bekannt ist, ist Knoten 7. Knoten 6 leitet nun die Nachricht an Knoten 7 weiter und speichert dabei, den Eintrag $\langle 1, 5, 7, 7\rangle$ in der Forwarding-Table.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
| | | | 2 |	| | | | 2 |
| | | 6 | 6 |	| | | 6 | 6 |
| | | 6 | 5 |	| | | 6 | 5 |
| | | 6 | 4 |	| | | 6 | 4 |
	| | | 6 | 1 |

Tabelle B.7: Aktualisierung der Forwarding-Tabelle von Knoten 7

Knoten 7 erhält nun die join-request-Nachricht von Knoten 1. Da Knoten 7 der nächste DT-Knoten zu Knoten 1 ist, erstellt Knoten 7 die join-reply-Nachricht. Ziel der join-reply-Nachricht ist Knoten 1, da Knoten 1 der Ursprung der join-request-Nachricht war. Um die join-reply-Nachricht an Knoten 1 schicken zu können, wird der Eintrag $\langle , , 6, 1\rangle$ in der Forwarding-Tabelle gespeichert. Knoten 6 ist der Nachfolger für Knoten 1, da die join-request-Nachricht über diesen Knoten erhalten wurde. Im Anschluss leitet Knoten 7 die erstellte join-reply-Nachricht an Knoten 6 weiter. Knoten 6 leitet die Nachricht entsprechend des erstellten Eintrags $\langle 1, 5, 7, 7\rangle$ der Forwarding-Tabelle an Knoten 5 weiter.

Forwarding-Tabelle vor der Aktualisierung				Forwarding-Tabelle nach der Aktualisierung			
		6	7			6	7
4	4	6	7	4	4	6	7
3	3	6	6	3	3	6	6
4	4	6	6	4	4	6	6
		4	4			4	4
		6	6			6	6
			1				1
1	1	6	6	1	1	6	7

Tabelle B.8: Aktualisierung der Forwarding-Tabelle von Knoten 5

Knoten 5 aktualisiert nun seinen Eintrag der Forwarding-Tabelle $\langle 1, 1, 6, 6\rangle$ mit Informationen der join-replay-Nachricht. Der Ziel-Eintrag wird von Knoten 6 auf Knoten 7 geändert.

Forwarding-Tabelle vor der Aktualisierung				Forwarding-Tabelle nach der Aktualisierung			
		5	5			5	5
		3	3			3	3
			2				2
						5	7

Tabelle B.9: Aktualisierung der Forwarding-Tabelle von Knoten 1

Bei Erhalt der join-Reply-Nachricht speichert Knoten 1 den Eintrag $\langle , , 5, 7\rangle$.
Über die etablierte Verbindung sendet nun Knoten 1 an Knoten 7 eine neighbor-set-request-Nachricht. Knoten 7 antwortet auf die neighbor-set-request-Nachricht mit einer neighbor-set-reply-Nachricht. Knoten 7 fügt Knoten 1 seiner lokalen DT und bestimmt die Nachbarknoten 5 und 6, der DT zu Knoten 1, die Knoten 7 bekannt sind. Diese Information wird der neighbor-set-reply-Nachricht angehängt, die anschließend weitergeleitet wird. Bei Erhalt der neighbor-set-reply-Nachricht, schickt Knoten 1 neue neighbor-set-request-Nachrichten an die neuen Knoten 5 und 6. Da durch die join-request- bzw. join-reply-Nachricht ein Pfad zwischen Knoten 1 und 7 existiert, und durch die Initialisierung der DT ebenfalls ein Pfad zwischen Knoten 7 und 6 besteht, ist garantiert, dass die Nachricht von Knoten 1 zu Knoten 6 weitergeleitet werden kann. Da Knoten 5 ein physischer Nachbarknoten ist, ändert sich das Forwarding nicht. Beim Senden der neighbor-set-request-Nachricht von Knoten 1 an Knoten 6 werden bestehende Informationen genutzt, um die Nachricht weiterzuleiten, d.h. als Zwischenstation (Relay) wird Knoten 7 verwendet. Die Weiterleitung erfolgt deshalb über den Nachfolger Knoten 5 zur Zwischenstation 7.

