Entwicklung einer automatisierten Messumgebung für das Constraint-basierte Datenbank-Caching

Diplomarbeit im Fachbereich Informatik
Prof. Dr. Theo Härder
Technische Universität Kaiserslautern

von

Dipl.-Inf. (FH)
Joachim Klein

Betreuer:
Prof. Dr. Theo Härder
Dipl.-Inf. Andreas Bühmann

Tag der Anmeldung: 1. Mai 2006
Tag der Abgabe: 31. Oktober 2006

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kaiserslautern, den 30. Oktober 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1 Motivation

Um die Antwortzeit von Webanwendungen zu verkürzen, werden Web-Caches eingesetzt. Sie puffern statische Webseiten oder dynamisch generierte Fragmente von Webanwendungen. Datenbank-Caches sind in ähnlicher Weise einsetzbar, um Anfragen an Datenbanken schneller ausführen zu können. Sie halten Mengen von Datenelementen des zugrunde liegenden Datenbanksystems (Backend) repliziert vor, möglichst nahe an der Anwendung.

Im Gegensatz zu Lösungsansätzen, denen die Wartung recht starr definierter materialisierter Sichten zugrunde liegt, strebt das Constraint-basierte Datenbank-Caching eine adaptive Verwaltung des Cache-Inhalts an: Dieser soll sich an den in der Vergangenheit nachgefragten Daten orientieren [Büh05; HB07].

Durch den Einsatz von Datenbank-Caches und der damit verbundenen Bildung von Replikaten von Datenelementen wird die Synchronisation und die Einhaltung der ACID-Eigenschaften für Transaktionen deutlich erschwert. Aus diesem Grund wird meist auf Caches nur ein lesender Zugriff erlaubt. Änderungsoperationen können nur durch eine synchrone Verarbeitung mit dem Backend-DBS durchgeführt werden, was zu deutlichen Leistungseinbußen führt und die Skalierbarkeit des gesamten Systems aus zentralem Backend und (steigender Anzahl von) Caches begrenzt.

Um die Implementierung von Constraint-basierten Datenbank-Caches zu verbessern und dadurch die Leistung und Skalierbarkeit des Gesamtsystems zu steigern, müssen Leistungsdaten abfragbar und auswertbar gemacht werden.

Standard-Datenbankbenchmarks wie z. B. der TPC-C- oder TPC-W-Benchmark sind hierfür nur bedingt einsetzbar. Eine reine Leistungsmessung erzeugt zwar wichtige Aussagen über die Leistungsfähigkeit des Gesamtsystems, kann aber zum Beispiel die Frage, in welcher Weise die Backend-Datenbank durch den Einsatz des Caches belastet oder entlastet wird, nicht klären. Diese und ähnliche Fragestellungen sind für die Entscheidung für oder gegen den Einsatz bzw. für das Aufspüren von Verbesserungspotentialen eines Constraint-basierten Datenbank-Caches besonders relevant und spielen in dieser Diplomarbeit eine wichtige Rolle.

Darüber hinaus ist es notwendig, eine große Fülle von Tests unter verschiedensten verteilten Konstellationen und Lastbedingungen durchzuführen. Dies erfordert ein leistungsstarkes Test- bzw. Messframework, das möglichst automatisiert beliebige Situationen erfassbar und auswertbar macht.

1.2 Zielsetzung der Arbeit

In der vorliegenden Diplomarbeit soll eine Messumgebung implementiert werden, die es ermöglicht, automatisiert Leistungsparameter eines Constraint-basierten Datenbank-Caches zu überprüfen und zu messen.

Eine Anwendung, die ein Datenbanksystem benutzt, stellt eine verteilte Anwendung mit den beiden Teilen Client und DB-Server dar. Bei Hinzunahme eines Cache-Systems lassen sich Client, DB-Cache und DB-Server unterscheiden. Jede Teilanwendung des Systems kann für eine Messung wichtige Messdaten liefern.

Darum soll ein Framework geschaffen werden, mit dessen Hilfe möglichst allgemeingültig die Eigenschaften einer Messung innerhalb eines verteilten Systems erfasst und abgebildet werden können. Die Vermessung eines Stand-Alone-Systems wird dadurch jedoch nicht ausgeschlossen. Dieses kann als Spezialfall eines verteilten Systems mit nur einer zu vermessenden Teilanwendung betrachtet werden.

Das Framework soll unabhängig von der Art des später zu vermessenden Systems sein. Es soll also keine Rolle spielen, ob z. B. ein verteiltes Datenbanksystem mit Client-Server-Struktur oder eine P2P-Anwendung vermessen wird.

Eine besondere Herausforderung ist hierbei die Definition von allgemeinen Schnittstellen, die es ermöglichen, die stets gleichbleibenden Aufgaben während einer Messung, wie z. B. das Erfassen der Struktur des verteilten Systems, das Erfassen von Leistungsdaten und das Speichern von Leistungsdaten sowie deren spätere Auswertung möglichst gut zu unterstützen.

Darüber hinaus bildet die Aufgabe, die Messung möglichst automatisiert durchzuführen, einen weiteren Schwerpunkt. Dazu gehört, eine zu vermessende Komponente möglichst automatisiert zu installieren, zu konfigurieren, zu starten und zu vermessen sowie nach der Messung für jeden Konfigurierungsschritt die notwendigen „Aufräumarbeiten“ zu leisten (z. B. eine Datenbank starten und nach der Messung wieder stoppen). Ebenso muss das Framework in der Lage sein, unabhängig von Benutzereingriffen, mehrere Messungen hintereinander durchzuführen, damit z. B. eine größere Menge von Messungen (evtl. auch in der Nacht) ohne ständige Beobachtung bearbeitet werden kann.

Um das zu vermessende System mit Aufgaben zu versorgen, soll die Möglichkeit bestehen, Testdaten und dazu passende Anfragelasten zu definieren. Zum Generieren von Testdaten soll evaluiert werden, inwiefern bereits bestehende Datengeneratoren verwendet werden können. Es soll aber auch möglich sein, anhand von Regeln Testdaten für beliebig vorgegebene Schemas anzulegen. Die Anfragelasten werden durch die Implementierung von Transaktionen definiert. Diese sollen während einer Messung durch einen Scheduler, der die Arbeitslast eines Client-Systems simuliert, ausgeführt werden können. So wird es z. B. möglich, die Transaktionstypen und Datenbankschemata des TPC-C- oder TPC-W-Benchmarks zu implementieren, was allerdings nicht Teil dieser Arbeit ist.

Nach der Implementierung der Messumgebung sollen einige relevante Leistungsparameter des Constraint-basierten Datenbank-Cachings identifiziert und beispielhaft gemessen werden. Hierunter fallen vor allem Messwerte wie die Anzahl der Anfragen, die alleine durch den Cache abgearbeitet werden konnten.

1.3 Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen des Constraint-basierten Datenbank-Caching vorgestellt. Der Fokus liegt auf dem Verständnis für den grundlegenden Aufbau dieses Datenbank-Cache-Systems. Die Beleuchtung der Hauptfunktionalitäten, die das Cache-System bereitstellen muss, wird dazu genutzt herauszufinden, welche Messwerte von besonderer Bedeutung bei der Vermessung eines solchen Systems sind.

Kapitel 3 analysiert möglichst detailliert die Aufgaben, die ein Framework zur Vermessung eines verteilten Systems übernehmen kann bzw. muss. Zur besseren Gliederung und verständlichen Darstellung werden zunächst übergeordnete Aufgabengebiete identifiziert, die dann in entsprechende Einzelaufgaben unterteilt werden. Konzepte und Überlegungen, die zur Lösung einer identifizierten Aufgabe entwickelt wurden, werden, eingebettet im dazugehörigen Abschnitt, vorgestellt und diskutiert.

Danach werden in Kapitel 4 wichtige Teile der Umsetzung des Frameworks im Detail gezeigt. Hierbei werden alle Konfigurationsmöglichkeiten und Schnittstellen vorgestellt, die es ermöglichen, verschiedenste Systeme zu vermessen.

Direkt darauf aufbauend wird in Kapitel 5 gezeigt, wie das entwickelte Framework dazu genutzt werden kann, wichtige Messdaten des Constraint-basierten Datenbank-Caching zu erfassen. Die hierzu notwendigen Implementierungen werden nur kurz vorgestellt.

Abschließend wird in der Zusammenfassung eine Bewertung des entwickelten Frameworks vorgenommen. Darüber hinaus werden weiterführende Ideen diskutiert und Möglichkeiten aufgezeigt, das Framework noch leistungsfähiger zu gestalten. Die nun ermittelbaren Messwerte können nicht nur für die Leistungsbestimmung herangezogen werden. Sie stellen eventuell wichtige Entscheidungshilfen und Statistiken dar, die das ACCache-System (Adaptive Constaint-based Cache System) verwenden kann, um noch adaptiver Entscheidungen zu treffen. Erste Ideen hierzu werden kurz diskutiert. Eventuelle Probleme, die bei der Vermessung des Constraint-basierten Cache-Systems entstanden sind, werden nochmals aufgegriffen und mögliche Lösungen und weiterführende Arbeitsschritte werden vorgestellt.

1.4 Basisszenario als allgemeiner Bezugsrahmen

Als Beispielszenario, welches in der Folge stets als Ausrichtungshilfe dienen soll, um die Zusammenhänge und Entscheidungen besser verdeutlichen zu können, betrachten wir typische Anwendungen eines weltweit agierenden Handelsunternehmens.

Das Handelsunternehmen mit Zweigstellen in mehreren Ländern vertreibt und entwickelt Produkte, die dann unter Ausnutzung verschiedener Handelswege vertrieben werden. Um die Produkte weltweit besser verkaufen zu können, entwickeln lokale Dienstleister spezielle, auf die Gegebenheiten im jeweiligen Verkaufsland zugeschnittene Anwendungen. Dabei müssen sie stets auf einen weit entfernten Serververbund

zugreifen, der die Stammdaten des Unternehmens, darunter auch die Produktinformationen und Spezifikationen, verfügbar hält. Um die durch die große Entfernung zum Stammdatenserver entstehende lange Antwortzeit ihrer Anwendungen zu verringern, entschließen sich die Entwickler, anwendungsnahe Daten auf ihren Anwendungsservern zu halten. Aufgrund der großen Entfernung und der oft nur recht kleinen benötigten Datenmenge kommt eine statische Replikation (z. B. aller Produktinformationen) nicht in Frage, auch wenn längst nicht alle Stammdaten repliziert werden müssten. Die Firma benötigt ein Cache-System, das möglichst adaptiv, je nach aufkommender Anfragelast, benötigte Teile der Stammdaten vorhält. Dabei sollen nur solche Daten zwischengespeichert werden, die mit hoher Wahrscheinlichkeit in naher Zukunft nochmals benötigt werden.

Die nachfolgende Abbildung (s. Abbildung 1.1) zeigt die Netzstruktur des fiktiven Unternehmens.

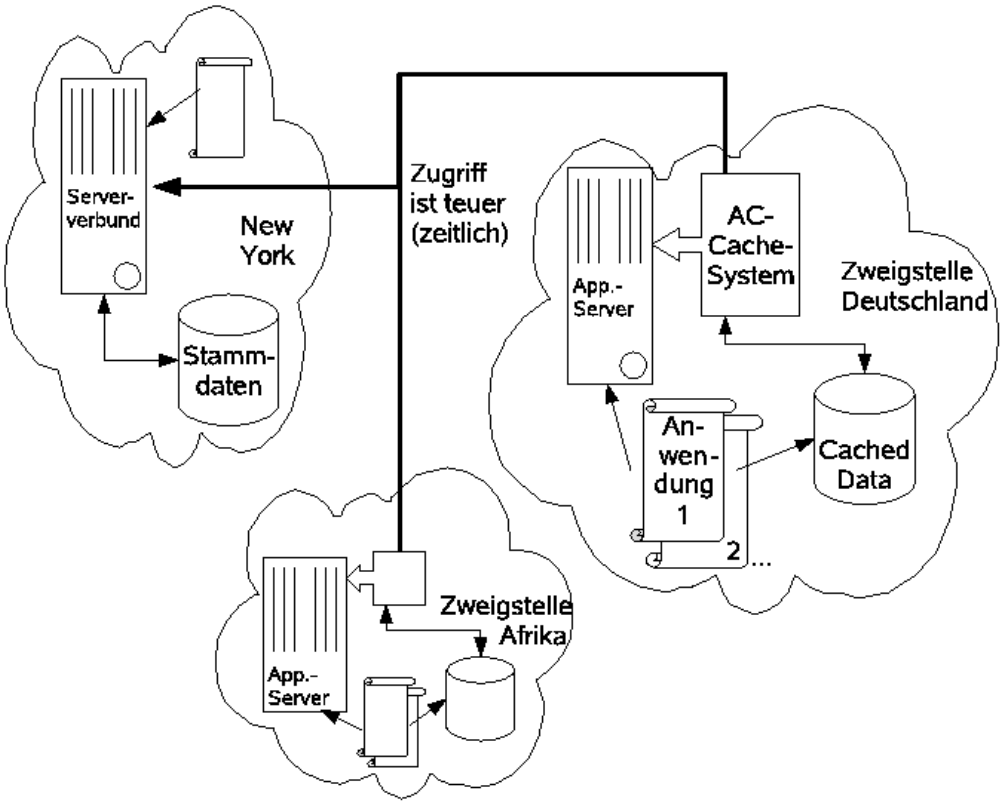

Abbildung 1.1: Übersicht über das Referenzszenario.

2. Grundlagen des Constraint-basierten Datenbank-Caching

Ähnlich wie bei Web-Caches, mit deren Hilfe es möglich wurde, die Antwortzeit von Browsern entscheidend zu verbessern, wird auch beim Datenbank-Caching versucht, die Datenelemente aus Datenbanken möglichst nahe der Anwendung zu halten, um deren Reaktionszeit zu verbessern. Durch das Zwischenspeichern der Datenbankelemente wird darüber hinaus versucht, den zentralen Datenspeicher (Backend) zu entlasten.

Jedoch ist die Erstellung eines leistungsstarken Datenbank-Cache-Systems, das zum einen möglichst adaptiv nur die Daten vorhält, die in naher Zukunft benötigt werden, und zum anderen die ACID-Eigenschaften aller über den Cache zugegriffenen Daten garantiert – bei gleichzeitiger Skalierbarkeit des Gesamtsystems – eine schwierige Aufgabe.

Ein besonderes Problem stellt hierbei die Kontrolle und Verarbeitung von Änderungen und somit die Synchronisation dar. In einem System mit zwei Cache-Instanzen (C_a und C_b) und einer Backend-Datenbank (B) darf z. B. eine Transaktion T_1, die über C_a abläuft, nur dann ein Datenelement d schreiben, wenn auf dieses von keiner anderen laufenden Transaktion (z. B. T_2 auf C_b) zugegriffen wird.

Für die Handhabung dieser oder ähnlicher Probleme muss das Gesamtsystem eine Lösung anbieten, was eine zusätzliche Belastung darstellt. Somit stehen der Entlastung des Backends durch den Cache auch zusätzliche Belastungen gegenüber. Dies ist vor allem dann der Fall, wenn auf dem Cache-System Datenbankwerte vorgehalten werden, auf die häufig schreibend und nur selten lesend zugegriffen wird, da ein schreibender Zugriff, in einem einfachen Fall, solange aufgehalten werden kann, bis sichergestellt ist, dass auf das entsprechende Datenelement über keinen Cache mehr zugegriffen wird. Der Einsatz eines Cache-System ist also nur dann sinnvoll, wenn die Entlastung des Gesamtsystems größer ist als der zusätzlich entstehende Aufwand.

Um möglichst viele lesende Anfragen schon auf dem Cache beantworten zu können, wird daher beim Constraint-basierten Datenbank-Caching versucht, für ein Anfrageprädikat P im Cache die sog. *Prädikatvollständigkeit* herzustellen. Hierdurch können alle Teilanfragen, die das Prädikat voll oder in Teilen abdecken, bereits auf dem Cache beantwortet werden [HB07].

Definition 2.1 (Prädikatvollständigkeit) *Eine Menge von Tabellen heißt* ***prädikatvollständig*** *in Bezug auf ein Prädikat p, wenn sie alle Datensätze beinhaltet, die zur Auswertung von p benötigt werden.*

Ein Cache beinhaltet eine Menge von Cache-Tabellen und eine Menge von Cache-Constraints. Die Cache-Tabellen repräsentieren die entsprechenden Tabellen aus dem Backend und werden dazu verwendet, jeweils Teilmengen der Tupel aus den Backend-Tabellen auf dem Cache vorzuhalten. Die Constraints beschreiben die Inhalte sowie die Abhängigkeiten zwischen den Inhalten der einzelnen Cache-Tabellen und definieren so die gültigen Zustände des Caches. Die Eigenschaften dieser Zustände können dann herangezogen werden, um zu entscheiden, ob eine gegebene Anfrage aus dem Cache teilweise oder komplett beantwortet werden kann [Büh06].

Die Definitionen von Cache-Tabellen und Cache-Constraints werden in sogenannten *Cache Groups* zusammengefasst. Abbildung 2.1 zeigt ein Beispiel für eine Cache-Group-Definition, wie sie für die Abteilung Afrika unseres Handelskonzerns möglich wäre.

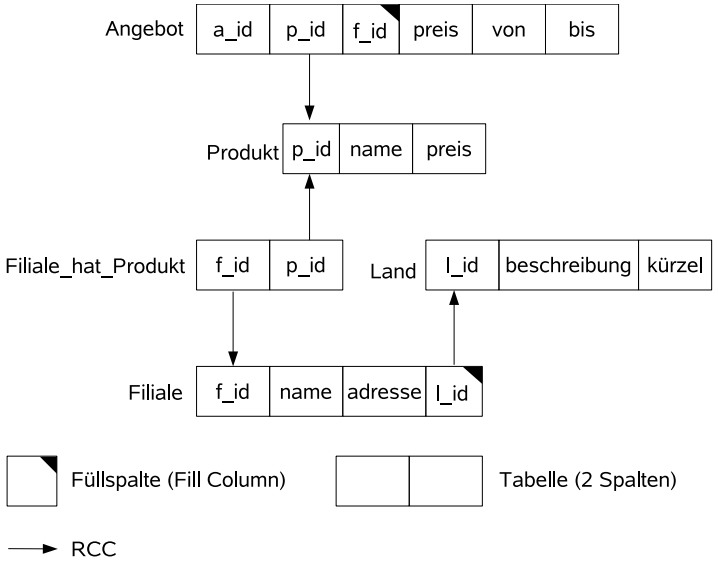

Abbildung 2.1: Beispiel einer Cache-Group-Definition.

Eine als Füllspalte definierte Spalte wird dazu benutzt sicherzustellen, dass ein Wert, sobald er durch eine Anfrage referenziert wird, wertvollständig im Cache gehalten wird.

Definition 2.2 (Wertvollständigkeit) *: Ein Wert w heißt* ***wertvollständig*** *in einer Spalte S.a genau dann, wenn alle Sätze aus $\sigma_{a=w}S_B$ in S enthalten sind [Büh06].*

Dabei lässt sich zu jeder Füllspalte auch eine Menge von Werten definieren, für die die Füllspalte keine Vollständigkeit herstellt, falls Tupel mit entsprechenden Werten eingelagert werden. Eine Anfrage, die einen entsprechenden Wert referenziert, muss also (zumindest in Teilen) immer an die Backend-Datenbank weitergeleitet werden. Auf diese Weise lassen sich Werte mit geringer Selektivität von der Einlagerung ausschließen. Eine Spalte, bei der jeder eingelagerte Wert wertvollständig ist, wird spaltenvollständig genannt.

Definition 2.3 (Spaltenvollständigkeit) *Eine Cache-Spalte S.a heißt genau dann **spaltenvollständig**, wenn alle Werte w in S.a wertvollständig sind [Büh06].*

Die Definition ist notwendig, da durch RCCs referenzierte Zielspalten zwar für einige Werte wertvollständig sind, aber nicht zwingend spaltenvollständig.

Hierbei ist ein RCC (Referenzieller Cache-Constraint) wie folgt definiert:

Definition 2.4 (RCC) *: Ein referenzieller Cache-Constraint $S.a \rightarrow T.b$ von einer Quellspalte S.a zu einer Zielspalte T.b ist genau dann erfüllt, wenn alle Werte w aus S.a vollständig in T.b sind [Büh06].*

Eine genauere Beschreibung von Cache Groups und ihren Eigenschaften findet sich in [HB07].

Für das Entwickeln einer Messumgebung, welche die Leistungsfähigkeit dieser Cache-Methode überprüfen soll, ist es notwendig, die Funktionen des Caches, die zur Anfragebearbeitung und zur Erhaltung der Konsistenz benötigt werden, näher zu betrachten. Während einer Messung sind diese Funktionen innerhalb einer Implementierung des Cache-Systems zu beobachten. Hierzu müssen geeignete Messwerte und Anfrageszenarien erstellt werden, wodurch ein grundlegendes Verständnis für den Aufbau des Cache-Systems unabdingbar ist.

In Abbildung 2.2 sind die Hauptkomponenten des ACCache-Systems, wie es derzeit in einer prototypischen Implementierung vorliegt, dargestellt [Mer05].

Die Darstellung zeigt vor allem die nebenläufig ablaufenden Prozesse. Die *Query Worker* bearbeiten eine Anfrage, die über die zur Verfügung gestellte JDBC-Schnittstelle aufgerufen wird. Bevor die Anfrage beantwortet werden kann, führt ein *Query Worker* eine sog. Sondierung (siehe 2.1) durch. Der *Fill Deamon* entscheidet nach jeder Anfrage, ob Werte nachgeladen werden müssen. Das Nachladen von Werten wird in Abschnitt 2.2 betrachtet. Der *Hit Counter* erfasst Daten über die Häufigkeit zugegriffener Tupel sowie Referenzzeitpunte. Mit dieser Funktion unterstützt er den *Garbage Collector* bei seiner Aufgabe, veraltete Tupel aus dem Cache zu verdrängen (vgl. 2.4). Der *Garbage Collector* verwendet die vom *Hit Counter* gesammelten Werte zur Implementierung einer geeigneten Verdrängungsstrategie. Die Verarbeitung einer Änderungsoperation (vgl. 2.3) wird derzeit nicht von der prototypischen Implementierung unterstützt. Trotzdem stellt die effiziente Lösung dieser Teilaufgabe in der Zukunft eine wichtige Herausforderung dar. Aus diesem Grund wird diese Funktion auch auf in der Zukunft erfassbare Messwerte untersucht.

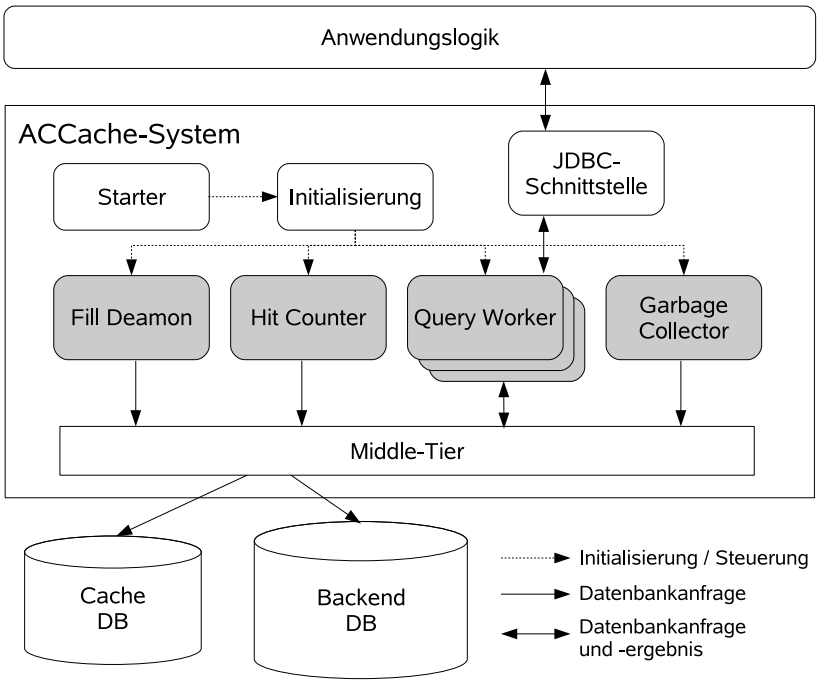

Abbildung 2.2: Die Hauptkomponenten des ACCache (früher CBCS) [Mer05].

2.1 Sondierung

Sobald eine Anfrage das Cache-System erreicht, muss die Sondierung durchgeführt werden. Dabei findet der Cache in einem ersten Schritt heraus, welche Teile einer Anfrage aus dem Cache heraus beantwortbar sind. Alle nicht aus dem Cache heraus beantwortbaren Teile müssen durch das Backend beantwortet werden [Büh06].

Der im zweiten Schritt erstellte Anfrageplan kann sowohl Backend- als auch Cache-Tabellen beinhalten. Da im ersten Schritt bereits festgestellt wurde, welche Tabellen des Caches sicher genutzt werden können, kann durch die Verwendung einer dieser Cache-Tabellen kein Fehler auftreten. Es spielt keine Rolle, dass die Cache-Tabelle nur einen Ausschnitt der auf dem Backend befindlichen Tabelle vorhält. Nach erfolgreicher Erstellung des Anfrageplans, kann die Auswertung der Anfrage durchgeführt werden [Büh06].

Die Durchführung der Sondierung stellt eine reine Mehrbelastung für das Gesamtsystem dar und wirkt sich direkt auf die Gesamtzeit aus, die benötigt wird, um eine Anfrage zu beantworten. Ohne Einsatz eines Cache-Systems fällt die Entscheidung über die Durchführbarkeit auf diese Art nicht an. Sie ist mit der Durchführungsentscheidung auf dem Backend nicht direkt vergleichbar. Dort wird z. B. nur die Existenz von über Namen angesprochenen Objekten mittels der Informationen aus dem Datenbankkatalog überprüft. Es stellt sich dabei aber nicht die Frage, welche Teile einer Tablelle abgespeichert sind, da immer die ganze Tabelle vorhanden ist.