Forwarding-Tabelle vor der Aktualisierung	Forwarding-Tabelle nach der Aktualisierung
			6	7
4	4	6	7
3	3	6	6
4	4	6	6
		4	4
		6	6
			1
1	1	6	7
		6	7
4	4	6	7
3	3	6	6
4	4	6	6
		4	4
		6	6
			1
1	1	6	7
1	1	6	6

Tabelle B.10: Aktualisierung der Forwarding-Tabelle von Knoten 5

Knoten 5 speichert hierbei den Eintrag $\langle 1, 1, 6, 6\rangle$. Da in den neighbor-set-reply-Nachrichten der Knoten 5 und 6 keinen neue Nachbarknoten für Knoten 1 bestimmt werden können, ist die Aktualisierung des Knotens 1 abgeschlossen.
Knoten 2 wird bei der Aktualisierung der Forwarding-Tabellen nicht berücksichtigt, da der Knoten noch nicht teil der DT ist.

Literaturverzeichnis

[Ac04] Ahmad, Y.; Çetintemel, U.: Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, S. 456–467. VLDB Endowment, 2004.

[AceJ+05] Ahmad, Y.; etintemel, U. c.; Jannotti, J.; Zgolinski, A.; Zdonik, S. B.: Network awareness in internet-scale stream processing. *IEEE Data Eng. Bull.*, Band 28, Nr. 1, S. 63–69, 2005.

[Aur91] Aurenhammer, F.: Voronoi diagrams a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, Band 23, S. 345–405, September 1991.

[BB03] Bonfils, B. J.; Bonnet, P.: Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the 2nd international conference on Information processing in sensor networks*, IPSN'03, S. 47–62. Springer-Verlag, Berlin, Heidelberg, 2003.

[BBD+02] Babcock, B.; Babu, S.; Datar, M.; Motwani, R.; Widom, J.: Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, S. 1–16. ACM, New York, NY, USA, 2002.

[BBS04] Balazinska, M.; Balakrishnan, H.; Stonebraker, M.: Contract-based load management in federated distributed systems. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, S. 15–15. USENIX Association, Berkeley, CA, USA, 2004.

[CcC+02] Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Seidman, G.; Stonebraker, M.; Tatbul, N.; Zdonik, S.: Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, S. 215–226. VLDB Endowment, 2002.

[CH08] Cormode, G.; Hadjieleftheriou, M.: Finding frequent items in data streams. *Proc. VLDB Endow.*, Band 1, Nr. 2, S. 1530–1541, August 2008.

[CHK+03] Cammert, M.; Heinz, C.; Krämer, J.; Markowetz, A.; Seeger, B.: Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technischer Bericht Nr. 32, Department of Mathematics and Computer Science, July 2003.

[CHKS04] Cammert, M.; Heinz, C.; Krämer, J.; Seeger, B.: Anfrageverarbeitung auf datenströmen. In *Datenbank-Spektrum*, S. 5–13. 2004. 4(11).

[CJ09] Chakravarthy, S.; Jiang, Q.: *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 1st. Auflage, 2009.

[DPT01] Devillers, O.; Pion, S.; Teillaud, M.: Walking in a triangulation. In *Proceedings of the seventeenth annual symposium on Computational geometry*, SCG '01, S. 106–114. ACM, New York, NY, USA, 2001.

[EFGK03] Eugster, P. T.; Felber, P.; Guerraoui, R.; Kermarrec, A.-M.: The many faces of publish/subscribe. *ACM Comput. Surv.*, Band 35, Nr. 2, S. 114–131, 2003.

[ES92] Edelsbrunner, H.; Shah, N. R.: Incremental topological flipping works for regular triangulations. In *Proceedings of the eighth annual symposium on Computational geometry*, SCG '92, S. 43–52. ACM, New York, NY, USA, 1992.

[Goo12] Google Inc.: Google Protocol Buffers, 2012. http://code.google.com/apis/protocolbuffers/. Stand: 14.02.2012.

[JC08] Jin, C.; Carbonell, J.: Predicate indexing for incremental multi-query optimization. In *Proceedings of the 17th international conference on Foundations of intelligent systems*, ISMIS'08, S. 339–350. Springer-Verlag, Berlin, Heidelberg, 2008.

[JFF07] Jerzak, Z.; Fach, R.; Fetzer, C.: Fail-aware publish/subscribe. In *Sixth IEEE International Symposium on Network Computing and Applications*, NCA 2007, S. 113–125. IEEE Computer Society, Cambridge, MA, USA, 2007.

[jun12] JUnit, 2012. http://www.junit.org/. Stand: 29.03.2012.

[KJ10a] Klein, A.; Jerzak, Z.: Ginseng for sustainable energy awareness: flexible energy monitoring using wireless sensor nodes. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, S. 109–110. ACM, New York, NY, USA, 2010.

[KJ10b] Klein, A.; Jerzak, Z.: The ginseng middleware for performance control in sensor networks. In *5th. International Conference on Management and Control of Production and Logistics*, MCPL '10, 2010.

[Led07] Ledoux, H.: Computing the 3d voronoi diagram robustly: An easy explanation. In *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering*, S. 117–129. IEEE Computer Society, Washington, DC, USA, 2007.

[LF98] Luckham, D. C.; Frasca, B.: Complex event processing in distributed systems. Technischer Bericht Nr. CSL-TR-98-754, Stanford University, August 1998.

[LL06] Lee, D.-Y.; Lam, S. S.: Protocol design for dynamic delaunay triangulation. Technischer Bericht Nr. TR-06-48, Department of Computer Science, University of Texas at Austin, Dezember 2006.

[LLS08] Lakshmanan, G. T.; Li, Y.; Strom, R.: Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, Band 12, S. 50–60, November 2008.

[LQ11] Lam, S. S.; Qian, C.: Geographic routing in d-dimensional spaces with guaranteed delivery and low stretch. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, S. 257–268. ACM, New York, NY, USA, 2011.

[MC11] Margara, A.; Cugola, G.: Processing flows of information: from data stream to complex event processing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, S. 359–360. ACM, New York, NY, USA, 2011.

[MFP06] Mühl, G.; Fiege, L.; Pietzuch, P.: *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Net12] Netty Project: Netty, 2012. http://netty.io. Stand: 08.02.2012.

[PLS+06] Pietzuch, P.; Ledlie, J.; Shneidman, J.; Roussopoulos, M.; Welsh, M.; Seltzer, M.: Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06. Atlanta, GA, April 2006.

[QL11] Qian, C.; Lam, S. S.: Greedy distance vector routing. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, S. 857–868. IEEE Computer Society, Washington, DC, USA, 2011.

[Sch03] Schäfer, T.: Anfrageoptimierung auf datenströmen. Diplomarbeit, Philipps-Universität Marburg, Marburg, Oktober 2003.

[Sel88] Sellis, T. K.: Multiple-query optimization. *ACM Trans. Database Syst.*, Band 13, S. 23–52, March 1988.

[SMMP09] Schultz-Møller, N. P.; Migliavacca, M.; Pietzuch, P.: Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, S. 4:1–4:12. ACM, New York, NY, USA, 2009.

[SMW05] Srivastava, U.; Munagala, K.; Widom, J.: Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, S. 250–258. ACM, New York, NY, USA, 2005.

[SRAZ08] Schuh, G.; Rozenfeld, H.; Assmus, D.; Zancul, E.: Process oriented framework to support plm implementation. *Comput. Ind.*, Band 59, Nr. 2-3, S. 210–218, März 2008.

[WB10] Wolf, B.; Behrens, I.: On-the-fly adaptation of data stream queries. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, S. 175–179. IEEE Computer Society, Washington, DC, USA, 2010.

[WDR06] Wu, E.; Diao, Y.; Rizvi, S.: High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, S. 407–418. ACM, New York, NY, USA, 2006.

[XLTZ07] Xiang, S.; Lim, H. B.; Tan, K.-L.; Zhou, Y.: Two-tier multiple query optimization for sensor networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, S. 39–. IEEE Computer Society, Washington, DC, USA, 2007.

[ZRH04] Zhu, Y.; Rundensteiner, E. A.; Heineman, G. T.: Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, S. 431–442. ACM, New York, NY, USA, 2004.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 17. April 2012

Andreas Meister