Aus diesem Grund darf die Durchführung der Sondierung keinen hohen Aufwand darstellen. Die Zeit, welche die Sondierung für eine Anfrage benötigt, muss daher minimiert werden. Daraus ergibt sich die Notwendigkeit, die Zeit messen zu können.

Erst vor kurzem veröffentlichte Forschungsergebnisse haben gezeigt, dass die Sondierung auf Cache Groups durch entsprechende Optimierungsideen gut und schnell durchführbar ist [Büh06; Sch06].

Während der Sondierung wird entschieden, welche Tupelmengen, die für eine Anfrage benötigt werden, bereits auf dem Cache vorhanden sind und welche Tupelmengen vom Backend abgefragt werden müssen. Die Teilprädikate einer Anfrage, die durch Tupelmengen des Caches abgedeckt sind, stellen eine wichtige Messgröße dar. Zusammen mit allen Teilprädikaten einer Anfrage lassen sich auch die durch das Backend zu beantwortenden Teile ermitteln. Wird die Gesamtmenge der Anfrageprädikate nicht erfasst, kann man sie im Nachhinein stets herleiten, wenn zumindest der Text der zugehörigen Anfrage erfasst wurde. Aus diesen Werten lässt sich z. B. ablesen, wie schnell der Cache warmläuft oder ob sich der Einsatz eines Cache-Systems überhaupt lohnt. Außerdem lässt sich hierüber auch ermitteln, wie stark das Backend durch den Einsatz des Cache-Systems entlastet wird.

2.2 Laden bzw. Nachladen von Werten durch den Cache

Ein Cache-System ist nur dann sinnvoll, wenn für die Daten, auf die über den Cache zugegriffen wird, eine hohe Lokalität besteht. Sie definiert das Optimierungspotenzial des Cache-Systems. Daher kommt der Auswahl der Daten, die auf dem Cache vorgehalten werden, eine besondere Bedeutung zu. Eine einfache Art zu entscheiden, welche Daten auf dem Cache-System eine hohe Lokalität besitzen, ist, Tupelmengen häufig verwendeter Anfrageprädikate vorzuhalten. Dieses Ziel wird beim Constraint-basierten Datenbank-Caching primär verfolgt.

Der Administrator eines Cache-Systems kann darüber hinaus entscheidenden Einfluss auf die Menge der nachzuladenden Werte nehmen, indem er das Design der Cache-Group anpasst. Die Aufgabe, für einen Cache eine geeignete Cache-Group-Definition zu entwickeln, erfordert ein hohes Maß an Expertise. Zum einen muss darauf geachtet werden, dass Anfragen an den Cache nicht zu einer Flut von nachzuladenden Werten führen. Dies bedeutet gleichzeitig, dass der Entwickler die Anfragecharakteristika der Anwendungen, die auf den Cache zugreifen, kennen muss. Zum anderen muss darauf geachtet werden, dass Werte nachgeladen werden, die (wahrscheinlich) oft abgefragt werden, aber eher selten geändert werden.

Eine wichtige Messgröße für die Überprüfung dieser Funktion ist daher, wie viele Werte über welchen Zeitraum hinweg nachgeladen werden mussten und welche Anfragen das Nachladen ausgelöst haben. Nachgeladene Werte müssen außerdem in der Zukunft dazu führen, dass mehr Anfrageteile direkt aus dem Cache heraus beantwortet werden können.

Die Erfassung dieser Werte kann in einem weiterführenden Entwicklungsschritt des Cache-Systems eventuell dazu genutzt werden, bessere Cache-Group-Designs durch das System vorschlagen zu lassen. Hiermit könnte man dem Ziel, das Design einer geeigneten Cache-Group-Definition komplett dem System zu überlassen, einen Schritt näher kommen.

Da das Nachladen von Werten durch einen asynchron laufenden Nachladeprozess (*Fill Deamon*) geschieht, ist der Cache in der Lage, eine Anfrage lange vor dem

Abschluss aller Nachladeoperationen zu beantworten. Der entstandene Aufwand belastet also in erster Linie das Backend-System. Werden jedoch zu viele oder unpassende Werte nachgeladen, so kann dies wiederum zu Leistungseinbußen führen. So kann z. B. das Nachladen eines Wertes, der oft geändert wird, dazu führen, dass dieser ständig vom Backend aus auf dem Cache-System gesperrt werden muss, was einen sehr hohen Kommunikationsaufwand zur Folge hätte. Zyklisches Nachladen kann dazu führen, dass der komplette Inhalt einer Tabelle nachgeladen wird, obwohl dieser in den seltensten Fällen benötigt wird.

2.3 Durchführung von Änderungen

In der Einleitung zu diesem Kapitel wurde bereits erwähnt, dass die Durchführung von Änderungsanfragen über ein Cache-System zusätzliche Belastungen erzeugt. In der derzeitigen Version der prototypischen Implementierung ist dieses Progrobem programmtechnisch noch nicht berücksichtigt. Hierdurch wird es im Rahmen dieser Arbeit auch nicht möglich, das Verhalten von schreibenden Transaktionen explizit zu vermessen.

Es gibt jedoch eine Vielzahl möglicher Lösungen für die Synchronisation verteilter Replikate, die hier zur Anwendung kommen könnten. Diese sind vornehmlich für Replikationsverfahren entwickelt worden. Meist wird dabei eine breitbandige Verbindung zwischen den Replikaten angenommen. Da ein Datenbank-Cache eingesetzt wird, um einen teuren entfernten Zugriff zu vermeiden, ist der Einsatz eines syntaktischen Verfahrens [1] aus dem Bereich der Datenbankreplikation nicht optimal. Auch im Bereich des Mobile Computing sind schlechte Verbindungen zu zentralen Systemen problematisch. Dort werden meist semantische Verfahren mit optimistischen Annahmen verwendet. Diese benötigen im Einzelfall aber den Eingriff des Menschen und können somit in ihrer Reinform für das Datenbank-Caching nicht eingesetzt werden.

Die Ideen aus den beiden Bereichen Replikation und Mobile Computing können aber wichtige Hinweise bieten, um ein möglichst gutes Verfahren für das Constraint-basierte Datenbank-Caching zu entwickeln.

2.4 Löschen von veralteten Cache Inhalten

Beim Löschen von Cache-Inhalten muss stets sichergestellt werden, dass alle angelegten Constraints auf dem Cache ihre Gültigkeit beibehalten. Die einfachste Methode dies zu tun ist, den Cache zu einem bestimmten Zeitpunkt komplett zu löschen. Will man dies nicht tun, genügt es allerdings nicht, einzelne veraltete Tupel zu identifizieren, um diese dann zu löschen. Es müssen ganze Mengen von Tupeln identifiziert werden, die bei gemeinsamer Löschung die Konsistenz auf dem Cache weiterhin sicherstellen.

[1] Syntaktische Synchronisationsverfahren stellen aufgrund der Implementierung des Verfahrens die sog. One-Copy-Serialisierbarkeit sicher. Diese stehen den semantischen bzw. optimistischen Verfahren gegenüber, welche die One-Copy-Serialisierbarkeit nicht stets garantieren können. Der Begriff der One-Copy-Serialisierbarkeit stellt die notwendige Erweiterung der Serialisierbarkeitsdefinition beim Vorhandensein verteilter Replikate dar. Die häufigere Unterteilung von Replikationsverfahren ist die Unterscheidung zwischen synchroner und asynchroner Replikation. Nähere Informationen hierzu befinden sich z. B. in [Bur98; BD96; Dad96].

Im vorliegenden Prototyp wurde hierfür eine erste Lösung implementiert. Sie wird in [Mer05] vorgestellt. Die Verdrängungsstrategie basiert auf dem LRU-Verfahren. Die hierfür benötigten Referenzzeitpunkte werden während der Sondierung ermittelt und aktualisiert.

Während der Vermessung eines Constraint-basierten Datenbank-Caches spielt die Beeinflussung des Systems durch den *Garbage Collector* nur eine indirekte Rolle. Der Löschvorgang wird durch einen asynchronen Prozess vorgenommen und beeinflusst somit die Anfragebeantwortung nicht direkt.

Trotzdem kann eine Ineffizienz in dieser Komponente erheblichen Schaden anrichten. Eine schlecht gewählte Verdrängungsstrategie kann dazu führen, dass Tupel gelöscht werden, die in naher Zukunft wieder benötigt werden. Somit sind vornehmlich Messungen sinnvoll, die Untersuchungen zulassen, wie gut zu löschende Tupelmengen ausgewählt werden.

2.5 Zusammenfassung

Das Constraint-basierte Datenbank-Caching definiert Cache-Groups, in die Werte aus dem Backend aufgenommen werden. Die aufgenommenen Tupel müssen dabei stets die durch eine Cache-Group definierten Constraints erfüllen, die es ermöglichen, sehr flexibel verschiedenste Teile von Anfragen durch den Cache zu beantworten.

Um die Leistung des Cache-Systems zu erfassen, sind folgende Messwerte besonders wichtig:

- für die Sondierung:
  - die Zeit, die für die Sondierung benötigt wurde
  - die Teile der Anfrage, die durch den Cache beantwortet werden konnten, bzw. die Teile der Anfrage, die auf dem Backend beantwortet werden mussten
- für das Nachladen von Werten:
  - wie viele Tupel nachgeladen wurden
  - welche Tupel nachgeladen wurden
  - wann Tupel nachgeladen wurden
- für die Durchführung von Änderungen:
  - die Zeit, die das Durchführen einer Änderung für ein Datenelement benötigt
  - welche Elemente geändert wurden
- für das Löschen veralteter Inhalte:
  - wann welche Tupelmengen gelöscht wurden
  - wie lange das Löschen dauert hat

Für das Gesamtsystem ist es natürlich wichtig, in welcher Zeit eine Anfrage oder eine ganze Transaktion durchgeführt werden konnte. Diese Werte sind aber vornehmlich aus der Sicht eines Benutzers interessant und werden nicht auf dem Cache-System sondern auf dem Client-System gemessen. Auf die Erfassung dieser Messwerte wird in Abschnitt 3.5.2 nochmals gesondert hingewiesen.

3. Messdaten in verteilten Systemen

Das nachfolgende Kapitel beschreibt konzeptionell die Aufgabengebiete und entsprechende Lösungsansätze, die zur Durchführung einer Messung in verteilten Systemen von Bedeutung sind. Der Schwerpunkt liegt hierbei auf der Entwicklung eines Frameworks, welches bei der Implementierung von Messungen für ein spezifisches System herangezogen werden kann und allgemeingültige Aufgaben löst. Hierbei soll es keine Rolle spielen, ob das betrachtete System eine Client-Server- oder P2P-Architektur aufweist. Ebenso soll die Art der erbrachten Dienste durch die verteilten Komponenten uneingeschränkt bleiben.

Im nachfolgenden Abschnitt werden zur besseren Strukturierung der durch das Framework zu unterstützenden Aufgaben und Funktionen übergeordnete Aufgabengebiete gebildet. Danach werden für diese Aufgabengebiete die Einzelaufgaben herausgearbeitet, für die dann Lösungsansätze und Konzepte diskutiert werden.

Es sind grundsätzlich die Begriffe Messumgebung und Messframework (oder kurz Framework) voneinander zu unterscheiden. Die Messumgebung ist eine Anwendung, die durch das Framework zur Verfügung gestellt wird, um den Messablauf zu koordinieren. Das Framework definiert darüber hinaus die Schnittstellen und Konfigurationsmöglichkeiten, die es erlauben, beliebige verteilte Systeme (durch eine entsprechende Implementierung) zu vermessen. Abbildung 3.1 versucht, diesen Zusammenhang anschaulich darzustellen.

3.1 Identifizierung der Aufgabengebiete

Um leichter die Aufgabengebiete identifizieren zu können, für die das zu entwickelnde Framework eine Unterstützung bieten muss, werden zunächst allgemeine Eigenschaften verteilter Systeme betrachtet. Diese Betrachtung erlaubt es bereits, erste später benötigte Grundfunktionalitäten zu identifizieren. Nachdem ein Aufgabengebiet identifiziert wurde, wird dies durch eine Aufzählung nochmals hervorgehoben und zusammengefasst.

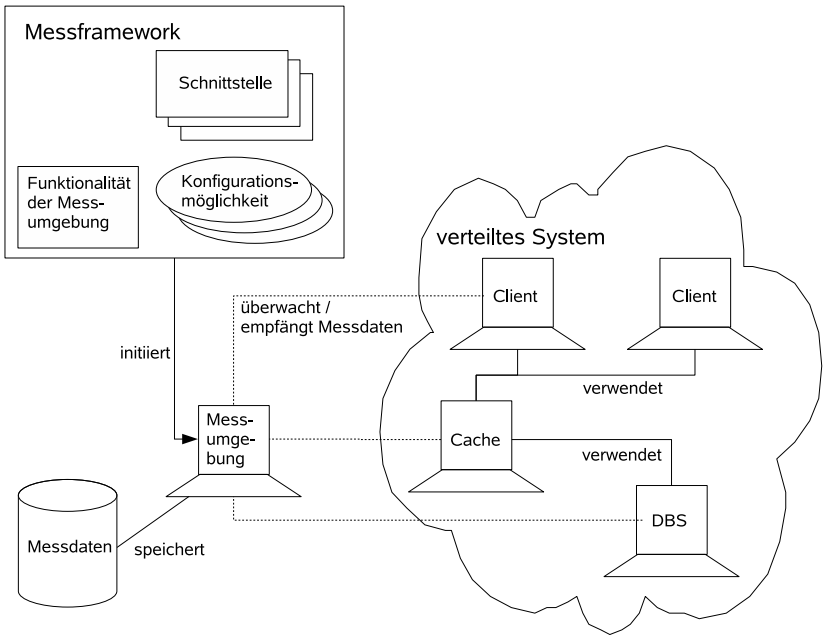

Abbildung 3.1: Begriffsabgrenzung Messumgebung - Messframework

Ein verteiltes System ist nach der Definition von Andrew Tanenbaum ein Zusammenschluss unabhängiger Computer, welcher sich für den Benutzer als ein einzelnes System präsentiert [TS03]. Dabei repräsentiert das Wort *unabhängig* die Eigenschaft, dass ein verteiltes System aus einer Menge interagierender Prozesse besteht, die über keinen gemeinsamen Speicher verfügen und daher über Nachrichten miteinander kommunizieren.

Jeder einzelne Prozess eines verteilten Systems kann für die Messung relevante Daten liefern. Damit das Framework diese erfassen kann, muss es in der Lage sein, die Struktur bzw. Teilstruktur des verteilten Systems, welches für die Messung relevant ist, zu erfassen. Diese grundlegende Strukturierung ermöglicht es dem Framework, alle zu erledigenden Aufgaben den Komponenten des verteiltes Systems zuzuordnen. Dies beginnt z. B. bei der Aufgabe, den Komponenten mitzuteilen, dass die Messung startet, und endet bei der Möglichkeit, die erfassten Messdaten entsprechend strukturiert abzulegen. Darüber hinaus können durch die Erfassung logische Abhängigkeiten überprüft werden. Wurde dem Framework z. B. mitgeteilt, dass ein Datenbanksystem nur über eine JDBC-Schnittstelle ansprechbar ist, kann es sicherstellen, dass nur Client-Systeme, die JDBC-Transaktionen ausführen können, mit dieser Datenbank verbunden werden. Die Erfassung der Struktur ist somit die Grundlage für jede weitere Aufgabenerfüllung.

- Es ist notwendig, die Struktur und die Eigenschaften des verteilten Systems erfassen zu können. Dies umfasst z. B. Aufgaben wie das Anlegen der am Test beteiligten Prozesse bzw. Anwendungen und die Möglichkeit, diese anzusprechen, sowie das Anlegen von Verbindungen zwischen den verschiedenen Komponenten usw.

In einem verteilten System stehen zwei miteinander agierende Teilprozesse immer in einer Client-Server-Beziehung. Dies gilt auch, wenn ein Prozess Diensterbringer

und gleichzeitig Dienstanforderer ist, wie es in einem P2P-Netzwerk der Fall ist. Durch einen initiierenden Client-Aufruf werden jeweils Aufgaben in das verteilte System eingebracht, die durch den aufgerufenen Serverprozess abgearbeitet werden. Das Framework soll in der Lage sein, Client-Aufrufe zu simulieren, um somit dem Gesamtsystem eine Arbeitslast zu diktieren. In den meisten Fällen entsprechen die zu simulierenden Client-Aufrufe den Handlungen eines Benutzers. Somit muss das Framework in der Lage sein, das Verhalten eines Benutzers bzw. Client-Systems simulieren zu können

- Das Benutzerverhalten muss simuliert werden können. Das bedeutet, dass Arbeitsschritte, die ein simulierter Benutzer ausführen soll, definiert und programmiert werden müssen.

Um Messungen automatisch vornehmen zu können, müssen die Komponenten eines verteilten Systems durch die Messumgebung selbstständig konfigurierbar sein. Die notwendigen Arbeitschritte zur automatischen Konfiguration eines Knotens können dabei stark unterschiedlich ausfallen. Sie können sich erstrecken vom einfachen Starten eines Dienstes bis hin zur kompletten Installation eines Rechners über das Netz, mit allen benötigten Diensten, Benutzern und Programmen.

Das Framework muss somit in der Lage sein, die Entwicklung automatischer Konfigurationsschritte für einen Knoten des verteilten Netzes zu unterstützen, diese durchzuführen und zu überwachen.

- Verwaltung automatischer Konfigurationsschritte für jeden Knoten.

„Automatisch“ in diesem Zusammenhang bedeutet aber auch, dass ein oder mehrere bereits konfigurierte Messabläufe vom Framework ohne weitere Benutzereinwirkung selbständig nacheinander vorgenommen werden können bzw. auf Wunsch auch mehrfach ausgeführt werden. Auf Fehler während der Messung muss das Framework möglichst robust reagieren. So sollte z. B. der Ausfall eines Dienstes innerhalb des zu vermessenden verteilten Systems die Durchführung eines nachfolgenden Tests nicht behindern, wenn dieser ohne Fehler durchgeführt werden könnte.

- Messabläufe müssen selbständig, evtl. auch mehrfach nacheinander, durchlaufen werden können.

Eine weiteres zentrales Aufgabengebiet ist die Verwaltung der Messdaten, die auf den jeweiligen Knoten des verteilten Systems anfallen. Die während einer Messung anfallenden Daten müssen dem Framework in einheitlicher Art und Weise übergeben werden, so dass eventuell bestehende Beziehungen zwischen Messwerten abgebildet werden können. Darüber hinaus können auch Messwerte, die auf verschiedenen Knoten erfasst wurden, miteinander in Beziehung stehen. So können z. B. die während des Ablaufs einer Datenbanktransaktion erfassten Messdaten in einer Client-Server-Umgebung miteinander in Beziehung stehen. Für den Client könnte hierbei z. B. die Bearbeitungszeit der Transaktion und auf dem Server die Bearbeitungszeit jeder

Anfrage innerhalb der ausgeführten Transaktion gemessen werden. Um die Bearbeitungszeit jeder Anfrage in Beziehung zu setzen mit der Bearbeitungszeit der Transaktion, muss eine entsprechende Verbindung zwischen den Werten gepflegt werden.

Darüber hinaus muss der Transport der erfassten Daten zu einem zentralen Speicherort für alle Messergebnisse für den Benutzer transparent erfolgen.

- Die Verarbeitung der Messdaten muss durch das Framework unterstützt werden.
- Die Beziehungen zwischen Messwerten müssen abgebildet werden können.

Nachdem die Messung abgeschlossen ist und die Messergebnisse erfasst wurden, müssen die Messergebnisse ausgewertet werden. Oft ist es erwünscht, die Messwerte über die Aufrufbeziehungen miteinander zu verknüpfen oder Messwerte zu aussagekräftigen Kennwerten zu aggregieren. Darüber hinaus muss sichergestellt sein, dass die erfassten Daten durch geeignete Exportfunktionen zur Weiterverarbeitung in externe Applikationen überführt werden können. Diese Auswertungsfunktionalitäten werden durch externe Applikationen meist wesentlich besser unterstützt, als dies das Framework anbieten könnte. Deshalb ist es notwendig, die Ablage der Messwerte und alle vorhandenen Kontextinformationen dazu entsprechend flexibel und komfortabel zu gestalten.

- Die Auswertung von erfassten Daten muss bestmöglich unterstützt werden.

Im nachfolgenden werden die vorangehend identifizierten Aufgabengebiete näher betrachtet und Lösungskonzepte entwickelt.

3.2 Konfiguration des zu vermessenden Systems

Damit das Framework in der Lage ist, die einzelnen Knoten (Teilanwendungen) eines verteilten Systems zu vermessen, müssen diese dem Framework bekannt sein. Eine Teilanwendung des verteilten Systems wird in der Folge als Arbeitsknoten (*Working Node*) bezeichnet. Der Begriff Arbeitsknoten wurde gewählt, weil die Arbeit, die eine Teilanwendung verrichtet, vermessen werden soll.

Definition 3.1 (Arbeitsknoten) *Ein Arbeitsknoten repräsentiert eine Anwendung innerhalb des zu vermessenden verteilten Systems, die für die Messung relevante Daten liefert und über eine beliebige Schnittstelle Funktionalität zur Verfügung stellt.*

Anhand des in der Einleitung erwähnten Szenarios und der gewünschten Eigenschaft, Messungen automatisch durchführen zu können, ergeben sich für einen Arbeitsknoten mehrere Eigenschaften, die das Framework soweit wie möglich abbilden muss.

3.2.1 Eigenschaften eines Arbeitsknotens

Aufrufbarkeit

Aus der Definition ist ersichtlich, dass jeder Arbeitsknoten über eine Schnittstelle aufrufbar sein muss. Für das Framework bedeutet dies, dass es möglich sein muss, jeweils eine Schnittstelle für einen Arbeitsknoten auszuzeichnen und die Kontextinformationen, die zur Beschreibung bzw. Aufruf der Schnittstelle relevant sind, abzuspeichern. Hierbei spielt auch der Typ der Schnittstelle eine große Rolle. Es muss z. B. unterscheidbar sein, ob ein Arbeitsknoten eine JDBC-Schnittstelle oder eine RMI-Schnittstelle zur Verfügung stellt. Mittels des Typs der Schnittstelle kann später entschieden werden, ob ein Arbeitsknoten mit einem anderen überhaupt in Beziehung stehen kann. Es soll nicht möglich sein, dass ein Arbeitsknoten innerhalb des Frameworks mehrere Schnittstellen auszeichnen kann, auch wenn diese z. B. genau den gleichen Funktionsumfang anbieten. Entscheidend für das Framework ist, dass ein Arbeitsknoten immer nur über eine Schnittstelle in der Messung verwendet wird. Wird die gleiche Anwendung über zwei verschiedene Schnittstellen angesprochen, ist es notwendig, zwei voneinander unterschiedliche Arbeitsknoten auszuzeichnen. Dies kann sogar durchaus erwünscht sein, da hierdurch auch verschiedene Beobachter ausgezeichnet werden können. Die Kontextinformationen der Schnittstelle können auch dazu verwendet werden, eine Anwendung zu initialisieren (siehe auch Kapitel 3.4).

Beziehungen zu anderen Arbeitsknoten

Arbeitsknoten können untereinander in Beziehung stehen. Wenn in einem verteilten System z. B. Knoten A auf Knoten B zugreift, um einen Dienst erbringen zu können, so steht der Knoten A mit dem Knoten B in Beziehung. Eine solche Beziehung wird in der Folge als Aufrufbeziehung bezeichnet. Eine Aufrufbeziehung ist dabei eine gerichtete Kante von A nach B. Wenn B seinerseits auch Funktionalitäten auf A aufrufen kann, muss eine zusätzliche Kante in die Konfiguration eingefügt werden.

Beobachtbarkeit

Damit überhaupt Messdaten erfasst werden können, muss die Anwendung, die der Arbeitsknoten repräsentiert, beobachtbar sein. Dies ist über mehrere Wege möglich.

Liegt die zu beobachtende Anwendung mit ihren Quellen vor, ist ein **freier Eingriff** möglich. In diesem Fall ist es möglich, beliebige Teilbereiche der Anwendung um eine Schnittstelle zu erweitern, die es ermöglicht, relevante Daten von diesem Teilbereich zu erhalten. Typisch hierfür ist eine Implementierung nach dem Observer-Entwurfsmuster.

Ist ein direkter Eingriff auf die Anwendung nicht möglich, kann eventuell eine vom Hersteller implementierte **Schnittstelle** verwendet werden, um Messdaten abzufragen.

Wenn auch diese Möglichkeit nicht besteht, kann die Zuhilfenahme von Funktionalität des umgebenden **Betriebssystems** helfen, relevante Daten abzufragen.

Besteht auch diese Möglichkeit nicht, so ist der Arbeitsknoten **nicht beobachtbar**. Dies bedeutet jedoch nicht, dass keine Aussagen über diesen Knoten gemacht werden können. Jedoch sind die Leistungsinformationen nur durch Beobachtung von

Komponenten ermittelbar, die von außen mit dem Knoten kommunizieren. Greift z. B. eine Client-Anwendung auf einen nicht beobachtbaren Web Service zu, so stellt der Arbeitsknoten *WebService* keine Messinformationen bereit, sondern nur der zugreifende Client-Arbeitsknoten.

Jeder im Framework angelegte Arbeitsknoten kann durch mehrere Beobachterkomponenten, die durch den Entwickler eines Messaufbaus zu implementieren sind, relevante Messdaten erfassen. Das Framework definiert dabei nur die minimale Schnittstelle einer Beobachterkomponente. Diese muss durch eine geeignete Implementierung eine Anwendung oder Teile davon überwachen. Es ist möglich, einige generische Beobachter für verschiedene Betriebssysteme zu implementieren. So wäre es z. B. denkbar, einen Beobachter für Linux zu implementieren, der für eine beliebige Anwendung jeweils die Prozessorauslastung über die Zeit der Messung abfragt.

Damit ein Arbeitsknoten weiß, welche Beobachter er als Messdatenlieferant benutzen kann, müssen diese dem Arbeitsknoten in der Konfiguration zugewiesen werden. Darüber hinaus ist es nicht immer gewünscht, dass jede Beobachterkomponente bei einer Messung aktiv ist. Falls für einen ACCache z. B. Beobachter für die Sondierung, das Nachladen von Werten und den Garbage Collector implementiert sind, kann man dadurch die Sondierung getrennt überwachen lassen. Daher sollte in der Konfiguration die Möglichkeit geschaffen werden, auf den Aktivitätsgrad einer Beobachtungskomponente Einfluss zu nehmen. Dies kann bedeuten, dass der Beobachter komplett ausgeschaltet werden kann oder dass dieser nur bestimmte Messwerte aus der Menge ihm möglicher beobachtbarer Werte erfasst.

Messdatenerfasser

Der Arbeitsknoten erhält von den Beobachtern die erfassten Messdaten und übergibt sie an die Datenschreiberkomponente, welche in der Lage ist, die Daten abzuspeichern. Damit die Daten nach ihrer Speicherung leicht zu einer späteren Analyse herangezogen werden können, werden diese standardmäßig in einer SQL-Datenbank abgelegt. Das Framework implementiert eine generische Möglichkeit, übermittelte Messdaten abzuspeichern. Jedoch muss das Framework darüber hinaus in der Lage sein, andere Speichermöglichkeiten für die Messdaten zuzulassen. Für einen Arbeitsknoten bedeutet dies, dass festgelegt werden muss, welche Schreibimplementierung zur Ablage der Daten verwendet wird.

Automatisch konfigurierbar

Eine Kernaufgabe des Frameworks ist die automatische Konfigurierbarkeit des Messablaufes. Da jeder Arbeitsknoten eine Anwendung darstellt, die für sich autonom konfigurierbar sein kann, muss das Framework die Möglichkeit bieten, entsprechende Konfigurationsschritte auszuzeichnen. Diese werden dann vor dem Beginn der Messung für jeden Arbeitsknoten ausgeführt. Damit auch nach dem Ablauf der Messung das Gesamtsystem wieder in seinen Ursprungszustand versetzt werden kann, müssen die Konfigurationschritte wieder rückgängig gemacht werden können, falls dies erwünscht oder notwendig ist. Das Framework sollte darüber hinaus in der Lage sein, die Durchführung eines Initialisierungsschrittes zu wiederholen, falls dieser beim ersten Versuch, den Messablauf zu starten, nicht erfolgreich war. Schritte, die bereits erfolgreich abgeschlossen sind, werden nicht wiederholt.

Um eine größtmögliche Unterstützung für die Implementierung automatischer Konfigurationsschritte zu gewährleisten, sollte das Framework Funktionalitäten zur Verfügung stellen, die es dem Entwickler erleichtern, komplexe Konfigurationen vorzunehmen. Außerdem sollten gewisse Standardaufgaben, wie z. B. das Starten der Datenbank oder das Anlegen eines Schemas in einer Datenbank, unterstützt werden. Hilfreich ist auch eine Möglichkeit, über einen SSH-Zugriff auf einem entfernten System Befehle abzusetzen.

3.2.2 Abbildung der Eigenschaften eines Arbeitsknotens

Abbildung 3.2 stellt die konzeptionelle Sicht eines Arbeitsknotens in einem ER-Diagramm dar.

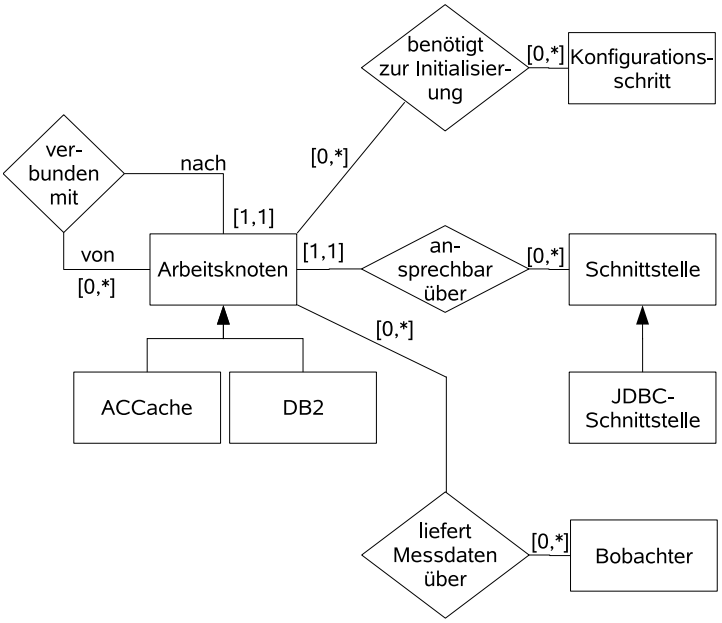

Abbildung 3.2: Grundlegende Eigenschaften eines Arbeitsknotens

Da die gewählte Implementierungssprache *Java* objektorientiert ist, empfiehlt sich die Abbildung der Eigenschaften und benötigten Funktionalitäten eines Arbeitsknotens durch entsprechende Klassen. Damit alle speziellen Eigenschaften eines bestimmten Typs von Arbeitsknoten durch das Framework erfassbar sind, wird das Bilden einer entsprechenden Unterklasse fest vorgeschrieben. Zum Vermessen eines Constraint-basierten Datenbank-Caches muss z. B. die Unterklasse ACCache gebildet werden, die nur die Auszeichnung einer JDBC-Schnittstelle zulässt. Wie die Eigenschaften eines Arbeitsknotens genau umgesetzt wurden, wird in Kapitel 4 erklärt.

3.2.3 Informationen eines Messablaufes

Damit eine Messung vorgenommen werden kann, muss eine Vielzahl an Informationen ausgezeichnet werden. Betrachtet man zunächst nur einmal die Eigenschaften, die ein Arbeitsknoten besitzt, sind für jede einzelne dieser Eigenschaften wiederum Auszeichnungen nötig. Muss z. B. vor einer Messung ein Datenbankschema kreiert werden, damit ein Arbeitsknoten (bzw. die repräsentierte Anwendung) lauffähig ist,

so müssen das Schema und die Verbindung zur Datenbank bekannt sein. Das Beispiel soll auch verdeutlichen, dass durch jeden neu implementierten Konfigurationsschritt eventuell neue Metadaten benötigt werden. Nachfolgend wird daher versucht, für die komplette Durchführung einer Messung die benötigten Informationen zu gliedern.

Die Messung wird durch die Messumgebung vorgenommen und überwacht. Somit sind alle Informationen der Messumgebung zu übergeben. Das Framework bietet dabei die Möglichkeit, Erweiterungen vorzunehmen, wie z. B. die Implementierung bzw. Auszeichnung neuer Konfigurationsschritte, Aufrufschnittstellen oder Arbeitsknoten.

Grundsätzlich sind alle Informationen der Messumgebung zugeordnet. Trotzdem empfiehlt es sich, die benötigten Informationen zu gliedern. So gibt es z. B. Informationen, die direkt für den korrekten Ablauf der Messumgebung benötigt werden. Ein Beispiel hierfür ist die Information, wohin die erfassten Messdaten zu speichern sind.

Damit die Struktur des zu vermessenden Systems erfasst werden kann, müssen entsprechende Untertypen von Arbeitsknoten gebildet werden. Für einen bestimmten Arbeitsknotentyp lassen sich Informationen erfassen, die jeder Instanz dieses Typs gemein sind. Ein Beispiel hierfür sind die Informationen über benutzbare Beobachter. In den nachfolgenden Abschnitten wird sich zeigen, dass vor allem die Auszeichnung verwendbarer Ausführungskontexte in diese Kategorie fällt (vgl. 3.5.3).

Die konkrete Struktur eines Messablaufes benötigt detaillierte Informationen über die vorhandenen Instanzen von Arbeitsknoten. Dies umfasst z. B. die Informationen über die Aufrufschnittstelle, die Aufrufbeziehungen zu anderen Arbeitsknoten, die zu verwendenden Beobachter, die durchzuführenden Initialisierungsschritte.

Somit lassen sich die zur Konfiguration benötigten Informationen in vier Kategorien gliedern:

- Informationen, die direkt der Messumgebung zugeordnet werden
- Informationen, die einem Arbeitsknotentyp zugeordnet werden
- Informationen, die einer bestimmten Arbeitsknoteninstanz zugeordnet werden und
- unabhängige, freie Konfigurationsinformationen z. B. für Konfigurationsschritte

3.3 Simulation fiktiver Benutzer

Ein Benutzer greift innerhalb eines verteilten Systems über eine Client-Anwendung auf andere Teilanwendungen eines verteilten Systems zu. Dabei verwendet er als Schnittstelle z. B. eine grafische Benutzeroberfläche. Aus den Aktionen des Benutzers werden dabei Anfragen an eine Datenbank durch die Client-Anwendung abgeleitet und ausgeführt. Der Benutzer definiert auf diese Weise eine Arbeitslast, die von verschiedenen Teilanwendungen eines verteilten Systems (z. B. einem Datenbanksystem) bearbeitet werden muss. Nach den bisher vorgestellten Erkenntnissen müsste

die Client-Anwendung, als eine Teilanwendung des verteilten Systems, welche für die Messung relevante Messdaten liefert, als Arbeitsknoten implementiert werden mit dem Ziel, die Vorgänge auf dem Client zu beobachten. Wenn die Client-Anwendung den benötigten Grad an Beobachtbarkeit zur Verfügung stellt, ist diese Vorgehensweise ohne weiteres möglich. Wenn die Notwendigkeit oder der Wunsch besteht, ein reales System mit realen Benutzern zu testen, ist sie sogar unausweichlich.

Der durchaus häufigere Fall ist, dass eine Simulation des Benutzerverhaltens bzw. einer Client-Anwendung gewünscht ist. Sei es, weil für einen Test gar nicht genug echte Benutzer zur Verfügung stehen, die Client-Anwendung nicht beobachtbar ist oder die Grenzen der Leistungsfähigkeit nur durch Rechnerleistung ausgetestet werden können. Das Framework muss also die Simulation einer Client-Anwendung und deren mögliches Anfrageverhalten simulieren können. Dies kann gelöst werden, indem das Framework eine spezialisierte Implementierung eines Arbeitsknotens zur Verfügung stellt. In der Folge wird eine solche Implementierung immer als *simulierter Client* bezeichnet. Dieser benötigt, damit die Messumgebung ihn ansprechen kann, eine geeignete Aufrufschnittstelle.

Da die Programmiersprache Java verwendet wird, empfiehlt sich eine Implementierung, die über RMI (Remote Method Invocation) angesprochen werden kann. Die Notwendigkeit einer entfernten Aufrufmöglichkeit besteht, da die zu simulierenden Client-Anwendungen verteilte Komponenten des Gesamtsystems darstellen. Wären diese innerhalb der Ablaufumgebung der Messumgebung realisiert, würde dies die Echtheit der Messung verfälschen, da mehrere Client-Anwendungen natürlicherweise in verschiedenen voneinander unabhängigen Ablaufumgebungen ablaufen.

Eigenschaften einer simulierten Client-Anwendung

Die Simulation einer Client-Anwendung wird im Framework durch einen spezialisierten Arbeitsknoten realisiert. Somit sind alle Eigenschaften und Aufgaben eines Arbeitsknotens, wie sie im vorangegangenen Abschnitt besprochen wurden, auch für die Simulation einer Client-Anwendung gültig. Zusätzlich muss der Arbeitsknoten aber noch weitere Aufgaben übernehmen.

Seine Hauptaufgabe besteht darin, eine **Arbeitslast zu simulieren**. Damit dies möglich wird, muss das Framework die Definition von Arbeitsschritten unterstützen, die einem zu simulierenden Client-Arbeitsknoten zugewiesen werden können. Mit Hilfe der Menge der zugewiesenen Arbeitschritte lässt sich dann eine Arbeitslast konstruieren. Dies ist z. B. möglich, indem Infornationen über die Häufigkeit der Ausführung der Schritte für den Arbeitsknoten definiert werden. Mit Hilfe eines Schedulers wird es dann möglich, die Arbeitslast zu generieren (s. Abbildung 3.3).

Einplanung und Verteilung von Arbeitsschritten

Bei der Einplanung von Arbeitsschritten innerhalb des Frameworks spielen zwei Faktoren eine wichtige Rolle: Zum einen, wie oft ein Arbeitsschritt, im Verhältnis zu anderen, durch den simulierten Client aufzurufen ist und zum anderen, über welche Verbindung ein Arbeitsschritt durchgeführt wird.

Der Scheduler übernimmt die Aufgabe zu entscheiden, welcher Arbeitsschritt als nächstes einzuplanen ist. Er übernimmt nicht die Aufgabe, Prozessen Rechenzeit

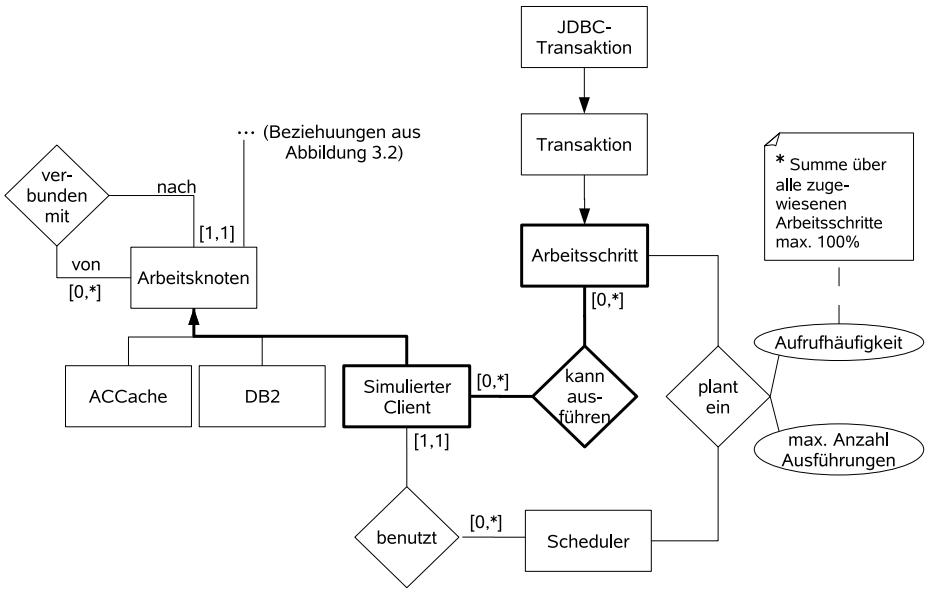

Abbildung 3.3: Die zusätzlichen Eigenschaften eines Client-Arbeitsknotens

zu gewähren, wie es der Scheduler eines Betriebssystems zusätzlich tut. Ein Arbeitschritt, sofern er als nächstes auszuführen ist, wird nach dem Starten durch den Scheduler des Frameworks nicht mehr unterbrochen.

Für einen simulierten Client sind nun mehrere Implementierungen möglich. So ist es z. B. möglich, einen Scheduler zu verwenden und die Verteilung des nächsten Arbeitsschrittes durch eine andere Komponente (ähnlich einem Router) bestimmen zu lassen. Dieser könnte dann eine Verbindung zufällig auswählen, feste Häufigkeitsvorgaben für die Verbindungen verwenden oder den Arbeitsschritt über alle möglichen Verbindungen ausführen. Dies würde voraussetzen, dass das Framework in der Lage ist, für einen Arbeitsschritt alle Verbindungen zu bestimmen, über die dieser ausgeführt werden kann. Darüber hinaus ließe sich auch für jede ausgehende Verbindung ein separater Scheduler verwenden, der pro Verbindung jeweils eine Teilmenge der ausgezeichneten Arbeitschritte einplant.

Abbildung 3.4 zeigt ein Beispiel für ein mögliches Zusammenspiel zwischen Scheduler und Router. Der Scheduler gibt eine neue JDBC-Transkaktion vom Typ c als nächsten Arbeisschritt vor. Dieser wird vom Router zur Verarbeitung auf der *DB2* vorgesehen. Sobald der Arbeitschritt a, der zurzeit auf der *DB2* abgearbeitet wird, fertig ist, wird c gestartet. Die Abbildung verdeutlicht auch, dass zurzeit eine Transaktion des Typs c über den Cache ausgeführt wird. Der Arbeitsschritt b wird gerade an die Schnittstelle des Dateiservers weitergeleitet.

Die daraus entstehende Fülle an Lösungen, die Arbeitslast, die eine Client-Anwendung erzeugen soll, zu verteilen, lässt sich dramatisch einschränken, indem der simulierte Client nur eine Verbindung zu Teilanwendungen des verteilten Systems nutzen darf. Eine Router-Komponente wäre dann nicht mehr notwendig, da es nur einen Weg gibt, übergebene Arbeitschritte auszuführen. Diese Lösung beschränkt auch kaum die Möglichkeit, Arbeitslasten zu generieren. Für mehrere Verbindungen sind dann lediglich mehrere Client-Instanzen notwendig (s. Abbildung 3.5).

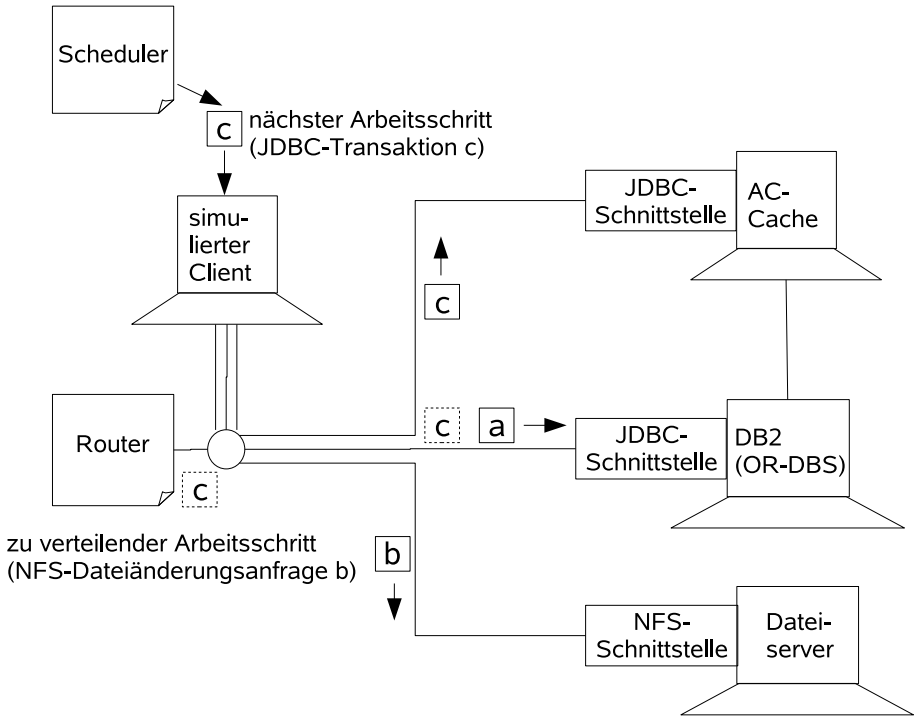

Abbildung 3.4: Beispiel für die Einplanungs- und Verteilmöglichkeiten von Arbeitsschritten.

Das entwickelte Framework verwendet für die Implementierung eines simulierten Clients eine Scheduler-Komponente und die Möglichkeit, nur eine Verbindung zu spezifizieren, wie in Abbildung 3.5 gezeigt. Dabei ist jedoch eine Erweiterung zu aufwändigeren Lösungen jederzeit möglich, z. B. durch eine erneute Spezialisierung eines Arbeitsknotens.

3.4 Automatisierung von Konfigurationsschritten

Im Abschnitt 3.2.1 wurde bereits erwähnt, dass es eine Eigenschaft von Arbeitsknoten ist, automatische Konfigurationsschritte vor Beginn einer Messung durchzuführen. Zusätzlich können auch andere Objekte des Frameworks diese Funktionalität benötigen, so z. B. die Messumgebung während ihrer Initialisierungsphase (vgl. auch Abschnitt 3.5.4).

Damit es für die verschiedenen Komponenten des Frameworks, z. B. für Arbeitsknoten, leichter ist, verschiedene Konfigurationsschritte einzuplanen und überwacht auszuführen, wird die Möglichkeit geschaffen, eine Konfigurator-Komponente einzubinden. Ein Konfigurator stellt eine geordnete Sammlung von Konfigurationsschritten dar. Diese sollen im Konfigurator abgelegt werden können, um hierüber eine eindeutige Ausführungsreihenfolge zu definieren.

Jeder Konfigurationsschritt soll über einen internen Status verfügen, der anzeigt, ob der Schritt noch nicht ausgeführt, erfolgreich ausgeführt, nicht erfolgreich ausgeführt (es sind Fehler entstanden) oder deaktiviert ist. Falls ein Konfigurationsschritt Fehler verursacht hat, sammelt er diese intern und setzt entsprechend den Ablaufstatus auf „nicht erfolgreich ausgeführt“. Jeder Schritt stellt eine Funktion zum Ausführen und

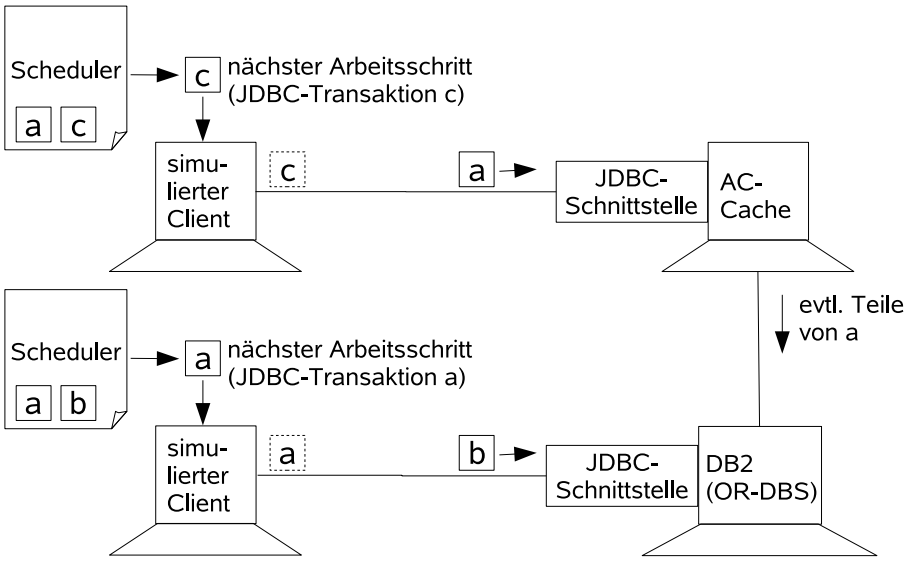

Abbildung 3.5: Eingeschränkte Eigenschaften (Scheduler + eine Verbindung).

Aufräumen bereit. Die Funktion Aufräumen wird dabei benutzt, um den gewünschten Zustand nach einer Messung herzustellen. So kann z. B. der Konfigurationsschritt *Datenbank starten* die Datenbank wieder stoppen, der Konfigurationsschritt *Schema anlegen* aber sollte das Schema nicht wieder löschen, um im Schema gesammelte Werte nicht zu vernichten. Der Entwickler muss also jeweils entscheiden, was er in die „Aufräumfunktion“ implementiert.

Somit wird es für den Konfigurator möglich, die verschiedenen Konfigurationschritte auszuführen, aufzuräumen, zu überwachen und gegebenenfalls zu wiederholen. Die Grundelemente des Konfigurators verdeutlicht nochmals Abbildung 3.6.

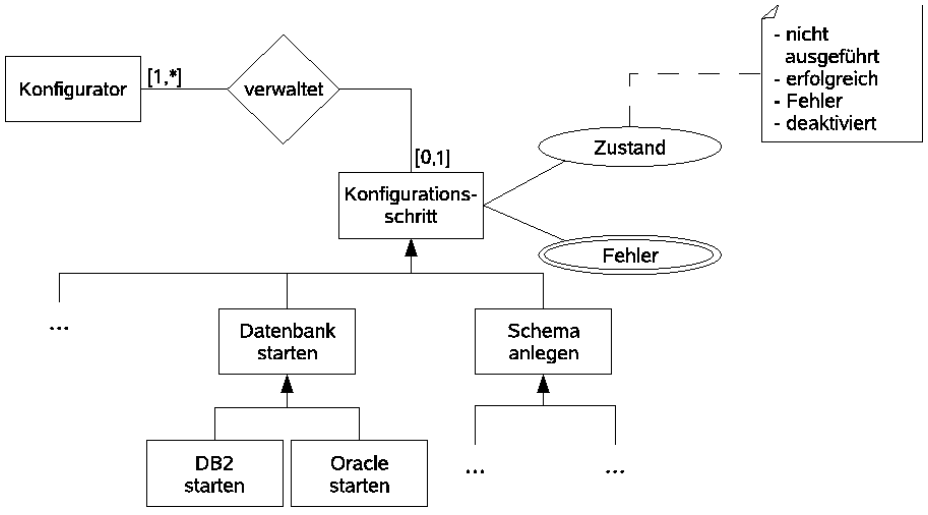

Abbildung 3.6: Die Grundelemente des Konfigurators.

3.5 Verwaltung von Messdaten

Jede Anwendung innerhalb eines verteilten Systems, die einen Dienst erbringt oder abfragt, kann für eine Messung wichtige Informationen liefern. Dabei kann die Art

und Weise, wie ein Messwert abzufragen ist, nicht durch das Framework vorbestimmt werden. Die Abfrage eines Messwertes muss immer über eine individuell angepasste Programmkomponente erfolgen, welche sich einer speziell für diesen Zweck zur Verfügung gestellten Schnittstelle bedient.

Die Art und Weise der Verwaltung bereits erfasster bzw. noch zu erfassender Messwerte kann jedoch durch das Framework festgelegt werden. Dabei sind durch das Framework folgende Teilaufgaben zu lösen:

1. Klassifikation erfassbarer Messwerte und ihrer Beziehungen
2. Übermittlung der Messwerte an die Messumgebung
3. Persistente Speicherung übermittelter Messwerte

In der nachfolgenden Abbildung (s. Abbildung 3.7) wird der grundlegende Ablauf einer Erfassung von Messwerten kurz dargestellt. Hierbei führt ein simulierter Benutzer eine vorher definierte Arbeitslast auf dem ACCache-System aus. Eine Beobachterkomponente ermittelt Messwerte und greift auf Funktionalitäten der Messwerterfassung zurück, um diesen an die Messumgebung zu senden. Dort wird der Messwert in einer Nachrichtenschlange entgegengenommen und abgespeichert.

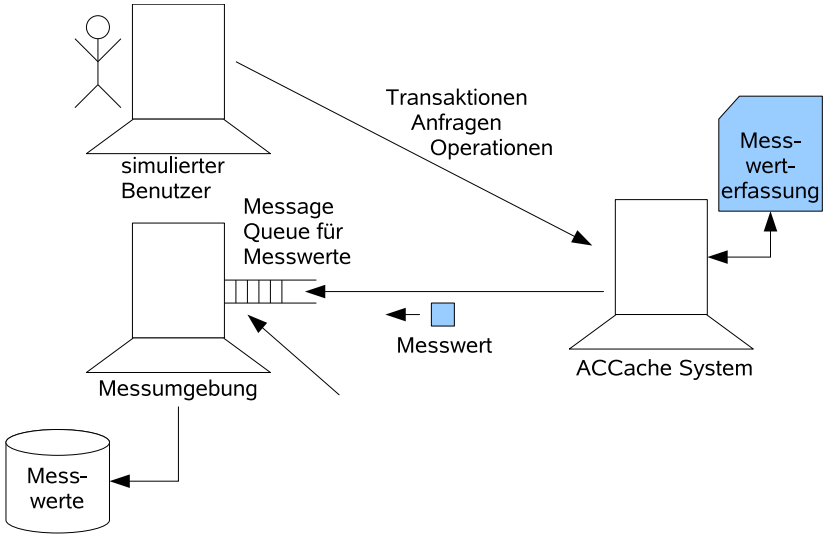

Abbildung 3.7: Grundablauf der Erfassung von Messwerten

3.5.1 Klassifikation erfassbarer Messwerte und ihrer Beziehungen

Für eine Teilanwendung, die auf einem Arbeitsknoten zu vermessen ist, kann die direkte Einflussnahme auf die Verarbeitungsinformationen stark unterschiedlich sein. Eine im Unternehmen selbstständig programmierte Anwendung kann man z. B. weitaus leichter um beliebige Analysefähigkeiten erweitern als eine extern programmierte Anwendung. Für eine extern programmierte Anwendung kann dies bedeuten, dass man nur ihr nach außen sichtbares Verhalten, nicht aber einzelne interne Komponenten

überwachen kann. In einigen Fällen sind Leistungsdaten aber auch über eine speziell zur Verfügung gestellte Schnittstelle abrufbar, wie es bei Datenbanksystemen häufig der Fall ist. Ein Entwickler kann für einen Dienst also jeweils nur diejenigen Messwerte erfassen, die er auch beobachten kann.

Darüber hinaus können auch Leistungsinformationen von Interesse sein, die nicht direkt vom erbrachten Dienst abgefragt werden können. Eine typische Größe ist z. B. die Prozessorauslastung, die man während eines Testlaufes auf einer Komponente eventuell periodisch abfragen will.

Es gibt somit viele verschiedene Wege, Messdaten abzufragen, die durch das Framework nicht vorgeschrieben werden können. Dies bedeutet, wie oben bereits erwähnt, dass das Abfragen oder Ermitteln der Leistungsdaten über eine selbst zu implementierte Programmkomponente erfolgen muss. Diese beobachtet die Teilanwendung und erfasst die anfallenden Messwerte.

Messergebnisse erhalten wie alle Daten aber erst dann ihre Bedeutung, wenn sie in Bezug auf einen Kontext interpretiert werden. So ist z. B. die Information, dass die Bearbeitungszeit 14 Stunden betrug, nutzlos, wenn sie nicht in Bezug zum abgelaufenen Batch-Job erfasst wurde. Für ein Messergebnis definieren sich die Kontextinformationen durch den aktuellen Ausführungskontext der Anwendung, der sich im Rahmen der gegebenen Abfragemöglichkeiten beliebig feingranular betrachten lässt. Falls z. B. für eine Datenbank erfasst werden soll, wie viele Seitenzugriffe getätigt wurden, dann kann dieser Wert in Bezug auf eine komplette Transaktion, eine Anfrage, eine Teilanfrage, ein Segment, für eine bestimmte Tablelle oder auch für eine Anfrage innerhalb einer Transaktion betrachtet werden.

Für die Verwaltung und Definition geeigneter Ausführungskontexte muss das zu entwickelnde Framework einen Rahmen bilden.

Der Kontext eines Messergebnisses

Als Beispiel für den Kontext eines Messergebnisses soll uns die Durchführung einer Transaktion in einem Datenbanksystem dienen. Im Bezug auf den Kontext einer Transaktion lassen sich Leistungsdaten erfassen, wie z. B. die Bearbeitungszeit der Transaktion. Jedoch werden auch innerhalb einer Transaktion mehrere Anfragen abgewickelt. Für diese lassen sich wiederum Leistungswerte ermitteln. Die Transaktion und die Anfrage als Kontexte für die Ermittlung von Messergebnissen stehen hierbei aber auch untereinander in Beziehung. Die Transaktion beinhaltet die Anfrage. Oder anders ausgedrückt: Die Transaktion war ursächlich für die Durchführung der Anfrage. Ähnlich verhält es sich bei Prozessen und deren Unterprozessen.

Ziel ist es somit, eine Möglichkeit zu schaffen, auf jedem Knoten des verteilten Systems Ausführungskontexte auszuzeichnen, welche die Baumstruktur des Programmablaufes für eine Messung bestmöglich wiedergeben. Somit besteht die Notwendigkeit, unter einem aktuellen Ausführungskontext weitere anzulegen, für welche der Vaterkontext ursächlich war. Messwerte sind dabei immer in Bezug auf einen Ausführungskontext zu erfassen.

Wie die Beziehungen zwischen den Ausführungskontexten eines Knotens des verteilten Systems aussehen können, verdeutlicht das ER-Diagramm in Abbildung 3.8.

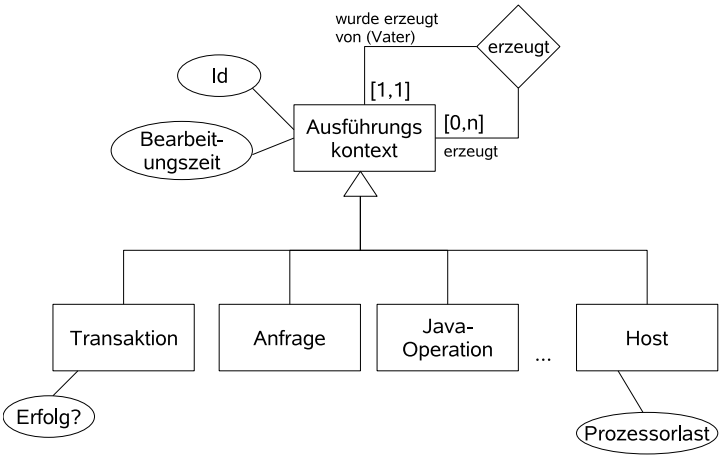

Abbildung 3.8: Ausführungskontexte

Das Framework definiert die Schnittstelle eines Ausführungskontextes, so dass für eine Teilanwendung des Systems die Möglichkeit besteht, spezielle Ausführungskontexte zu definieren.

Eine besondere Bedeutung kommt dem initialen Ausführungskontext zu. Er beschreibt den Start- und Endzeitpunkt einer Messung und bildet somit die Klammer für alle während der Messung erfassten Messwerte. Durch Übermittlung des initial zu verwendenden Kontextes an die Beobachter einer Teilanwendung werden diese in der Lage versetzt, Messwerte zu erfassen. Somit wird durch das Verschicken des initialen Kontextes die Messung gestartet.

Für jeden Ausführungskontext lassen sich einzelne Messwerte und auch periodisch ermittelte Messwerte erfassen. Periodisch ermittelte Messwerte werden in der Folge als mehrwertige Attribute gekennzeichnet (s. Abbildung 3.12). Dabei können Messwerte nur solange in einem Ausführungskontext erfasst werden, wie dieser *geöffnet* ist. Der initiale Kontext ist während der ganzen Zeit eines Messablaufes offen. Seine Schließung bedeutet das Ende der Messung und muss von der Messumgebung signalisiert werden. Um z. B. die Prozessorauslastung alle 3 Sekunden während der kompletten Messung für eine Teilanwendung abzufragen, wird diese im initialen Kontext als periodisch erfasster Wert modelliert.

Ein Beobachter, der eine Instanz eines Ausführungskontextes besitzt (z. B. den initialen Kontext), kann über diesen weitere Unterkontexte bilden. Welche er dabei anlegen darf, muss für den Arbeitsknoten, auf dem der Beobachter eingesetzt ist, vorab festgelegt werden. In Abbildung 3.8 lassen sich unter dem initialen Kontext z. B. eine Transaktion und darunter eine Anfrage als Unterkontexte bilden.

Es lassen sich darüber hinaus auch Messwerte identifizieren, die für fast alle Kontexte gleichermaßen erfassbar sind. So ist zum Beispiel die Bearbeitungszeit stets ermittelbar, indem die Zeit, bevor die Aktion gestartet wird, und die Zeit nach Beendigung einer Aktion jeweils gemessen wird, so dass sich die Bearbeitungszeit ergibt durch:

$$Bearbeitungszeit = Startzeit - Endzeit$$

Das Instanzendiagramm in Abbildung 3.9 zeigt einen Baum von Ausführungskontexten, wie er während der Messung auf einem Knoten entsteht. In ihm werden die Messwerte gesammelt, wodurch die Einordnung und Bedeutung des Messwertes festgelegt ist.

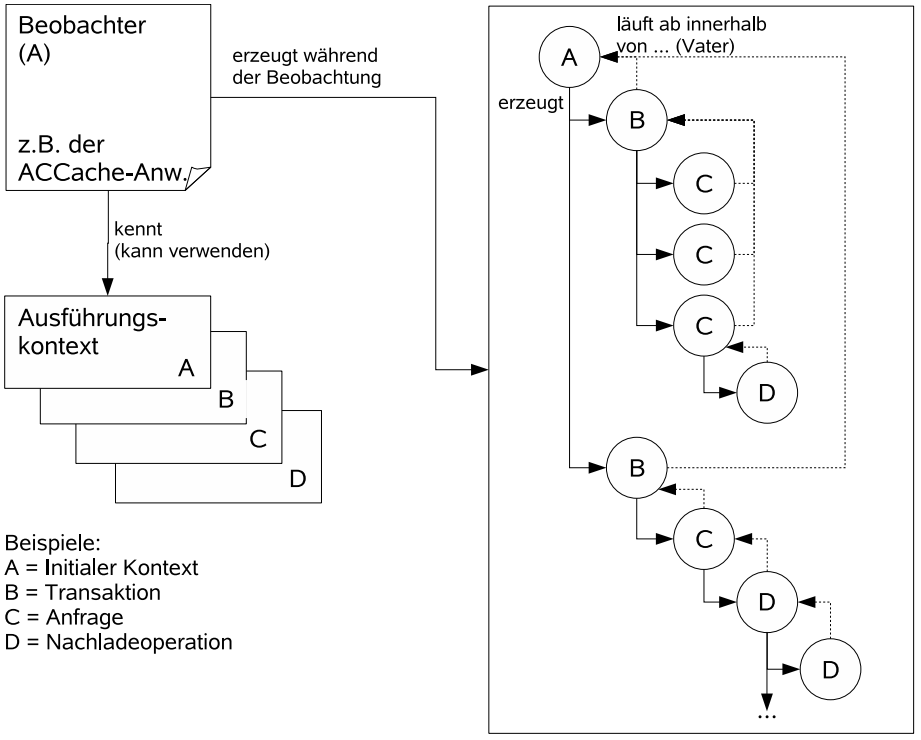

Abbildung 3.9: Beispiel für den generierten Instanzenbaum eines Beobachters.

Wie bereits in Abschnitt 3.2.1 erwähnt können auch Beziehungen zwischen den Messwerten verschiedener Arbeitsknoten bestehen.

Beziehungen zwischen Ausführungskontexten verschiedener Arbeitsknoten

Durch die innerhalb eines verteilten Systems stattfindenden Aufrufe zwischen den Teilanwendungen bestehen zwischen den entsprechenden Client- und Server-Anwendungen zusätzliche Beziehungen. Wie bereits im Beispiel aus Abschnitt 3.1 ersichtlich, ist es wichtig, auch diese Beziehungen erfassen zu können.

Wird eine Anwendung mit Hilfe des Frameworks beobachtet, können Messwerte nur mit Hilfe eines Ausführungskontextes erfasst werden. Beispielsweise kann eine Client-Anwendung nur dann die Bearbeitungszeit einer Transaktion messen, wenn sie einen Ausführungskontext angelegt hat, der die Transaktion repräsentiert. Innerhalb der Transaktion laufen Anfragen ab, die durch einen Aufruf an die Datenbank beantwortet werden. Erfasst der Client auch die Bearbeitungszeit jeder dieser Anfragen auf dem Client, so muss er auch für jede Anfrage innerhalb der Transaktion einen Ausführungskontext anlegen.

Jedoch laufen die Transaktion und ihre Anfragen auf dem Datenbanksystem ab. Werden diese auch auf dem Datenbankserver beobachtet, sollte es eine Möglichkeit

geben, den Ausführungskontext *Transaktion* auf dem Client mit dem entsprechenden des Servers in Beziehung zu setzen.

Die beiden Ausführungskontexte dürfen hierbei jedoch nicht gleich gesetzt werden. Wenn beide Kontexte die Bearbeitungszeit messen, erfasst der Client die Bearbeitungszeit der Transaktion auf dem Client und der Server die Bearbeitungszeit der Transaktion auf dem Server.

Hierbei wird auch die Bedeutung der Beziehung zwischen den beiden Kontexten ersichtlich. Falls diese ausgewertet werden kann, lässt sich ohne explizite Messungen ermitteln, wie lange die reine Bearbeitungszeit durch den Client war, inklusive der notwendigen Übertragungszeit der Daten über das Netzwerk. Steht sogar die reine Rechenzeit des Client-Prozesses für diese Transaktion zur Verfügung, könnte die Übertragungszeit der Daten rechnerisch ermittelt werden. In Abbildung 3.10 wird dieser Zusammengang verdeutlicht.

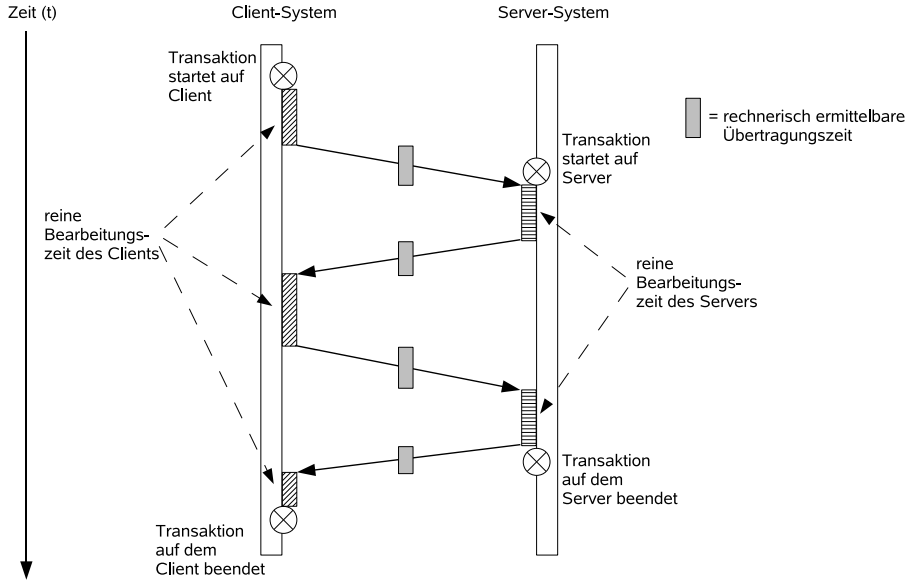

Abbildung 3.10: Beispiel für rechnisch ermittelbare Übertragungszeit bei bekannter Aufrufbeziehung

Darüber hinaus lässt sich ein Client-Kontext mit einem Server-Kontext nur dann in Verbindung bringen, wenn die Aufrufbeziehungen zwischen verschiedenen Arbeitsknoten abbildbar sind.

Abbildung 3.11 zeigt die durch jeden Aufruf neu entstehenden Beziehungen zwischen Ausführungskontexten. Hierbei wird innerhalb des Ausführungskontextes von D (einer Nachladeoperation) auf dem Client (ACCache) ein Aufruf an den Server (DB2-Server) abgeschickt. Auf dem DB2-Server ist daraufhin die Bearbeitung einer Transaktion B beobachtbar, die aufgrund des Client-Aufrufes gestartet wurde. Kontext $ACCache.D$ ist also ursächlich für Kontext $DB2.B$.

Ein Client-Ausführungskontext kann, z. B. durch mehrere Aufrufe, mit n Server-Ausführungskontexten in Verbindung stehen. Ein Server-Ausführungskontext kann immer nur mit einem Client-Ausführungskontext assoziiert sein, da für einen Server-Ausführungskontext nicht mehrere Clients ursächlich gewesen sein können.

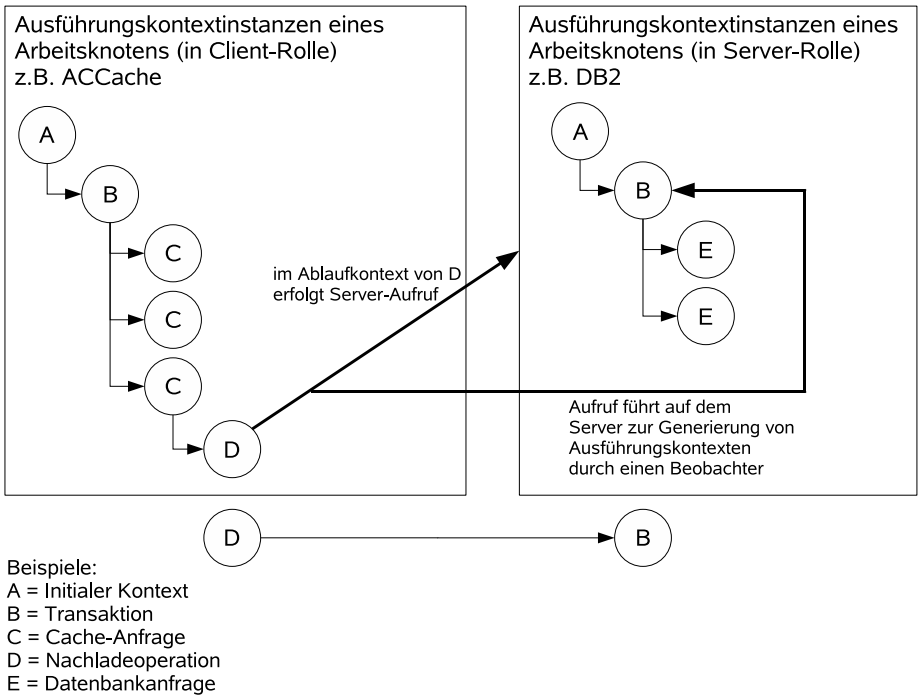

Abbildung 3.11: Beispiel für eine Beziehung zwischen Ausführungskontexten verschiedener Arbeitsknoten.

Doch wie lassen sich die Beziehungen zwischen den Ausführungskontexten erfassen? Eine eindeutige Identifikation ist möglich, wenn mit dem Client-Aufruf eine eindeutige Kontextidentifikation mitgeschickt wird. Diese kann dann von einer Beobachterkomponente auf dem Server in einem neu angelegten Server-Ausführungskontext vermerkt werden. Die nachfolgend stattfindende Übermittlung der Kontextinformationen an die Messumgebung (vgl. Abbildung 3.5.3) erlaubt es, die Beziehung auszuwerten. Eine entsprechende Zuordnung ist aber nur dann denkbar, wenn die Aufrufschnittstelle einer Teilanwendung dahingehend geändert werden kann.

Ist dies nicht möglich, kann der Entwickler sich bemühen, die Zuordnungen so gut wie möglich durch eigene Funktionalitäten oder Annahmen über die Zugehörigkeit abzubilden. Im folgenden wird ein Beispiel konstruiert, in dem die Zuordnung auf anderem Wege bewerkstelligt wird.

Falls auf dem Server beobachtet werden kann, von welchem Arbeitsknoten (Teilanwendung) ein Aufruf kam, ist damit der korrekte Ausführungskontext des Clients noch nicht identifiziert. Eine erste Zuordnung ist damit aber schon gewährleistet: Der aufgrund eines Aufrufs vom Server angelegte Ausführungskontext kann mit dem Arbeitsknoten assoziiert werden. Ebenso kann der Client vor dem Aufruf in seinem derzeitigen Ausführungskontext vermerken, welchen Arbeitsknoten er aufrufen wird. Nach der Übermittlung der Kontextinformationen an die Messumgebung, lässt sich eine korrekte Zuordnung vornehmen, wenn die Erzeugungsreihenfolge der Kontexte auf dem Arbeitsknoten bekannt ist (s. Abbildung 3.12).

Ein Problem bei dieser Methode besteht nur dann, wenn sich Aufrufe einer Client-Anwendung an dieselbe Server-Anwendung gegenseitig überholen können.

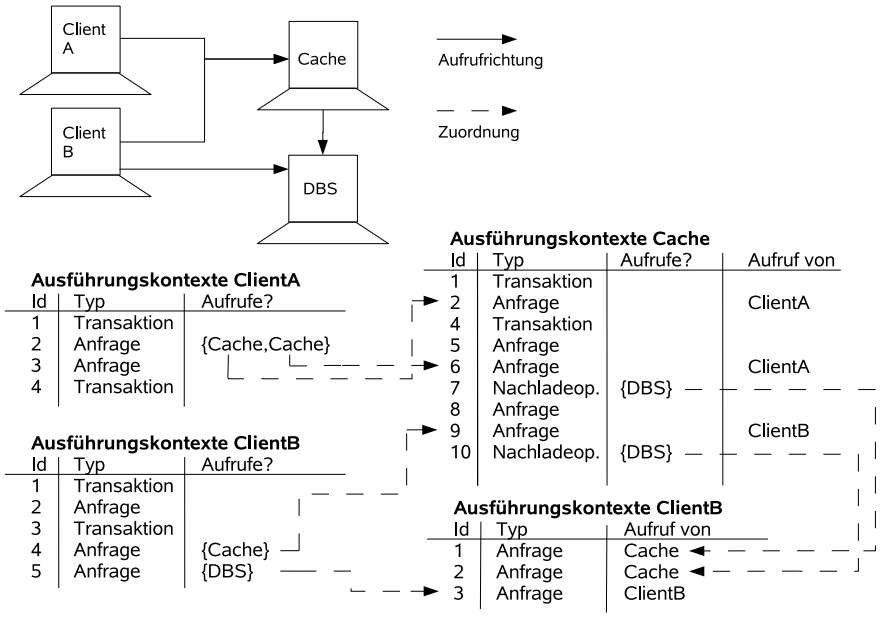

Abbildung 3.12: Zuordnungsbeispiel Ausführungskontexte - indirekte Zuordnung.

Falls eine Server-Anwendung nur über eine Verbindung aufgerufen wird oder eine Client-Anwendung nur über eine Verbindung Aufrufe absetzen kann, lässt sich die Information, von welchem Arbeitsknoten der Aufruf kam bzw. wer Aufrufer war, sogar weglassen. Es reicht der Vermerk, dass ein Aufruf stattfand. Client *A* in Abbildung 3.12 kann z. B. nur den Cache aufrufen, sodass statt der geordneten Liste mit der Auszeichnung {*Cache, Cache*} auch der Vermerk zweier Aufrufe genügen würde.

Als weitere darüber hinausgehende Möglichkeit lassen sich Verbindungen zwischen Kontexten evtl. nach der Messdatenerfassung anhand semantischer Bedingungen vornehmen, z. B. durch Zeitstempelangaben über die Erzeugung von Kontexten. Dies würde dann aber einen speziellen Auswertungsschritt bedeuten, der nicht vom Framework automatisch vorgenommen werden kann.

3.5.2 Wichtige Messgrößen und Ausführungskontexte beim Constraint-basierte Datenbank-Caching

Um Messungen für das Constraint-basierte Datenbank-Caching vornehmen zu können, müssen für das Framework Ausführungskontexte und die durch sie erfassbaren Messwerte für die verschiedenen Arbeitsknoten ausgezeichnet werden. Damit es besser gelingt, die notwendigen Informationen zusammenzutragen, betrachten wir zunächst nochmals die typische Zugriffs- und Netzstruktur für Systeme mit Constraint-basiertem DB-Cache.

Es gibt drei unterschiedliche Komponenten, auf denen Messdaten erfasst werden können:

- den Client-Rechner
- den Constraint-basierten Datenbank-Cache und

- das zentrale Datenbank-System

Der Client-Rechner oder auch zu simulierende Benutzer führt dabei als Arbeitsschritte Transaktionen durch, die wiederum Anfragen enthalten, die durch den Constrainbasierten Datenbank-Cache beantwortet werden. Da das Cache-System nur zur Leistungssteigerung dient, können die Transaktionen bzw. Anfragen alternativ auch direkt durch das Backend ohne Cache beantwortet werden.

Ganz gleich, ob mit oder ohne Cache, lassen sich auf dem Client also stets die gleichen Messwerte erfassen. Die Durchführung der gleichen Messkonfiguration einmal mit und einmal ohne Cache-System lässt also schon Rückschlüsse auf die Leistungsfähigkeit des Systems zu, ohne Messergebnisse auf anderen Komponenten zu betrachten. Sie sind allerdings nicht dazu in der Lage, die Schwachstellen des Systems aufzudecken.

Typische Leistungswerte, die auf dem Client ermittelt werden können, sind die bereits durch einschlägige Datenbank-Benchmarks bekannten Messgrößen, wie z. B. :

- die Anzahl durchgeführter Transaktionen pro Sekunde
- die Anzahl erfolgreicher, die Anzahl zurückgesetzter Transaktionen
- die mittlere Bearbeitungszeit für den jeweiligen Transaktionstyp

Darüber hinaus sind auch denkbar:

- pro Transaktion die Bearbeitungszeit jeder einzelnen Anfrage
- pro Anfrage die Anzahl zurückgelieferter/geänderter Tupel
- die mittlere Prozessorauslastung oder
- die reine Rechenzeit auf dem Client

Um die Messgrößen einfach und überschaubar zu halten, bietet es sich an, im Rahmen der Diplomarbeit nur Messgrößen für den Ausführungskontext einer Transaktion zu betrachten. Das Verhalten des Caches im Bezug auf eine einzelne Anfrage zu analysieren, ist durch die Betrachtung einer Transaktion mit nur einer Anfrage immer noch möglich.

Somit wird für den Client die Auszeichnung eines einzelnen Ausführungskontextes *Transaktion* vorgenommen. Da durch die Auszeichnung dieses Kontextes nun jede durchgeführte Transaktion, zusammen mit ihrer Bearbeitungszeit usw., erfasst werden kann, ist implizit die Anzahl durchgeführter Transaktion bekannt sowie die Anzahl durchgeführter Transaktionen pro Sekunde ermittelbar.

Durch die auf dem Client erfassbaren Messwerte lässt sich schon recht gut die Gesamtleistung des Systems beurteilen. Um auch Rückschlüsse auf die Ursachen positiver wie negativer Leistungswerte zu erhalten, müssen auch Messdaten auf dem im Fokus stehenden Cache-System erhoben werden. Hier stehen vor allem die in Kapitel 2 beschriebenen Verarbeitungsverfahren, wie z. B. die Sondierung oder das Füllen des Caches, zusätzlich unter Beobachtung.

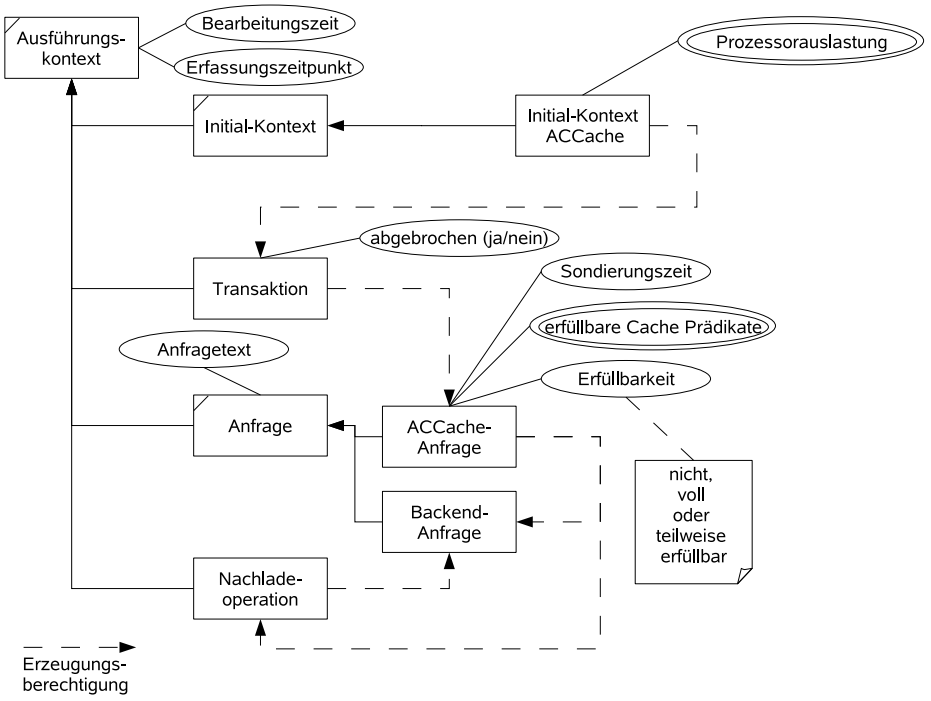

Abbildung 3.13: Ausführungskontexte für den ACCache.

Um die in Kapitel 2 schon beschriebenen Messwerte erfassen zu können, sind auch hier Ausführungskontexte zu beschreiben, welche die Messdaten bestmöglich erklären. Abbildung 3.13 zeigt die für den ACCache entwickelten Kontexte.

Für das DB2-Datenbanksystem werden keine Ausführungskontexte entwickelt, da die Implementierung dieser Teilanwendung nicht beeinflussbar ist und sie für die Leistungsfähigkeit des ACCache-Systems nur eine untergeordnete Rolle spielt. Die Erfassung von Messdaten, z. B. über die Schnittstelle des Event-Monitors des Datenbanksystems, wäre aber durchaus möglich.

3.5.3 Übermittlung von Messwerten an die Messumgebung

Um Messwerte an die Messumgebung übermitteln zu können, müssen nicht nur die Daten sondern auch die Kontextinformationen übermittelt werden. Hierbei ist es ausreichend, einen Kontext eindeutig identifizierbar zu machen und eine Beschreibung der in ihm erfassbaren Messwert-Attribute zur Messumgebung zu übermitteln.

Eine Beschreibung des Ausführungskontextes muss der Messumgebung nur einmal bekannt gemacht werden. Dies kann entweder gelöst werden, indem vor Beginn der Messung die Beschreibungen übermittelt werden oder indem diese explizit durch die Messumgebung abrufbar sind bzw. dieser bereits bekannt gemacht wurden. Die Beschreibung des Kontextes definiert somit einen neuen Ausführungskontexttyp.

Ausführungskontexttyp

Um der Messumgebung einen neuen Ausführungskontext bekannt zu machen, benötigt man eine Beschreibung der in ihm erfassbaren Messwerte. Diese Beschreibung zusammen mit der Vergabe eines Namens definiert einen neuen Ausführungskontexttyp. Dabei sind die in Abschnitt 3.5.1 beschriebenen Eigenschaften abzubilden.

Die Beschreibung umfasst für jeden Arbeitsknoten (Vorsicht: aber nicht für jede Instanz):

- den Namen des Ausführungskontextes
- die anlegbaren Unterkontexte und
- die Definition der erfassbaren Messwerte (Einzelwerte und periodische Werte)

Aus der Beschreibung können durch einen entsprechenden Generator die Java-Klassen der Ausführungskontexte generiert werden. Dies verhindert, dass der Entwickler eines Beobachters generische Methoden aufrufen muss, die bei Falschanwendung eventuell Ausnahmen produzieren. So kann z. B. durch das Generieren jeweils eine eigene Methode zum Anlegen eines bestimmten Unterkontextes zur Verfügung gestellt werden, so dass der Versuch, einen nicht berechtigten Unterkontext anzulegen, ausgeschlossen ist.

Übertragung der Kontextinformation

Sobald für eine Teilanwendung durch einen Beobachter eine Ausführungskontextinstanz angelegt wird, wird dieser eine eindeutige Identifikation (z. B. fortlaufend ansteigender Integer-Wert) zugewiesen und der Messumgebung bekannt gemacht. Ist der angelegte Kontext innerhalb eines Vaterkontextes angelegt worden, so wird die Identifikation des Vaterkontextes zusätzlich mitgeliefert, so dass eine entsprechende Zuordnung möglich wird. Nachdem eine Instanz eines Kontextes angelegt und übermittelt wurde, ist diese als *offen* gekennzeichnet. Die Messumgebung geht davon aus, dass für diesen Kontext nun Messwerte übermittelt werden. Ein Beobachter ist in der Lage, für einen Kontext das Erfassen der Messdaten zu beenden, indem er den Ausführungskontext schließt. Dabei wird eine weitere Nachricht an die Messumgebung gesendet, die das Schließen des Kontextes signalisiert. Diese wird aber erst dann versendet, wenn auch für alle Unterkontexte gilt, dass sie geschlossen sind.

Übertragung von Messwerten

Solange ein Ausführungskontext den Status *geöffnet* besitzt, können Messwerte für diesen erfasst werden. Die Erfassung erfolgt über einen Methodenaufruf, der entweder durch einen Entwickler für jeden Messwert einzuführen ist oder der anhand der Beschreibung generiert wird. Die Methode versendet mit Hilfe einer generischen *Versende*-Funktion den Messwert. Dazu wird ein generisches Nachrichtenformat verwendet.

Nachrichtenformate

Für die Übermittlung von Messdaten sind folgende Nachrichten zu unterscheiden:

- das Anlegen (= Öffnen) einer Kontextinstanz
- das Schließen eines Kontextinstanz
- das Versenden eines periodisch erfassten Messwertes und

- das Versenden eines einzeln erfassten Messwertes

Da der Messumgebung die Beschreibungen der Ausführungskontexte bekannt sein müssen, sind nur noch Informationen zu versenden, die sich stets ändern. Die Messwerte jedes Arbeitsknotens werden darüber hinaus nur über eine eigens zur Verfügung gestellten Nachrichtenschlange erfasst, was bedeutet, dass keine Identifikation für den Arbeitsknoten übermittelt werden muss. Somit müssen für das Anlegen einer Kontextinstanz folgende Werte übermittelt werden:

- der Name des Ausführungskontexttyps
- die eindeutige Identifikation des Ausführungskontextes und
- optional die eindeutige Identifikation des Vaterkontextes

Diese Werte werden auch beim Schließen des Kontextes übermittelt, wobei die Angabe des Vaterkontextes nicht notwendig ist.

Für die Übermittlung eines Messergebnisses sind folgende Werte zu übermitteln:

- die eindeutige Identifikation des Ausführungskontextes
- der Name des Messwertes
- der Index des Messwertes und
- der Wert des Messergebnisses

Um einen periodisch erfassten Wert zu versenden, wird ein stets unterschiedlicher Index verwendet, der die Reihenfolge der erfassten Einzelwerte kennzeichnet. Ein einzeln zu erfassender Messwert wird immer mit dem Indexwert Null verschickt. Dabei muss nicht übermittelt werden, ob ein versendeter Messwert ein periodischer oder ein einzeln erfasster Messwert ist. Dies kann anhand des Namens und der vorliegenden Beschreibung überprüft werden. Der Wert des Messwergebnisses muss jedoch in einer serialisierten Form, z. B. per Zeichenkette, übertragen werden. Der Typ des Messwertes wird anhand der Beschreibung bestimmt.

3.5.4 Speicherung übermittelter Messdaten

Durch die eingeführte Beschreibung übermittelbarer Kontexte und ihrer Daten sowie durch das generische Nachrichtenformat für die Übermittlung von Messwerten ist die Ablage der Daten unabhängig von der Art der Übermittlung von Mess- und Leistungsgrößen.

Die Art der Speicherung der Messwerte sollte jedoch so erfolgen, dass nachfolgende Auswertungen leicht durchgeführt werden können. Aus diesem Grund wurde entschieden, die Daten in eine objekt-relationale Datenbank zu speichern, mit deren Hilfe Auswertungen leicht möglich sind. Darüber hinaus wird hierdurch die Flexibilität, die Daten in andere Formate übertragbar zu machen, deutlich erhöht.

Das Framework sollte in der Lage sein, dem Benutzer die Wahl darüber zu lassen, wie und in welchem Format er die Daten abspeichern will, indem er die Standardimplementierung überschreibt. Jedoch sollte für jeden definierbaren Ausführungskontexttyp eine generische Speicherung der Daten möglich sein.

Generische Speicherung übermittelter Messdaten

Bei der Speicherung der Messdaten spielen nicht nur die übermittelten Mess- und Kontextinformationen eine Rolle. Damit Auswertungen im Nachhinein besser durchführbar sind, sollten auch Informationen über den Aufbau des verteilten Netzes (insbesondere die Arbeitsknoten und ihre Verbindungen untereinander) mit abgespeichert werden. Außerdem ist darauf zu achten, dass mehrere Messungen, die eventuell nacheinander durchgeführt werden, zusammen abspeicherbar sind.

Damit die Sicherung der Messwerte nicht zu einer unnötigen Satzexplosion führt, wird für jeden Ausführungskontexttyp eine eigene Tabelle aus der Beschreibung des Kontextes generiert. Hierbei bildet jede neue Instanz einen neuen Satz in der entsprechenden Tabelle. Die einzelnen Messwerte werden in den dazu generierten Spalten abgelegt. Für periodisch zu erfassende Werte wird eine Detailtabelle erzeugt, so dass die Ablage mehrerer Werte pro Ausführungskontexttypinstanz gewährleistet wird.

Abbildung 3.14 zeigt ER-Modellierung des generischen Schemas. Dabei ist darauf zu achten, dass pro Ausführungskontexttyp neue generische Entitäten zu bilden sind. Somit unterteilt sich das Schema in eine statischen Teil und einen dynamischen Teil. Der statische Teil beeinhaltet alle Entitäten, die nur ein Mal für alle nachfolgend durchgeführten Messabläufe, angelegt werden müssen. Die Entitäten *Messwerte_<Kontext-Id>* und *periodische Messwerte_<Kontext-Id>_<PeriodicValueName>* sind erst während eines konkreten Messablaufes eindeutig bestimmt und verändern sich dynamisch je nach durchgeführter Messkonfiguration.

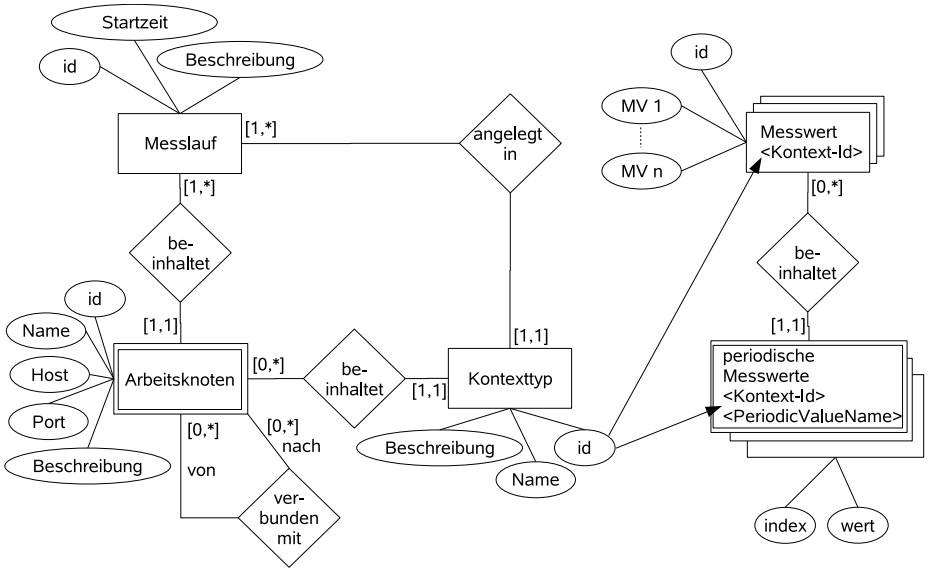

Abbildung 3.14: Das dynamsch erweiterbare Datenbankschema.

3.6 Unterstützung für Auswertungen

Die Speicherung der Daten in einem relationalen Datenbankschema, wie sie im vorangegangen Abschnitt beschrieben ist, bietet bereits sehr viele Möglichkeiten, die gewonnenen Daten weiterzuverarbeiten. Aus diesem Grund wird in dieser Arbeit davon Abstand genommen, ergänzende Konzepte zu entwickeln.

Während des praktischen Einsatzes können durchaus noch Erweiterungen notwendig werden, die dann aber je nach Bedarf zu implementieren sind.

4. Das Framework zur Vermessung von verteilten Systemen

In diesem Kapitel wird die Implementierung der Ideen und Konzepte aus Kapitel 3 beschrieben, die zur Erstellung eines Frameworks für die Vermessung verteilter Systeme entwickelt wurden.

Dabei werden, ähnlich wie auch schon in Kapitel 3, die Implementierungen anhand der in Abschnitt 3.1 identifizierten Aufgabengebiete strukturiert.

Die Programmiersprache *Java* wurde in der Version 1.5 eingesetzt. Um ein späteres Arbeiten mit dem Framework zu erleichtern, wird zunächst die Paketstruktur vorgestellt und erklärt. Abbildung 4.1 zeigt die Grundstruktur der Paketaufteilung.

Das Wurzelpaket `Measurement Manager (mm)` ist in zwei Bereiche unterteilt. Der erste Bereich umfasst die Pakete `initsteps`, `connection` usw., wie im unteren Teil der Abbildung 4.1 dargestellt. Sie enthalten die Funktionalität der Messumgebung, die nach dem Model-View-Controller-Entwurfsmuster implementiert ist.

Eine View wurde jedoch noch nicht entwickelt. Sie hätte primär die Aufgabe, die Konfiguration nicht über Konfigurationsdateien, sondern in einer angemessenen graphischen Oberfläche vorzunehmen, da die Messung automatisch geschieht. Somit ist die Realisierung weitgehend unabhängig von einer View, deren Implementierung einen zu großen Zusatzaufwand bedeutet hätte.

Als Modell werden die Klassen und Pakete des zweiten Bereiches `(self)` verwendet. Diese definieren die Komponenten des Frameworks und deren Funktionalität, welche durch die Implementierung von Klassen für konkret zu überwachende Systeme erweitert werden kann. Die notwendigen Erweiterungen zur Überwachung des ACCache-Systems werden in Kapitel 5 besprochen und sind in der Paketstruktur `accache` implementiert.

Die Pakete der Framework-Komponenten

Die Klassen des Paketes `self` bilden die Grundstruktur der Modellklassen, welche in Abschnitt 4.1 erklärt werden. Alle weiteren Pakete und Klassen, die unter dem Paket

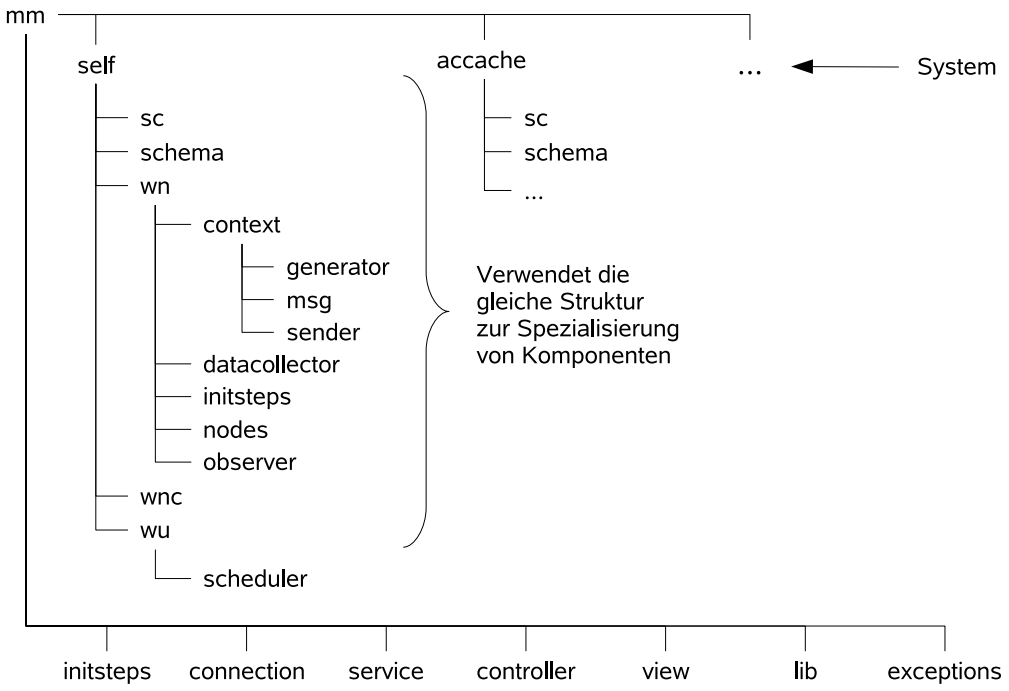

Abbildung 4.1: Grundstruktur der Paketaufteilung.

self angesiedelt sind, beschreiben Informationen und Funktionalitäten, die während des Messablaufes benötigt werden.

Wie bereits in Kapitel 3 beschrieben, ist es das Hauptziel, die Eigenschaften eines Arbeitsknotens abzubilden. Diese Aufgabe wird primär durch das Paket `mm.self.wn` und dessen Unterpakete geleistet. Im Paket `mm.self.wn.nodes` sind die Stammklassen zur Realisierung eines Arbeitsknotens (vgl. Abschnitt 4.1.1) und eines simulierten Client-Arbeitsknotens (vgl. Abschnitt 4.2) implementiert. Ein simulierter Client-Arbeitsknoten benötigt zusätzliche Arbeitschritte, die er ausführen kann, sowie einen Scheduler (vgl. Abschnitt 3.3). Da Arbeitsknoten und Arbeitsschritte die Grundelemente für die Vermessung eines Systems darstellen (Repräsentierung der Anwendungen und Definition der Anfragelast), sind die Pakete `mm.self.wn` und `mm.self.wu` gleichrangig angeordnet. Eine Anordnung innerhalb des Paket `mm.self.wn` wäre auch gerechtfertigt gewesen, auch wenn nicht jeder Arbeitsknoten Arbeitsschritte ausführen kann. Eine sehr tiefe Schachtelung wie z. B. `mm.self.wn.nodes.client.wu` sollte jedoch vermieden werden. Das Scheduling der Arbeitschritte wird durch das Paket `mm.self.wu.scheduler` vorgenommen. Die entsprechenden Implementierungen betrachtet Abschnitt 4.2.1.

Um die Kontextinformationen der Aufrufschnittstelle eines Arbeitsknotens beschreiben zu können, wurde das Paket `mm.self.wn.sc (ServiceContext)` entwickelt. Es implementiert die Eigenschaft der Aufrufbarkeit für einen Arbeitsknoten (vgl. Abschnitt 3.2.1, Aufrufbarkeit). Die Verbindungen zwischen den Arbeitsknoten sind im Paket `mm.self.wn.wnc` spezifiziert (vgl. Abschnitt 3.2.1, Beziehungen zwischen Arbeitsknoten). Beide Implementierungen werden in Abschnitt 4.1.1 vorgestellt.

Das Paket `initsteps`, welches beiden Paketbereichen angehört, kapselt die Funktionalität zur Durchführung von Konfigurationsschritten (vgl. Abschnitt 3.2.1, au-

tom. Konfiguration). Da diese Funktionalität ein übergreifendes Konzept darstellt, wurden die Grundkonzepte im Paket `mm.initsteps` implementiert. Für die Arbeitsknoten, die diese Funktionalität ebenfalls benötigen, wurden im Paket `mm.self.wn.initsteps` weitere Spezialisierungen vorgenommen. Die Funktionalität für die Durchführung von Konfigurationsschritten wird in Abschnitt 4.3 erläutert.

Im Paket `mm.self.wn.observer` ist die Beobachterkomponente (vgl. Abschnitt 3.2.1, Beobachtbarkeit) implementiert. Durch sie wird es möglich, einem Arbeitsknoten Beobachter zuzuweisen. Beobachter erfassen die Messdaten mit Hilfe von Ausführungskontexten, die dem Arbeitsknoten zugewiesen wurden. Aus diesem Grund enthält das Paket `mm.self.wn.context` die Schnittstelle für Ausführungskontexte. Konkrete Ausführungskontexte können über eine in der Konfiguration gegebene Beschreibung (vgl. auch Abschnitt A.2) generiert werden. Wie Ausführungskontexte definiert sind und wie die Generierung funktioniert, wird in Abschnitt 4.4.2 diskutiert. Die Generierung wird durch die Klassen des Pakets `mm.self.wn.context.generator` realisiert.

Damit die durch Ausführungskontexte erfassten Daten auf einheitliche Art und Weise zur Messumgebung transportiert werden können, müssen ein Messdatensender, ein Messdatenempfänger und die benötigten Nachrichtenformate definiert sein. Diese Komponenten werden in den Paketen `mm.self.wn.context.sender` (Messdatensender), `mm.self.wn.context.datacollector` (Messdatenempfänger) und `mm.self.wn.context.msg` implementiert und in Abschnitt 4.4.3 erklärt.

Ein eigenes Konzept stellt das Paket `mm.self.schema` dar. Es stellt eine Schnittstelle zur Verfügung, um dem Framework Datenschemata bekannt zu machen. Es ist die Möglichkeit vorgesehen, Daten für ein Schema spezifizieren zu können und diese durch einen Validator überprüfen zu lassen. Die Auszeichnung eines Schemas kann dem Framework helfen, Integritätsbedingungen sicherzustellen und zusätzliche Funktionalitäten für die Schemaverwaltung bereit zu stellen. So ist z. B. nicht jeder Arbeitsschritt, der über eine JDBC-Schnittstelle ausführbar ist, auf jedem Arbeitsknoten durchführbar. Ein während der Konfigurationsphase auf einem Arbeitsknoten angelegtes Datenbankschema muss z. B. mit dem in einem Arbeitsschritt angegebenen Schema übereinstimmen. Diese Funktionalitäten sind jedoch noch nicht vollständig implementiert und für die Durchführbarkeit von Messungen nicht zwingend notwendig. Sie werden jedoch verwendet, um das sich selbst erweiternde Datenschema (vgl. Abschnitt 3.5.4, Generische Speicherung von Messdaten) zu implementieren. Die Implementierung wird in Abschnitt 4.4.4 vorgestellt.

Zusätzliche Pakete der Messumgebung

Nach der Definition der Modellklassen benötigt die Messumgebung noch Controller- und View-Komponenten, um die Implementierung zu komplettieren. Die Controller-Klassen sind im Paket `mm.controller` untergebracht. Sie definieren die aufrufbare Befehlsschnittstelle und die Ablauflogik für die Messumgebung. Eine Implementierung der View ist im Paket `mm.view` vorgesehen. Eine Benutzeroberfläche wurde jedoch während der Diplomarbeit nicht realisiert. Sie könnte die Definition der benötigten Konfigurationen deutlich erleichtern. Diese sind zurzeit in Dateien abgelegt, die von Hand editiert werden müssen. Durch eine Benutzeroberfläche könnte auch die Darstellung wichtiger Messergebnisse, die zurzeit noch durch selbsterzeugte Anfragen aus der Datenbank ausgelesen werden, erfolgen.

Im Paket `mm.connection` ist das Erstellen einer Verbindung mit der Datenbank zum Wegschreiben der Messdaten gekapselt.

Alle Ausnahmen, die während der Verarbeitung auftreten können, wurden im Paket `mm.exceptions` gesammelt.

Das Paket `mm.lib` stellt komplett unabhängige Funktionalitäten zur Verfügung, wie z. B. die korrekte Verarbeitung eines SSH-Aufrufes.

Da die in diesem Bereich angesiedelten Klassen nur das Funktionieren der Messumgebung sicherstellen, werden diese in der Diplomarbeit nicht besprochen. Der Schwerpunkt liegt auf den für das Framework entwickelten Modellklassen.

4.1 Konfiguration des zu vermessenden Systems

In Abschnitt 3.2.3 wurde bereits deutlich gemacht, dass für einen Messlauf eine große Menge an Informationen benötigt wird. Um eine Grobstruktur zu bilden, in der die Informationen der Messumgebung und eines Messablaufes aufgenommen werden können, wurden die Klassen `Environment`, `Project` und `MeasurementConfig` gebildet, wie in Abbildung 4.2 dargestellt.

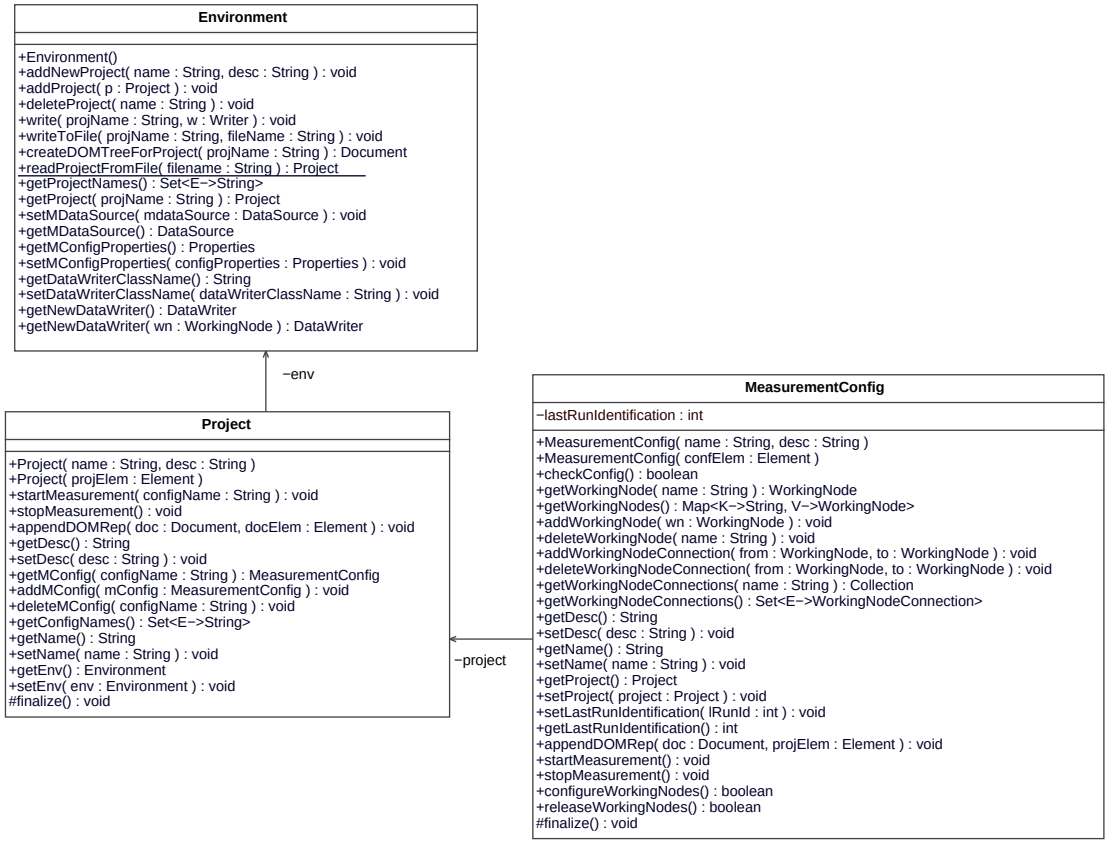

Abbildung 4.2: Klassen der Strukturierung für alle weiteren Komponenten

Die Klasse `Evironment` erfasst alle Informationen, die direkt der Messumgebung zugeordnet sind. Sie speichert vor allem eine Menge von Projekten (`Project`), die sie eingelesen hat. In jedem Projekt ist es möglich, mehrere Messkonfigurationen

(`MeasurementConfig`) anzulegen. Die Klasse `Project` bietet dabei nur eine zusätzliche Möglichkeit der logischen Unterteilung, so dass mehrere Messkonfigurationen, die z. B. das gleiche Ziel verfolgen, zusammengefasst werden können. Außerdem ist es möglich, eine Messung für alle Messkonfigurationen eines Projektes zu starten.

Ein Projekt wird durch seinen Namen eindeutig identifiziert und kann mit einer Beschreibung versehen werden. Innerhalb eines Projektes ist jede Messkonfiguration ebenso durch ihren Namen eindeutig identifiziert. Auch diese kann um eine Beschreibung erweitert werden.

Damit es möglich wird, mehrere Messumgebungen zu starten, wird für jede neu gestartete Messumgebung eine neue Instanz der Klasse `Environment` erzeugt. In ihr werden zusätzlich Informationen, die für das Abspeichern der Messdaten benötigt werden, gespeichert. Dabei wird der zu verwendende Datenschreiber (`DataWriter`) und der zu verwendende Speicherort (`DataSource`) festgelegt. Im Rahmen der Diplomarbeit wurde für das Abspeichern von Messdaten eine Standardmethode (`DefaultDataWriter`) entwickelt. Die Implementierung hierzu wird in Abschnitt 4.4.4 erklärt.

Die Informationen über die Zusammensetzung eines Projektes und die in ihm enthaltenen Messabläufe werden in einer XML-Konfigurationsdatei persistent abgespeichert. Sie muss dem Dokumenttyp *MM-ProjectDef* entsprechen. Ein Beispiel für eine Projektkonfiguration befindet sich im Anhang dieser Diplomarbeit (vgl. Anschnitt A.3). Die Daten aus der Konfigurationsdatei können eingelesen und auch geschrieben werden.

Zum Einlesen der Daten wird die Datei durch einen XML-DOM-Parser serialisiert und validiert. Danach wird durch Übergabe des Project-Elements an den Konstruktor `Project(Element e)` eine neue `Project`-Instanz erzeugt. Das Anlegen der Instanzen von benötigten Unterklassen wird dementsprechend durch die Implementierung eines Konstruktors mit Element-Parameter realisiert. So legt z. B. der Konstruktor `Project(Element e)` alle im Projekt definierten Messkonfigurationen mit Hilfe des Konstruktors `MeasurementConfig(Element e)` an.

Das Schreiben einer Projekt-Konfiguration wird durch die Erzeugung eines DOM-Baumes realisiert. Dabei legt jede Klasse ihre Informationen unter dem ihr übergebenen Element-Knoten ab. Dies wird stets über die Funktion `appendDOMRep(Dokument d, Element e)` gewährleistet. Der resultierende DOM-Baum wird als XML-Datei serialisiert und abgespeichert.

Diese Vorgehensweise beim Einlesen und Speichern garantiert eine problemlose Erweiterbarkeit der abzuspeichernden Daten, falls neue Komponenten ins Framework aufgenommen werden.

Im nachfolgenden Abschnitt werden die weiteren Bestandteile einer Messkonfiguration besprochen. Hierunter fällt vor allem die Abbildung eines Arbeitsknotens durch eine entsprechende Komponente.

4.1.1 Bestandteile einer Messkonfiguration

In Abschnitt 3.2 wurde verdeutlicht, dass ein zu vermessendes System durch die Definition von Arbeitsknoten repräsentiert wird. Für die Realisierung bedeutet dies, dass

eine Messkonfiguration aus einer Menge von Arbeitsknoten besteht. Ein Arbeitsknoten wird dabei durch die Klasse `WorkingNode` repräsentiert. Diese implementiert mit der Hilfe anderer Klassen alle Eigenschaften, die in Abschnitt 3.2.1 beschrieben sind.

Eine Messung benötigt im wesentlichen die folgenden drei Konfigurationsdateien:

- die Konfigurationsdatei der Messumgebung, welche alle Informationen beinhaltet, die direkt der Messumgebung zugeordnet sind
- die Konfigurationsdatei für Arbeitsknotentypen, welche alle Informationen beinhaltet, die für jeden Arbeitsknotentyp benötigt werden und
- die Konfigurationsdatei für Messabläufe, welche die Informationen über konkret durchzuführende Messabläufe enthält

Im Anhang A.1 ist für jede Konfigurationsdatei ein Beispiel aufgelistet. Darüber hinaus können vor allem zur Durchführung von Konfigurationsschritten auch noch weitere Konfigurationsdateien zum Einsatz kommen. Deren Struktur und Ablage ist jedoch nicht vorgegeben, weil sie Informationen beinhalten, die von einer neu entwickelten Komponente benötigt werden.

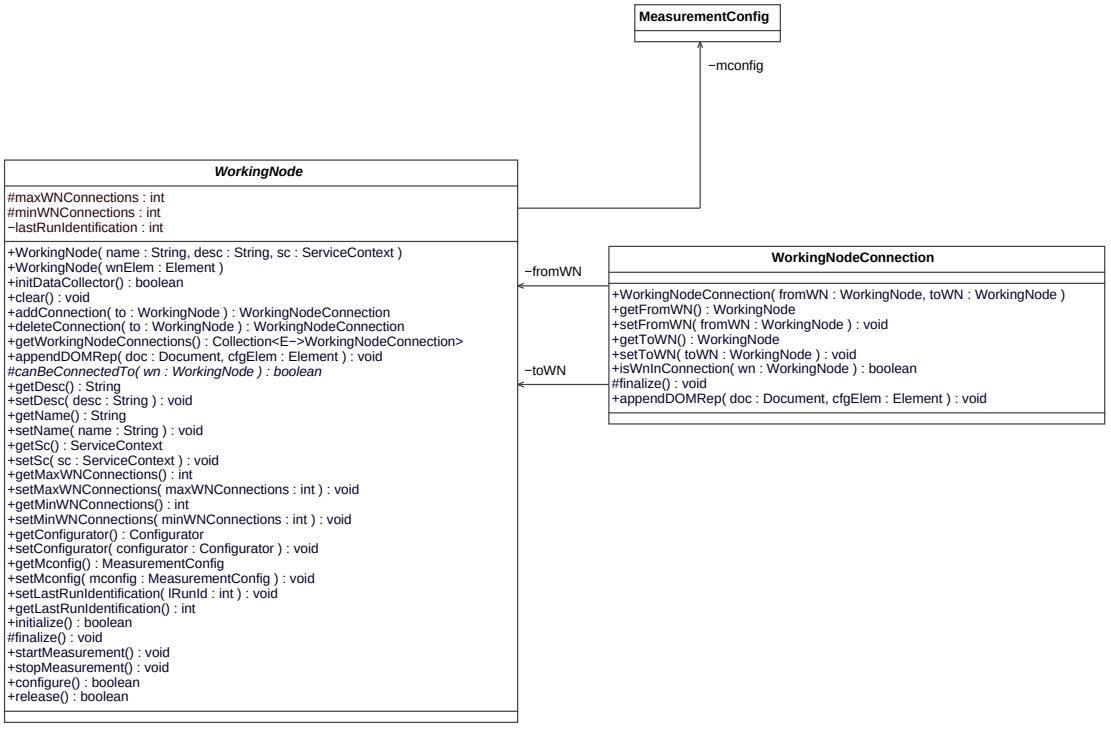

Abbildung 4.3: Beziehungen zwischen Messkonfiguration, Arbeitsknoten sowie deren Verbindungen

Im Folgenden werden die Komponenten des Frameworks beginnend mit der Klasse `WorkingNode` (s. Abbildung 4.3) beschrieben. Die vorgestellten Komponenten müssen zur Vermessung eines konkreten Systems weiter spezialisiert werden. Danach wird es möglich, für die entwickelten Komponenten die Konfiguration in den Konfigurationsdateien vorzunehmen.

Abbildung der Verbindungen zwischen Arbeitsknoten

Die Aufrufverbindungen zwischen den Arbeitsknoten werden durch eine eigenständige Komponente `WorkingNodeConnection` erzeugt. Diese werden ebenso der Messkonfiguration zugewiesen.

Abbildung 4.3 zeigt die Beziehungen zwischen Messkonfiguration, Arbeitsknoten und deren Verbindungen untereinander. Ein Arbeitsknoten wird hierbei innerhalb einer Konfiguration durch seinen Namen eindeutig identifiziert. Jedem Arbeitsknoten kann zusätzlich eine Beschreibung zugewiesen werden.

Abbildung der Aufrufbarkeit

Jeder Arbeitsknoten besitzt eine Beschreibung der Aufrufschnittstelle der von ihm repräsentierten Anwendung (`ServiceContext`). Damit spezifische Aufrufschnittstellen ausgezeichnet werden können, muss die Klasse `ServiceContext` weiter spezialisiert werden. Die Informationen einer RMI-Schnittstelle (`RMIServiceContext`) und die einer JDBC-Schnittstelle (`JDBCServiceContext`) sind als Standardkomponenten ins Framework eingebaut. Wird eine noch speziellere Auszeichnung notwendig, könnte man z. B. die Klasse `ACCacheJDBCServiceContext` von `JDBCServiceContext` ableiten.

In Abbildung 4.4 sind die Klassen mit ihren Beziehungen untereinander dargestellt. Damit auch ein `ServiceContext` in der Konfigurationsdatei ausgezeichnet werden kann, sind für jeden `ServiceContext` der Konstruktor `ServiceContext(Element e)` und die Methode `addDOMRep(Document d, Element e)` zu überschreiben, deren Funktion bereits mit den Klassen `Project` und `MeasurementConfig` vorgestellt wurde.

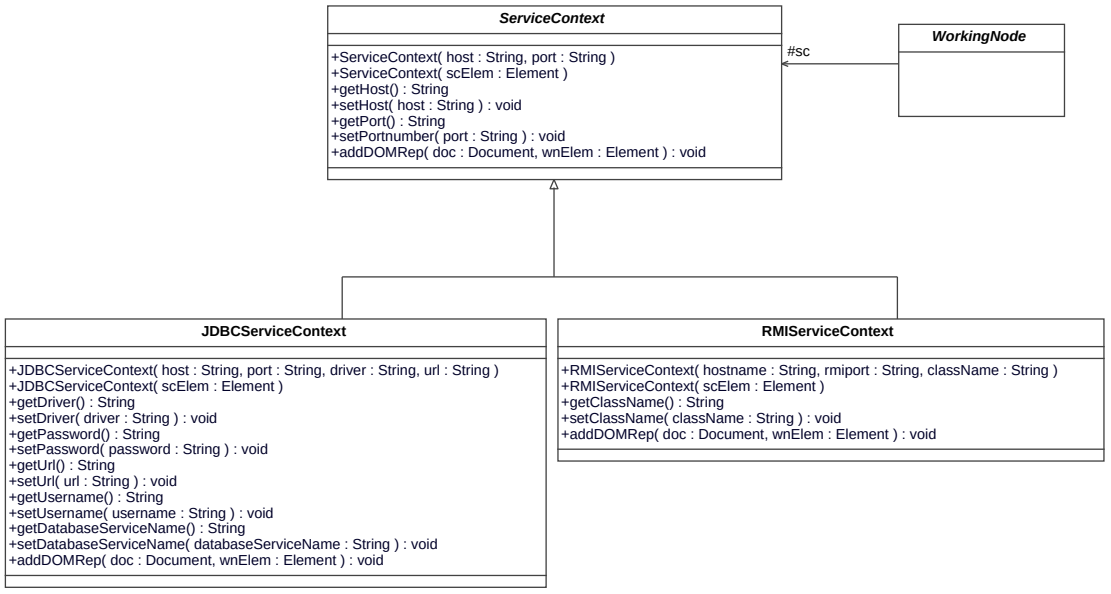

Abbildung 4.4: Jeder WorkingNode muss einen ServiceContext auszeichnen

Abbildung der Beobachtbarkeit

Für jeden Arbeitsknoten können beliebig viele Beobachter (`MeasurementObserver`) in der Konfigurationsdatei für Arbeitsknotentypen (vgl. Abschnitt A.2) ausgezeichnet werden.

Für jeden Beobachter kann über die Methode `setIsActive(boolean b)` bestimmt werden, ob er während der Messung aktiv oder inaktiv ist. Aus diesem Grund sollten Beobachter unabhängig voneinander agieren können.

Um Messdaten erfassen zu können, benutzen die Beobachter Ausführungskontexte. Jedem Beoachter wird zu Beginn der Messung der initiale Ausführungskontext vom Arbeitsknoten zur Verfügung gestellt, mit dessen Hilfe er Messdaten erfassen kann. Wie ein Beobachter Ausführungskontexte exakt verwenden muss, um Messwerte zu erfassen, wird in Abschnitt 4.4 explizit behandelt. Da er oft Unterkontexte bilden wird, ändert sich häufig sein aktuell verwendeter Ausführungskontext. Er wird vom Beobachter über das Attribut `currentExecutionContext` verwaltet.

Ein Beobachter implementiert die Schnittstelle `java.util.Observer`, welche es ihm erlaubt, sich bei beobachtbaren Objekten zu registrieren. Diese benachrichtigen jeden registrierten Beobachter über die Funktion `update(Observable obs, Object o)`, falls sich der Status des Objektes ändert. Dies ist die empfohlene Vorgehensweise, um ein System zu beobachten. Ein Beobachter kann aber auch durch gezielte Aufrufe und Abfragen seine Informationen erhalten. Die konkrete Implementierung ist nicht vorgeschrieben (s. Abbildung 4.5).

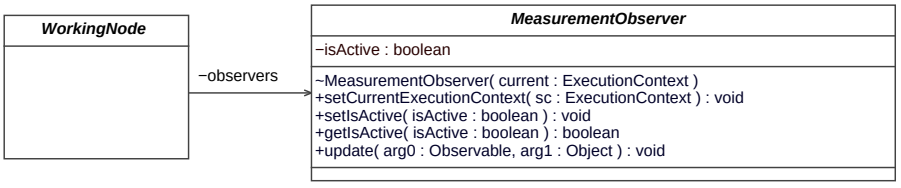

Abbildung 4.5: Der Arbeitsknoten hält die Menge ihm zur Verfügung stehender Beobachter

Abbildung der Messdatenerfassung

Jeder Arbeitsknoten stellt eine Möglichkeit zur Verfügung, die für ihn erfassten Informationen aufzunehmen. Dies tut er, indem er einen `MeasurementDataCollector` initialisiert (s. Abbildung 4.6). Dieser stellt eine Message-Queue für die übermittelten Messinformationen zur Verfügung. Die Message-Queue wurde mit Hilfe des *JMS (Java Messaging Service)* implementiert. Sie wird von den Ausführungskontexten genutzt, welche die erfassten Messwerte und Kontextinformationen an die Queue senden. Nähere Informationen über die Messdatenverwaltung finden sich in Abschnitt 4.4.

Abbildung der automatischen Konfigurierbarkeit

Damit ein Arbeitsknoten auch Konfigurationsschritte ausführen kann, benutzt er eine Konfiguratorkomponente (`Configurator`). Diese sammelt die auszuführenden

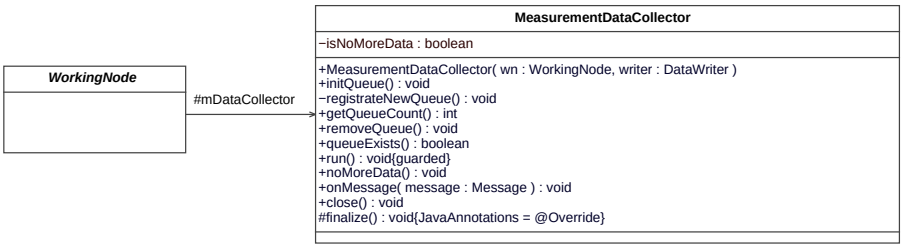

Abbildung 4.6: Der Arbeitsknoten erfasst übergebene Messdaten mit Hilfe der Klasse MeasurementDataCollector

Konfigurationsschritte und führt sie über die Funktion `doConfiguration()` aus, bevor die Messung startet (s. Abbildung 4.7). Hierbei können Fehler entstehen, welche die Durchführung der Messung verhindern. Beispielsweise kann es vorkommen, dass eine Datenbank nicht gestartet werden kann oder eine Teilanwendung nicht erreichbar ist. Der Konfigurator kann daraufhin angewiesen werden, nicht erfolgreiche Konfigurationsschritte nochmal durchzuführen. Dies geschieht, indem die Funktion `doConfiguration()` ein weiteres Mal aufgerufen wird.

Welche Konfigurationsschritte für einen Arbeitsknoten durchzuführen sind, wird in der Projekt-Konfigurationsdatei (vgl. Abschnitt A.3) für jede Instanz separat festgelegt. Sind Konfigurationsschritte für einen Arbeitsknoten immer vorzunehmen, können diese auch in der spezialisierten WorkingNode-Klasse dem Konfigurator hinzugefügt werden. Nähere Informationen über die Implementierung der Konfigurationsschritte finden sich in Abschnitt 4.3.

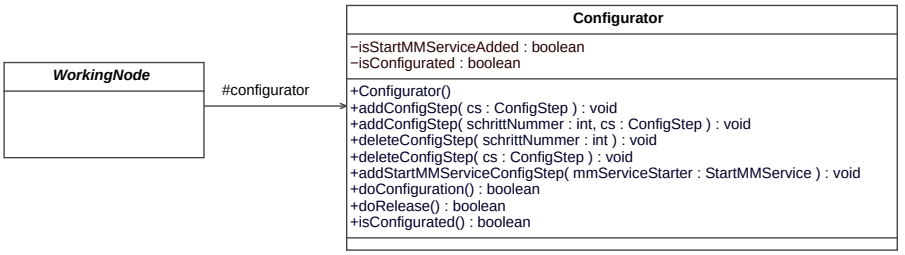

Abbildung 4.7: Der Arbeitsknoten benutzt einen Konfigurator

4.2 Simulation fiktiver Benutzer

Um eine Client-Anwendung, wie in Abschnitt 3.3 beschrieben, zu beobachten, genügt es, eine Unterklasse von der Klasse `WorkingNode` zu bilden und diese zu implementieren, wobei die benötigten Beobachter, Ausführungskontexte sowie weitere Komponenten für diesen Arbeitsknoten festgelegt werden.

Damit eine Simulation möglich wird, müssen die speziellen Eigenschaften eines simulierten Clients berücksichtigt werden. Aus diesem Grund wurde die Unterklasse `SimulatedClient` von `WorkingNode` abgeleitet (s. Abbildung 4.8). Diese enthält eine Menge von Arbeitsschritten (`WorkUnit`), die sie mit Hilfe des Schedulers (`Scheduler`) einplant.

Da für den ACCache vor allem JDBC-Transaktionen als Arbeitsschritte durchzuführen sind, wurde eine entsprechende Spezialisierung bereits in das Framework aufgenommen. Die konkret durchzuführenden JDBC-Transaktionen werden jedoch erst für den ACCache implementiert.

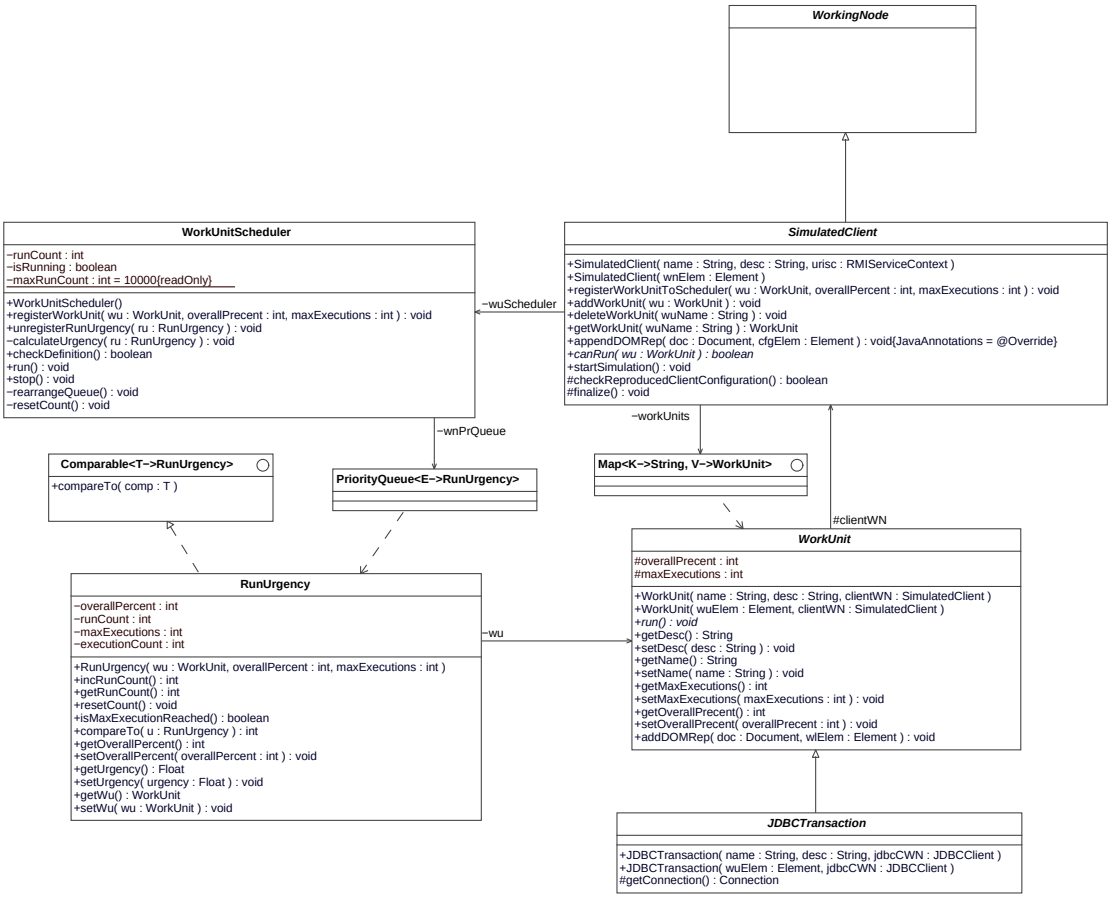

Abbildung 4.8: Der simulierte Client benutzt WorkUnit und Scheduler

Jeder simulierte Client-Arbeitsknoten darf nur eine ausgehende Verbindung zu anderen Arbeitsknoten besitzen. So entfällt die Implementierung eines Routers, der die Pakete auf die ausgehenden Verbindungen verteilt. Daher wird im nachfolgenden Abschnitt nur die Funktionalität des Scheduler näher betrachtet.

4.2.1 Einplanung der Arbeitsschritte

Die Arbeitsschritte, die ein simulierter Client-Arbeitsknoten ausführen soll, werden in der Konfigurationsdatei für Messkonfigurationen festgelegt. Dabei muss für jeden Arbeitschritt festgelegt werden, wie oft (in Prozent) er im Vergleich zu den anderen angelegten Arbeitschritten ausgeführt werden soll. Ergibt die Summe aller Ausführungshäufigkeiten nicht 100 Prozent, endet das Einlesen der Konfigurationsdatei mit einem Fehler. Um die Anzahl der Ausführungen begrenzen zu können, kann die maximale Anzahl von Ausführungen zusätzlich festgelegt werden. Soll dies nicht geschehen, lässt sich dies durch Angabe der Konstante `noLimit` festlegen. Das nachfolgende Code-Beispiel verdeutlicht die Art der Verwendung.

```
<workingnode class="SimulatedClient" ...>
    <workunit class="WorkUnitA" ... >
        <schedule maxExecutions="noLimit" overallPercent="60"/>
    </workunit>
    <workunit class="WorkUnitB" ... >
        <schedule maxExecutions="300" overallPercent="30"/>
    </workunit>
    <workunit class="WorkUnitC" ... >
        <schedule maxExecutions="noLimit" overallPercent="10"/>
    </workunit>
</workingnode>
```

Die im Beispiel angelegte *WorkUnitB* wird dabei nur 300 mal ausgeführt, während die *WorkUnitA* 600 Durchläufe und die *WorkUnitC* 100 Durchläufe vollzieht. Danach wird nur noch *WorkUnitA* und *WorkUnitC* ausgeführt, bis die Messung von der Messumgebung gestoppt wird.

Der Scheduler implementiert genau das zuvor beschriebene Verhalten. Während des Einlesens der Konfigurationsinformationen registriert der simulierte Client-Arbeitsknoten über die Funktion `registerWorkUnitToScheduler(WorkUnit w,int overallPercent,int maxExecutions)` alle für ihn definierten Arbeitschritte beim Scheduler. Dieser legt aufgrund der ihm übergebenen Werte ein neues `RunUrgency`-Objekt in seiner Prioritätswarteschlange ab.

Dabei wird für jedes `RunUrgency`-Objekt die Dringlichkeit der Ausführung berechnet. Das `RunUrgency`-Objekt mit der höchsten Dringlichkeit wird jeweils als nächstes eingeplant. Nach der Ausführung eines Arbeitsschrittes werden alle Dringlichkeitswerte neu berechnet.

Die Dringlichkeit wird vor der Ausführung des ersten Arbeitsschrittes durch die Ausführungshäufigkeit (`overallPercent`) definiert. Danach wird sie für einen Arbeitsschritt k wie folgt berechnet:

$$\textit{Dringlichkeit}\ (k) = \textit{Zielhäufigkeit}\ (k) - \frac{\textit{Ausführungen Arbeitsschritt}\ (k)}{\sum_{i=1}^{n} \textit{Ausführungen Arbeitschritt}\ (i)}$$

Da durch diese einfache Berechnung der Wert der Dringlichkeit immer kleiner wird, wird sie nach dem 10000. Durchlauf wieder auf die ursprünglichen Ausführungshäufigkeit zurückgesetzt. Die tatsächliche prozentuale Einplanhäufigkeit ändert sich dadurch kaum. Wird ein Arbeitsschritt nicht mehr ausgeführt, verzerrt er die Anzahl der Einplanhäufigkeiten für die übrig gebliebenen Arbeitsschritte, da er nicht mehr mit eingeplant wird. Das Verhältnis zwischen den übrigen Arbeitschritten bleibt aber gewahrt.

Abbildung 4.9 zeigt, wie sich die Dringlichkeitswerte nach Durchführung eines Arbeitsschrittes ändern und verdeutlicht, dass die gewünschten Einplanhäufigkeiten erreicht werden. In der Abbildung wurden dabei die Werte aus dem Code-Beispiel verwendet. Es ist jeweils der Zustand der Priority-Queue fortlaufend dargestellt. Nach zehn Ausführungen wurde hier bereits das korrekte Mengenverhältnis erreicht.

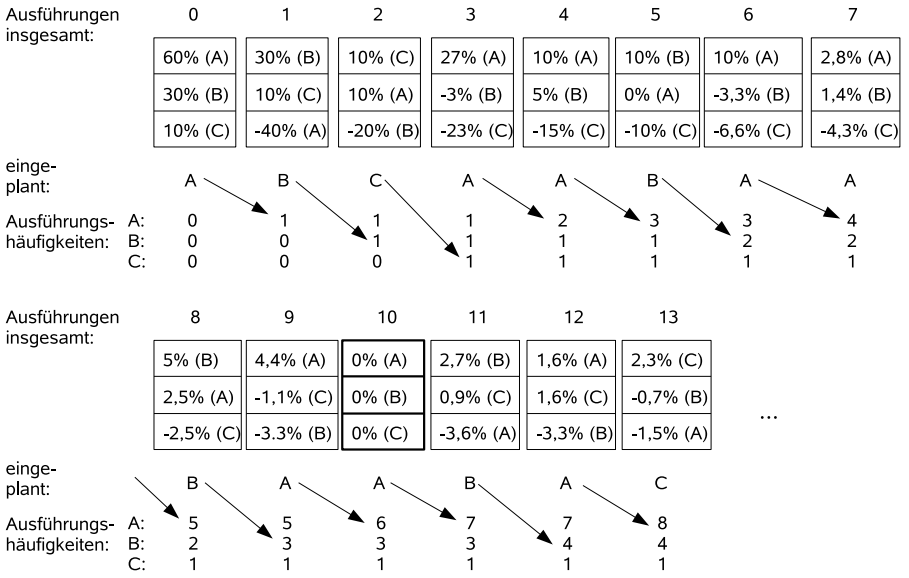

Abbildung 4.9: Die fortlaufende Berechnung der Dringlichkeit und die sich daraus ergebende Einplanung der Arbeitsschritte.

4.3 Automatisierung von Konfigurationsschritten

Die Durchführung von Konfigurationsschritten ist für die Messumgebung ebenso notwendig wie für jeden zu vermessenden Arbeitsknoten. Da es durchaus denkbar ist, dass sich die Anzahl der durchzuführenden Konfigurationsschritte für den gleichen Arbeitsknotentyp ändert, können diese in der Konfigurationsdatei für Messkonfigurationen für jede Arbeitsknoteninstanz angegeben werden.

Um eine Menge von Konfigurationsschritten zu verwalten, muss eine Komponente des Frameworks einen Konfigurator einsetzen, wie es bereits in Abschnitt 4.1.1 (Konfigurierbarkeit) für den Arbeitsknoten gezeigt wurde. Für die Messumgebung tut dies der `EnvironmentController`. Er entscheidet aufgrund der Einstellungen in der Konfigurationsdatei der Messumgebung (vgl. Abschnitt A.1), welche Konfigurationsschritte in den Konfigurator aufzunehmen sind.

Ein Konfigurationsschritt benutzt den internen Status `CsState`, um dem Konfigurator anzuzeigen, ob die Durchführung der Konfiguration erfolgreich war oder nicht. Wurde der Konfigurationsschritt noch nicht durchgeführt oder hat er alle seine Änderungen wieder rückgängig gemacht, befindet er sich im Status `CS_NOT_EXECUTED`. Wurde der Konfigurationsschritt erfolgreich durchgeführt, wechselt er in den Status `CS_READY`. Fallen während der Durchführung Fehler an, so wechselt der Status auf den Wert `CS_ERROR`. Dabei speichert der Konfigurationsschritt alle entstandenen Ausnahmen, um dem Benutzer über den Konfigurator die aufgetretenen Fehler mitteilen zu können. Nachdem ein Fehler aufgetreten ist, kann dieser durch die Funktion `getError()` jederzeit abgefragt werden. Hierdurch ist es z. B. auch möglich, eine Konfiguration abzubrechen, die Fehler in eine Log-Datei zu schreiben und mit der Durchführung der nachfolgenden Messkonfiguration fortzusetzen. Zusätzlich besteht die Möglichkeit, den Konfigurationsschritt über die Funktion `setDeactivated()` zu deaktivieren. Damit es für einen Benutzer (z. B. über eine grafische Oberfläche) mög-

lich ist, einen Konfigurationsschritt wiederholen zu lassen, kann er über die Methode `setNotExecuted()` manuell zurückgesetzt werden.

In Abbildung 4.10 lässt sich erkennen, dass `EnvironmentController` und `WorkingNode` den Konfigurator verwenden, um Konfigurationsschritte auszuführen. Zusätzlich sind die während der Startphase der Messumgebung verwendeten Konfigurationsschritte aufgenommen. Diese werden im nachfolgenden Abschnitt nochmals kurz vorgestellt.

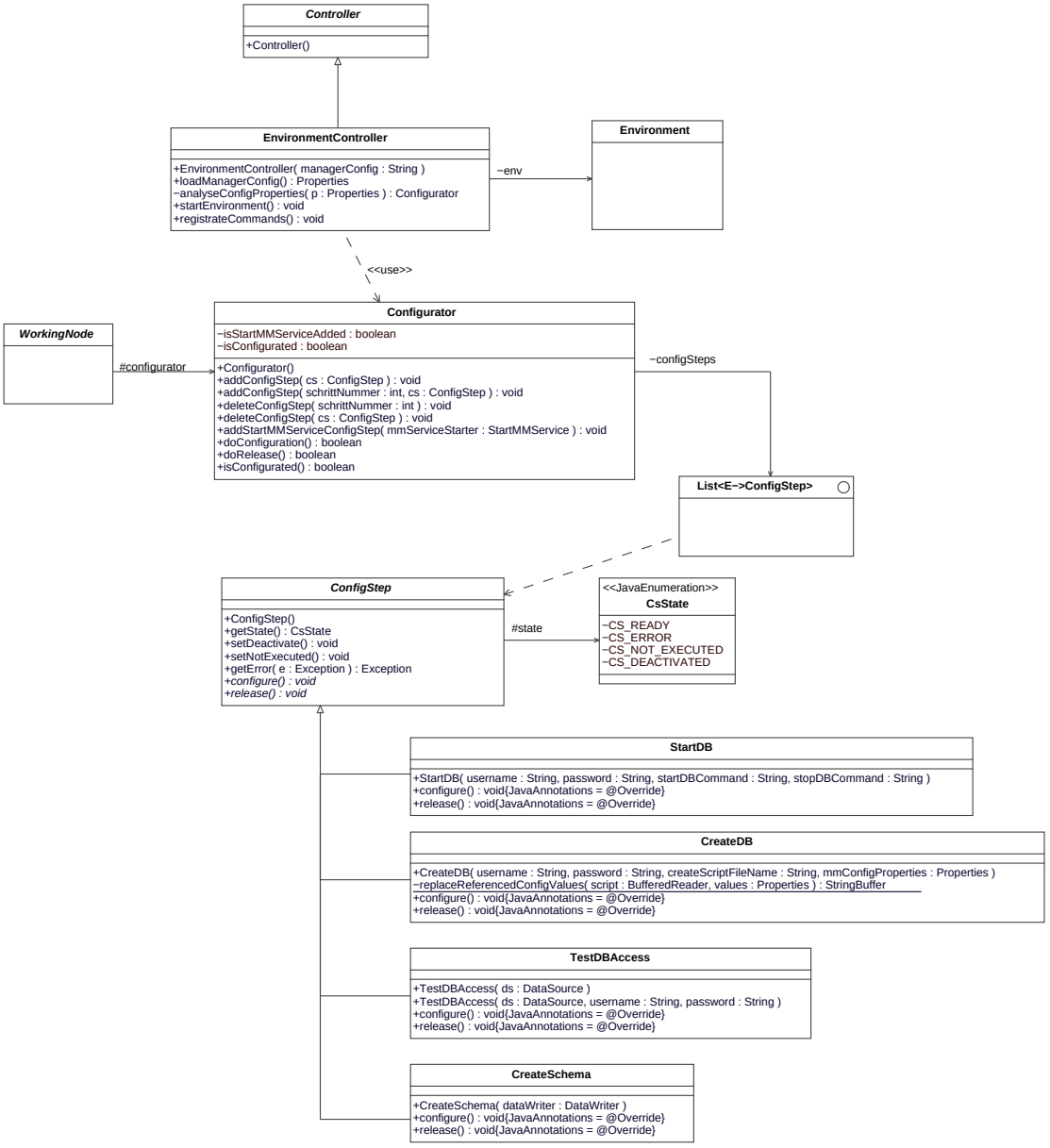

Abbildung 4.10: Die Konfigurationsschritte der Messumgebung im Zusammenhang mit dem Konfigurator

Konfigurationschritte der Messumgebung

Die Typen der Konfigurationsschritte, welche die Messumgebung beim Starten ausführen kann, sind fest vorgegeben. Je nachdem wie die Werte der Konfigurationsdatei

der Messumgebung angepasst wurden, ändert sich nur die Anzahl der durchgeführten Schritte. Für jede Instanz einer Messumgebung muss sichergestellt sein, dass sie nach dem Starten der Messung die erhaltenen Messdaten auch abspeichern kann. Aus diesem Grund ist in der Konfigurationsdatei der Messumgebung anzugeben, von welchem Datenschreiber `Datawriter` die erhaltenen Messdaten abgespeichert werden. Wie die Schnittstelle des Datenschreibers aussieht und welche Komponenten ihm zugeordnet sind, wird in Abschnitt 4.4.4 genau erklärt.

Zur Zeit werden die Daten in eine DB2-Datenbank gespeichert. Die Messumgebung wurde daher mit der Möglichkeit ausgestattet, eine Datenbankinstanz selbstständig zu starten, falls dies notwendig ist. Die Klasse `StartDB` implementiert diesen Konfigurationsschritt, welcher die Möglichkeit zur Verfügung stellt, mit Hilfe einer SSH-Verbindung den Datenbankserver zu starten. Dazu muss für SSH-Zugriffe die Authentifikationsmethode *password* oder *public-key* zur Verfügung stehen. Über das Attribut `MData:doStartDB` kann entschieden werden, ob der Konfigurationsschritt ausgeführt wird (`MData:doStartDB="true"`) oder nicht (`MData:doStartDB="false"`).

In der Konfigurationsdatei der Messumgebung müssen zur Durchführung dieses Schrittes die folgenden Werte abgelegt sein:

1. `MData:DBHost` - Der Hostname (oder IP-Adresse), auf dem die Datenbankinstanz installiert ist.
2. `MData:StartDB:Username` - Der Benutzername des Benutzers, der berechtigt ist, die Datenbankinstanz zu starten.
3. `MData:StartDB:Password` - Das Passwort des Benutzers, der berechtigt ist, die Datenbankinstanz zu starten. Diese Angabe wird nur benötigt, wenn die Authentifikationsmethode *password* verwendet werden muss bzw. soll.
4. `MData:StartDB:StartCommand` - Der Kommandozeilenbefehl, der die Datenbankinstanz startet.
5. `MData:StartDB:StopCommand` - Der Kommandozeilenbefehl, der die Datenbankinstanz stoppt.

Ist das Starten der Datenbankinstanz vorgesehen, wird eine SSH-Verbindung zum Zielhost aufgebaut, der Benutzer angemeldet und der angegebene Befehl ausgeführt. Im Prinzip kann auf die gleiche Art und Weise, durch Ausführung des `StopCommand`, z. B. beim Beenden der Messumgebung, die Datenbankinstanz wieder gestoppt werden. Dies wird jedoch derzeit nicht getan.

Wenn die Datenbankinstanz gestartet wurde oder bereits läuft, kann eine neue Datenbank automatisch angelegt werden. Den dafür notwendigen Konfigurationsschritt implementiert die Klasse `CreateDB`. Der Wert des Attributs `MData:doCreateDB` entscheidet, ähnlich wie beim vorherigen Konfigurationsschritt, über die Durchführung des Schrittes, der die folgenden Konfigurationsinformationen umfasst:

1. `MData:DBHost` - Der Hostname (oder IP-Adresse), auf dem die Datenbankinstanz installiert ist.

2. `MData:CreateDB:Username` - Der Benutzername des Benutzers, der berechtigt ist, die Datenbank anzulegen.

3. `MData:CreateDB:Password` - Das Passwort des Benutzers, der berechtigt ist, die Datenbank anzulegen.

4. `MData:CreateDB:CreateScriptTplFile` - Die Angabe eines SQL-DDL-Skripts, das zum Anlegen der Datenbank ausgeführt werden soll. Es kann mit Werten aus der Konfigurationsdatei parametrisiert werden.

Zum Anlegen der Datenbank wird wiederum eine SSH-Verbindung zum Datenbankserver aufgebaut. Nach erfolgreicher Anmeldung werden die SQL-Befehle im Skript über ein Kommandozeilentool (in unserem Fall *db2 <sql-command>*) ausgeführt. Im angegebenen SQL-File lassen sich Werte, die in der Konfigurationsdatei enthalten sind, verwenden, indem der Name des Wertes in spitzen Klammern eingefasst als Platzhalter verwendet wird. Ein Beispiel hierzu befindet sich in Abschnitt A.1.1.

Die beiden eben vorgestellten Konfigurationsschritte sind optional und werden nur ausgeführt, wenn es in der Konfigurationsdatei der Messumgebung angegeben ist. Der nun folgende Konfigurationsschritt `TestDBAccess` wird immer ausgeführt. Er testet die Verbindung zur angegebenen Datenbank. Dabei wird zum Test eine Tabelle angelegt, in der einige Testwerte gespeichert werden. Danach wird die Tabelle ausgelesen und wieder gelöscht.

Als letztes wird optional der Konfigurationsschritt `createSchema` durchgeführt. Dieser legt die statischen Tabellen und alle zur Erzeugung eindeutiger Primärschlüsselwerte benötigten Sequenzen in der Datenbank an. Dazu führt er die Funktion `createStaticSchemaObjects()` auf dem ihm übergebenen Datenschreiber aus. Welche Objekte für den `DefaultDataWriter` angelegt werden müssen, wird in Kapitel 4.4.4 erklärt.

Für die beiden Konfigurationsschritte `TestDBAccess` und `CreateSchema` müssen in der Konfigurationsdatei stets die folgenden Werte angegeben sein:

1. `MData:DBHost` - Der Hostname (oder die IP-Adresse) der Datenbankinstanz.

2. `MData:DBName` - Der Name der Datenbank.

3. `MData:JDBCUrl` - Die JDBC-URL zum Verbinden mit der Datenbank.

4. `MData:Driver` - Der Klassenname des JDBC-Treibers.

5. `MData:Username` - Der Name des Datenbankbenutzers mit Schreibberechtigung.

6. `MData:Password` - Das Passwort des Datenbankbenutzers.

7. `MData:DataWriterClass` - Der Klassenname des zu verwendenden Datenschreibers.

8. `MData:doCreateSchema` - Die Angabe, ob das Schema neu erstellt werden soll (`true`) oder nicht (`false`).

Dabei müssen Werte, die bereits für andere Konfigurationsschritte aufgenommen wurden, nicht redundant angegeben werden.

Nach Durchführung aller nötigen Konfigurationsschritte für die Messumgebung und der Betrachtung der in Abschnitt 4.1 vorgestellten Möglichkeiten, die Struktur der verteilten Teilanwendungen zu erfassen, wird nun erklärt, wie die Messumgebung die Messdaten mit Hilfe von Beobachtern erfasst, überträgt und abspeichert.

4.4 Verwaltung von Messdaten

Im Folgenden wird die Implementierung der Aufgaben zur Messdatenerfassung besprochen, die in Kapitel 3 vorgestellt wurden. Um besser vertehen zu können, welche Komponenten des Frameworks hierfür zusammenarbeiten und wie sie die Messdaten erfassen, wird zunächst der konkrete Ablauf der Messdatenerfassung vorgestellt. Besonders wichtig ist hierbei, dass jeder Arbeitsknoten während der Messdatenerfassung durch einen Messdatendienst unterstützt wird, um die Beobachtung einer Teilanwendung von einem entfernten Host aus zu gewährleisten. Der Messdatendienst kann von der Messumgebung über eine RMI-Schnittstelle aufgerufen werden. Diese stellt eine generisch[1] verwendbare Funktion `runCommand (String cmdClassName, String[] args)` zur Verfügung, welche es ermöglicht, die Menge der ausführbaren Befehle dynamisch zu erweitern. Außerdem können über die Funktion `runNative(String cmdLine)` native Programme auf dem Zielhost auf einfache Art und Weise, ohne expliziten Aufbau einer SSH-Verbindung, zur Ausführung gebracht werden.

4.4.1 Ablauf der Messdatenerfassung

Abbildung 4.11 verdeutlicht den Ablauf der Messdatenerfassung durch 11 nacheinander ablaufende Schritte. Dazu wird angenommen, dass die Messumgebung auf einem anderen Host abläuft als die zu beobachtende Teilanwendung. Auf Host *A* wurde die Messumgebung gestartet. Sie hat bereits alle notwendigen Konfigurationsinformationen eingelesen und verarbeitet. Für die bereits verarbeitete Messkonfiguration 1 wurden die notwendigen Konfigurationsschritte für alle Arbeitsknoten bereits durchgeführt, so dass der eigentliche Messlauf gestartet werden kann.

Im Beispiel aus Abbildung 4.11 wird die Messung aus der Sicht eines einzelnen Arbeitsknotens betrachtet. Er repräsentiert eine zu vermessende Teilanwendung, z. B. das ACCache-System. Die Messdatenerfassung läuft für alle anderen Arbeitsknoten der Messkonfiguration auf gleiche Art und Weise ab.

Im ersten Schritt instanziiert der *ACCache*-Arbeitsknoten einen Messdatenerfasser (`DataCollector`). Dieser stellt eine JMS-Queue (*Java Messaging Service*)[2] zur Verfügung. Als JMS-Implementierung wurde das *openjms*-Projekt benutzt, welches bei SourceForge[3] zur Verfügung steht. Die JMS-Queue wird mit dem Namen des Arbeitsknotens, erweitert um den Zusatz „_queue“, über einen JNDI-Service (*Java*

[1] In der generischen Programmierung werden einzelne Funktionen und Klassen immer möglichst allgemein geschrieben, so dass sie für unterschiedliche Datentypen verwendet werden können. Im vorliegenden Fall wird eine Funktion implementiert, die beliege Command-Instanzen ausführen kann und dabei beliebig viele Sting-Parameter an diese übergibt.

[2] http://de.wikipedia.org/wiki/Java_Message_Service und http://java.sun.com/products/jms/

[3] http://openjms.sourceforge.net/

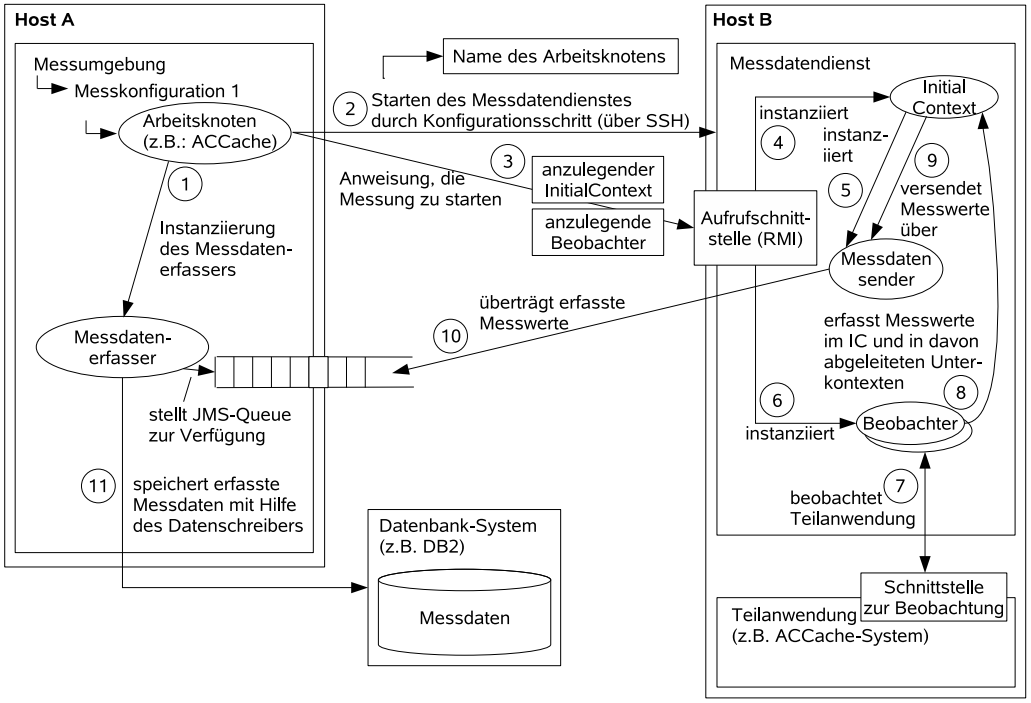

Abbildung 4.11: Konkreter Ablauf der Messdatenerfassung über Beobachter.

Naming and Directory Interface)[4] für entfernt ablaufende Komponenten zugänglich gemacht. Der JNDI-Service läuft zusammen mit dem openjms-Server auf dem Host der Messumgebung ab.

Nachdem die Messumgebung in die Lage versetzt wurde, Messdaten zu empfangen, kann sie im zweiten Schritt den Messdatendienst (`MeasurementService`) starten. Der Messdatendienst ist die „rechte Hand“ eines Arbeitsknotens. Der Dienst erlaubt es, entfernt Programmkomponenten auszuführen. Ohne die Einrichtung eines eigenständigen Dienstes müssten alle Beobachter auf dem Host der Messumgebung instanziiert werden. Dies würde bedeuten, dass alle Beobachtungen nur über einen entfernten Nachrichtenaustausch möglich wären, was zu vermeiden ist, um die Messergebnisse durch ein zu hohes Nachrichtenaufkommen nicht zu verfälschen. Darüber hinaus besteht nun die Möglichkeit, die Beobachter auf eigens dafür vorgesehenen Hosts auszuführen. Die Initialisierung des Messdatendienstes wird durch einen SSH-Aufruf vorgenommen. Dieser wird aus den Konfigurationsinformationen des `mmServiceParameter`-Elements (vgl. Abschnitt A.3) abgeleitet. Dabei wird zusätzlich der Name des Arbeitsknotens übermittelt, mit dem der Messdatendienst zusammenarbeitet.

Nach dem erfolgreichen Start des Messdatendienstes kann nun die Messung gestartet werden. Dazu ruft der Arbeitsknoten per RMI (Remote Message Invocation)[5] die Funktion `runCommand("StartMeasurementCmd", new String[] {...})` auf. Dabei wird im String-Array der Klassenname des anzulegenden initialen Kontextes und alle Klassennamen der anzulegenden Beobachter übermittelt. Der Aufruf erzeugt eine Instanz der Klasse `StartMeasurementCmd` und führt auf ihr die Methode `execute()`

[4] http://java.sun.com/products/jndi/

[5] http://java.sun.com/products/jdk/rmi/ und http://de.wikipedia.org/wiki/Remote_Method_Invocation

aus (vgl. auch Abschnitt 5.1, Automatische Konfigurationsschritte und Abbildung 5.2).

Der initiale Kontext führt dazu, dass im vierten Schritt eine Instanz des initialen Kontextes angelegt wird.

Der initiale Kontext erzeugt im fünften Schritt als neue Instanz den Messdatensender, der dabei eine Verbindung zur JMS-Queue herstellt, die zuvor vom Arbeitsknoten zur Verfügung gestellt wurde. Nun ist der `InitialContext` in der Lage, Messdaten zu versenden.

Im sechsten Schritt werden nun alle Beobachter instanziiert, die bei der Anweisung, die Messung zu starten, angegeben wurden. Ihnen wird die im vierten Schritt angelegte Instanz des `InitialContext` übergeben, über die sie Messdaten erfassen können (vgl. auch Abschnitt 4.4.2).

Von nun an (siebter Schritt) ist es für die Beobachter möglich, sich an der Teilanwendung zu registrieren und Messdaten zu erfassen.

Wurde ein Messwert ermittelt, so muss der Beobachter ihn nur über den dafür vorgesehenen Ausführungskontext erfassen (Schritt acht).

Im neunten Schritt versendet der initiale Kontext von ihm selbst oder durch Unterkontexte erfasste Messwerte über den Messdatensender an die Messumgebung. Es kann hierbei entschieden werden, wann diese Übertragung stattfinden soll. Die Übertragung findet entweder nach Beendigung der Messung statt oder nachdem der initiale Kontext eine bestimmte Anzahl zu übertragender Nachrichten gesammelt hat. Dies lässt sich über den Parameter `messageCountToSend` in der Konfigurationsdatei der Messkonfigurationen einstellen. Ist der Parameter auf den Wert 0 eingestellt, so findet die Übertragung erst zum Ende der Messung statt. Ein Wert größer 0 zeigt an, wie viele Nachrichten gesammelt werden müssen, bevor eine Übertragung stattfindet. Somit signalisiert der Wert 1, dass die vorhandene Nachricht sofort gesendet werden soll, ohne weitere zu sammeln.

Im zehnten Schritt werden die zu übermittelnden Nachrichten durch den Messdatensender übertragen.

Danach werden durch den Messdatenerfasser im elften Schritt alle Nachrichten ausgepackt und mit Hilfe des Datenschreibers z. B. in einer Datenbank abgespeichert.

In den nachfolgenden drei Abschnitten werden die wichtigsten Komponenten, die zur Messdatenerfassung beitragen, genauer betrachtet. Da Beobachter zum Erfassen von Messdaten auf die Benutzung von Ausführungskontexten festgelegt sind, werden die Definition und die Generierung von Ausführungskontexten im nachfolgenden Abschnitt betrachtet. Die Übertragung von Messdaten erfolgt über fest vorgeschriebene Nachrichtenformate, die in Abschnitt 4.4.3 vorgestellt werden. Zum Schluss wird im Abschnitt 4.4.4 die Speicherung der Messdaten präzisiert.

4.4.2 Definition und Generierung von Ausführungskontexten

Neue Klassen von Ausführungskontexten können genau wie andere Komponenten des Frameworks durch Ableiten von der entsprechenden Oberklasse implementiert werden. Für Ausführungkontexte ist dies die Klasse `ExecutionContext` oder `InitialExecutionContext` (s. Abbildung 4.12).

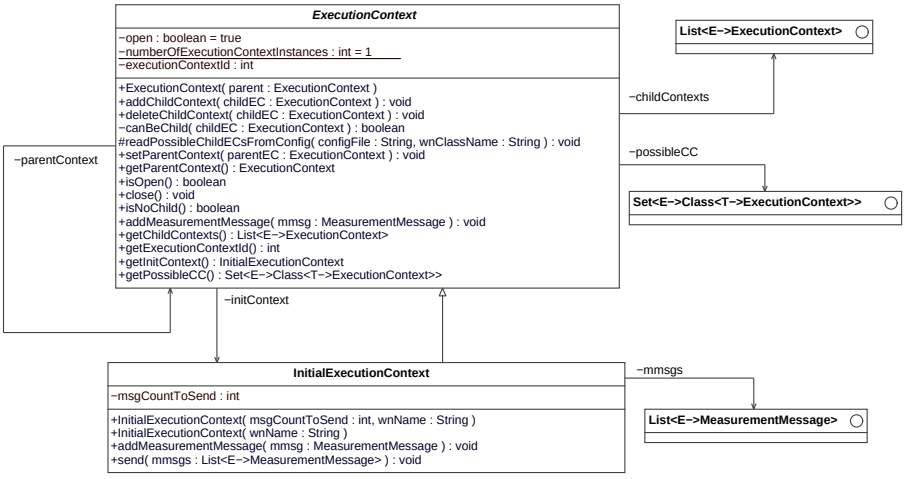

Abbildung 4.12: ExecutionContext und InitialExecutionContext

Da für die verschiedenen Teilanwendungen eines verteilten Systems evtl. sehr viele Ausführungskontexttypen auszuzeichnen sind, diese aber sehr ähnliche Funktionalität zur Verfügung stellen, lässt sich die Implementierung auch anhand der Beschreibung von Ausführungskontexten generieren. Die Beschreibung eines Ausführungskontextes wird gleichzeitig benötigt, um die erfassten Messdaten abzuspeichern (vgl. Abschnitt 4.4.4). Sie wird in der Konfigurationsdatei für Arbeitsknoten vorgenommen, weil sie für jede Instanz einer Klasse von Arbeitsknoten gleichermaßen gilt.

Beschreibung eines Ausführungskontext

Bei der Definition von Ausführungskontexten wird zwischen initialen Kontexten und allgemeinen Kontexten unterschieden, die keine initialen Kontexte darstellen. Die definierbaren Elemente sind jedoch jeweils gleich.

Für beide Kontextarten lassen sich jeweils der Name, die einmalig und periodisch erfassbaren Messwerte sowie die anlegbaren Unterkontexte definieren (vgl. Abschnitt 3.5.3). Darüber hinaus lässt sich angeben, ob die Beschreibung dazu genutzt werden soll, eine entsprechende Ausführungskontextklasse zu generieren.

Das nachfolgende Code-Beispiel zeigt die Auszeichnungen für einen initialen Kontext (`InitialContext`-Element) und einen allgemeinen Ausführungskontext (`Context-Def`-Element), wobei der allgemeine Kontext (`Transaction`) unter dem initialen Kontext angelegt werden darf.

```
<workingnode name="wn1" ...>
   <InitialContext id="icId"
                   generateClass="false"
                   class="InitialContext">
      <AllowableSubContexts>
         <ContextPermission idref="tcId"/>
      </AllowableSubContexts>
   </InitialContext>
```

```
   <ContextDef id="tcId"
               name="Transaction"
               generateClass="true">
      <MeasurementValue name="workingTime" javatype="float"
                        sqltype="decimal(10,2)"/>
      <MeasurementValue name="checkTime" javatype="java.util.Date"
                        sqltype="date"/>
      <PeriodicMeasurementValue name="cpuUsage"
                  javatype="String" sqltype="clob"
                  joinedStorage="true" separator=","/>
   </ContextDef>
</workingnode>
```

Die erlaubten Unterkontexte werden im Element `AllowableSubContexts` definiert. Es dürfen beliebig viele Berechtigungen durch die Aufnahme eines `ContextPermission`-Elements, welches über das Attribut `idref` einen Ausführungskontext referenziert, spezifiziert werden.

Ein einmalig in einer Instanz des Ausführungskontextes erfassbarer Wert wird über das Element `MeasurementValue` definiert. In ihm müssen stets die Attribute `name`, `javatype` und `sqltype` definiert sein.

Für die Beschreibung eines periodisch erfassbaren Messwerts muss das Element `PeriodicMeasurementValue` erstellt werden. Es muss die gleichen Attribute beinhalten wie ein `MeasurementValue`-Element. Zusätzlich kann für die Speicherung der Messdaten noch das Attribut `joinedStorage` und `separator` ausgezeichnet werden. Was diese Attribute bewirken, wird in Abschnitt 4.4.4 besprochen.

Mit Hilfe dieser Auszeichnungen ist es nun möglich, die Ausführungskontexte automatisch zu generieren.

Generierung der Ausführungskontexte

Die Generierung der Ausführungskontexte wird durch die in Abbildung 4.13 gezeigten Klassen vorgenommen. Ziel der Generierung ist es, aus den Beschreibungen der Kontexte Java-Quelltexte zu erzeugen, welche nachfolgend von einem Java-Compiler in Bytecode übersetzt werden können. Die Klasse `ContextGenerator` analysiert nach dem Aufruf der Methode `generateJavaClasses()` die Informationen aus der im Konstruktor übergebenen Konfigurationsdatei. Die im XML-Format vorliegende Datei wird durch einen XML-Parser eingelesen. Dazu bedient er sich der Klasse `WorkingNodeConfigCH`, die den `ContentHandler` eines SAX[6]-Parsers implementiert. Sie legt für jeden in der Konfigurationsdatei gefundenen Ausführungskontext und dessen Auszeichnungen eine neue Instanz der Klasse `ContextRep` an, die den zu generierenden Quelltext repräsentiert. Die Kontextrepräsentationen werden an den `ContextGenerator` übermittelt, der sie in einer Liste sammelt. Sind alle Kontexte gefunden, schreibt der `ContextGenerator` die Java-Quelltexte auf die Festplatte. Dabei generiert er den Quelltext über die Funktion `ContextRep.toJavaSource()`.

[6] http://de.wikipedia.org/wiki/JAXP

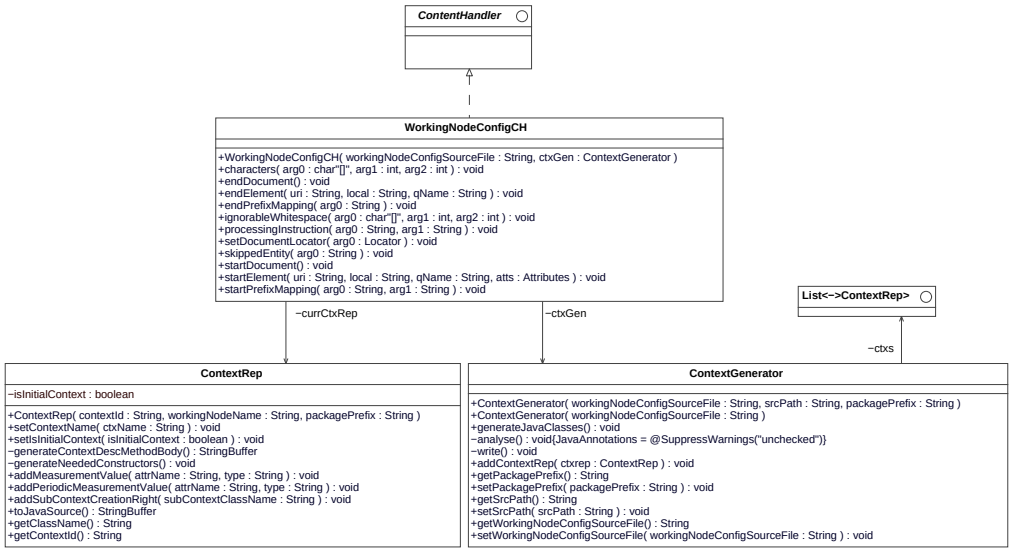

Abbildung 4.13: Die Klassen zur Generierung der Ausführungskontexte

4.4.3 Übertragung der Messdaten

In diesem Abschnitt werden die Implementierungen der verschiedenen benötigten Nachrichtentypen, des Datensenders und des Datenerfassers betrachtet. Danach wird in Abschnitt 4.4.4 die Speicherung der Daten durch einen Datenschreiber beschrieben.

Nachrichtentypen

Zur Übertragung von Messwerten an die Messumgebung werden die in Abschnitt 3.5.3 (Nachrichtenformate) vorgestellten Nachrichtentypen benötigt. Abbildung 4.14 zeigt die Klassen, welche die benötigten Nachrichtentypen implementieren.

Jede Nachricht übermittelt stets die eindeutige Identifikation `executionContextId` der Ausführungskontextinstanz, welche beim Anlegen eines Kontextes vergeben wird. Sobald ein Beobachter eine neue Ausführungskontextinstanz erzeugt, wird eine neue Instanz der Klasse `ContextNewMessage` erzeugt. Sie überträgt zusätzlich den Klassennamen des angelegten Kontextes (`contextClassName`). Anhand des Klassennamens können in der Messumgebung alle weiteren benötigten Informationen mit Hilfe der Beschreibung des Kontextes abgeleitet werden.

Erfasst der Beobachter einen neuen Messwert über den Kontext, wird eine neue Instanz der Klasse `MeasurementValueMessage` erzeugt. Sie enthält die `executionContextId`, den Wert des Messwertes (`mValue`), den Namen des Messwertes (`mValueName`), den Zeitpunkt, zu dem der Wert erfasst wurde (`mTimestamp`) sowie einen Index (`index`). Wenn ein einmalig zu erfassender Messwert zu übertragen ist, wird der Indexwert 0 versendet. Für periodisch erfassbare Messwerte wird der Index, beginnend bei Null für den ersten erfassten Wert, stets um eins erhöht, wenn ein erneuter Wert für ein Attribut erfasst wurde.

Wird ein Kontext durch einen Beobachter geschlossen, wird eine neue Instanz der Klasse `ContextCloseMessage` erzeugt. Nachdem diese Nachricht an die Messumgebung übermittelt wurde, kann die Messumgebung keine Messwerte mehr für diesen

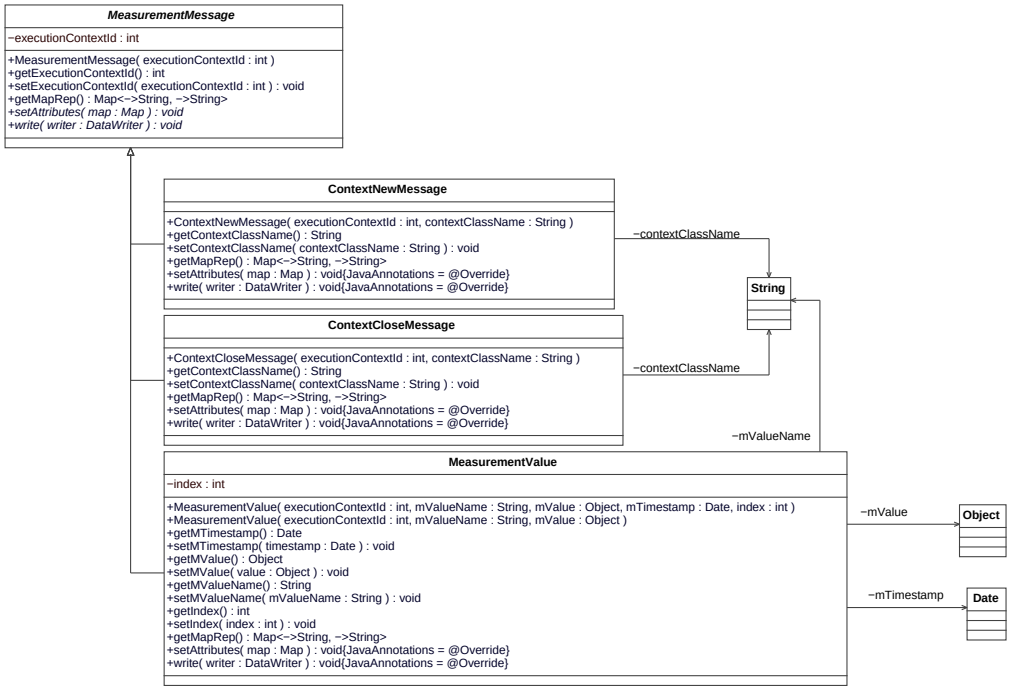

Abbildung 4.14: Die Nachrichten zum Versenden von Messdaten und Kontextinformationen

Kontext abspeichern. Es werden, wie bei der `ContextNewMessage`, nur die Informationen `executionContextId` und `contextClassName` übermittelt.

Die verschiedenen Nachrichtenformate werden serialisiert als `JMSMapMessage` übertragen. Aus diesem Grund wurden für jede `MeasurementMessage` die Methoden `getMapRep()` und `setAttributes(Map m)` implementiert. Die Methode `getMapRep()` schreibt die Werte der Nachricht in eine Hashtabelle, welche als Schlüssel und Werte nur Zeichenketten verarbeitet (`Map<String,String>`). Diese können später direkt als `JMSMapMessage`versendet werden. Dabei wird auch der volle Klassenname der Nachricht übertragen. Sobald die Messumgebung eine Nachricht in Form einer `JMSMapMessage` erhält, wird aus ihr der Typ extrahiert und mit Hilfe von Java-Reflection-Methoden wieder eine `MeasurementMessage` erzeugt. Die Attributwerte der Nachricht werden über die Methode `setAttriutes(Map m)` wieder hergestellt.

Nachdem die Nachricht übertragen wurde, kann sie von einem Datenschreiber verarbeitet und weggeschrieben werden. Da es nur sehr schwer möglich ist, die übertragene `MeasurementMessage` auf ihren korrekten Laufzeittyp mit Hilfe von Reflection-Methoden umzusetzen (*type casting*), wird die Polymorphie objektorientierter Sprachen ausgenutzt, indem die jeweils überschriebene Methode, `write(DataWriter d)` auf einer Nachricht aufgerufen wird. Diese ruft die korrekte Methode des Datenschreibers auf, die für die Handhabung der Nachricht verantwortlich ist

Messdatensender

Der Messdatensender `MeasurementDataSender` wird durch den initialen Ausführungskontext instanziiert, der von einem Messdatendienst angelegt wird. Da jeder Ausführungskontext eine Referenz auf den zugehörigen initialen Kontext besitzt,

kann dieser zum Versenden von Daten leicht erreicht werden. Das Versenden der Daten wird in der Klasse `InitialContext` von der Funktion `send(List<MeasurementMessage> mmsgs)` vorgenommen. Sie versendet eine Menge von Nachrichten an die Messumgebung und benutzt dafür die Methode `sendData(MeasurementMessage mmsg)` des Datensenders (s. Abbildung 4.15).

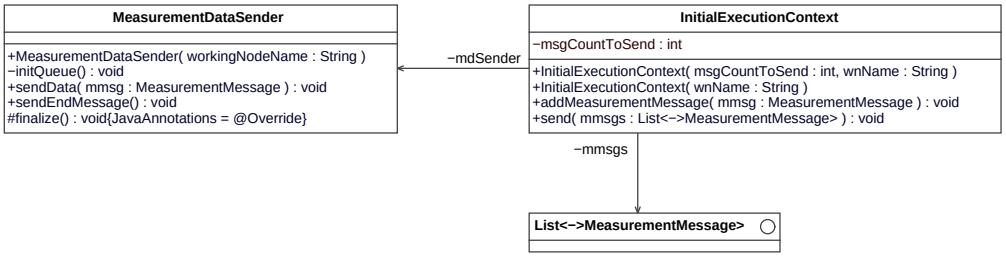

Abbildung 4.15: Der InitialContext versendet die Nachrichten über den Datensender.

Der Datensender erstellt mit der Funktion `initQueue` eine Verbindung zur Queue, die der Messdatenempfänger zur Verfügung gestellt hat. Die Queue wird über den Namen des Arbeitsknoten gefunden, der bei der Instanziierung im Konstruktor anzugeben ist. Er wird vom Messdatendienst an den initialen Kontext übergeben, der ihn an den Messdatensender weiterleitet.

Durch die Methode `sendEndMessage()` kann das Ende aller Übertragungen signalisiert werden. Dies muss nicht zwingend geschehen, weil keine Nachrichten mehr zu erwarten sind, sobald für den initialen Kontext, der immer den Identifikationswert 1 besitzt, eine `ContextCloseMessage` versendet wurde.

Messdatenerfasser

Der Arbeitsknoten instanziiert, wie schon in den Abbildungen 4.6 und 4.11 gezeigt, den Messdatenerfasser. Dieser verbindet sich dabei zum *openJMS*-Server und registriert eine neue Queue unter dem Namen des Arbeitsknotens, der um den Zusatz „_queue“ erweitert wird.

Sobald eine Nachricht vom Datensender in die Queue abgelegt wird, wird sie vom Messdatenerfasser verarbeitet, bei dem automatisch die Funktion `onMessage(Message m)` aufgerufen wird, da er die Schnittstelle `MessageListner` implementiert. Somit ist er in der Lage, alle Nachrichtentypen, die der Java Messaging Service bereitstellt, zu empfangen. Es werden zurzeit jedoch nur `JMSMapMessages` versendet.

Abbildung 4.16 zeigt die Schnittstelle des Datenerfassers.

4.4.4 Generische Speicherung der Messdaten

Die in Abbildung 4.17 dargestellten Klassen sind für die Speicherung der Messdaten verantwortlich. Aus den vorhergehenden Abschnitten ist bereits bekannt, wie die Messdaten zum Datenschreiber `DataWriter` gelangen.

Die Standardmethode, Messdaten abzuspeichern, wird durch den `DefaultDataWriter` definiert, der die abstrakte Klasse `SQLDataWriter` erweitert und somit auch das Interface `DataWriter` implementiert. Die Klasse `SQLDataWriter` stellt im Vergleich

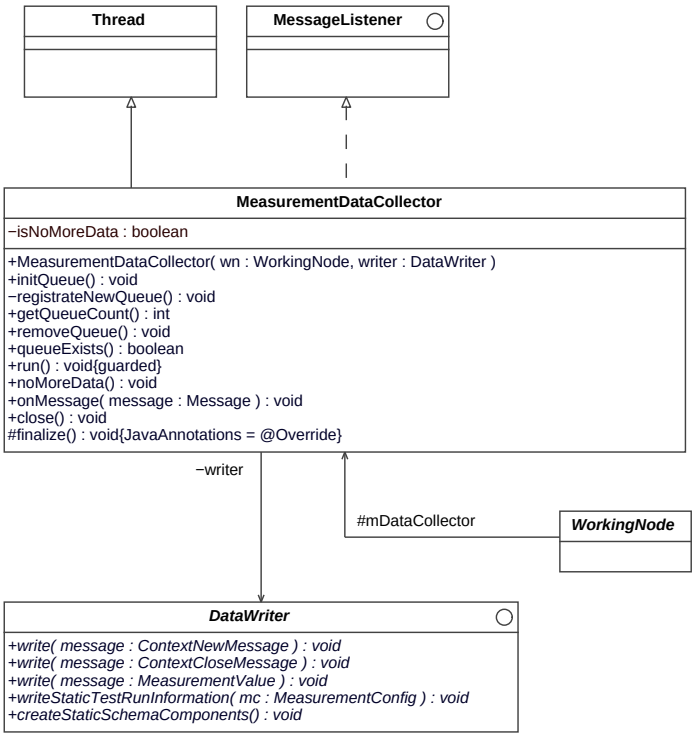

Abbildung 4.16: Der Datenerfasser verwendet einen Datenschreiber zum Speichern der Messwerte.

zum Interface `DataWriter` die Möglichkeit zur Verfügung, eine Verbindung zur Datenbank herzustellen.

Bevor Daten auf die Datenbank geschrieben werden können und der Messlauf gestartet wird, muss der Datenschreiber die Datenbank vorbereiten. Zum einen muss er sicherstellen, dass die statischen Elemente des verwendeten Schemas angelegt sind und zum anderen schreibt er wichtige Konfigurationsdaten, die den Aufbau der Messkonfiguration widerspiegeln, in die Datenbank.

Die Klasse `DefaultDataWriterSchema` repräsentiert die Umsetzung der in Abschnitt 3.5.4 vorgestellten Modellierung eines sich selbst erweiternden Datenbankschemas. Es besteht aus Relationen, die statisch angelegt werden und Relationen, die dynamisch – je nach übermitteltem Ausführungskontexttyp – das Schema erweitern.

Im Folgenden werden zunächst die statischen Teile des Schemas vorgestellt, die vor dem Messlauf in der Datenbank angelegt sein müssen. Daher werden sie bereits während des Startvorganges der Messumgebung angelegt. Falls diese Teile des Schemas schon angelegt wurden, kann das Anlegen des Schemas übersprungen werden (vgl. Abschnitt 4.3).

Statische Schemakomponenten

Um grundlegende Informationen über den Aufbau der ablaufenden bzw. abgelaufenen Messung speichern zu können, wurden die folgenden Relationen gebildet. Sie wurden aus der in Abschnitt 3.5.4 vorgestellten Modellierung abgeleitet. Die benötigten DDL-Definitionen sind in Abschnitt B.1 aufgeführt.

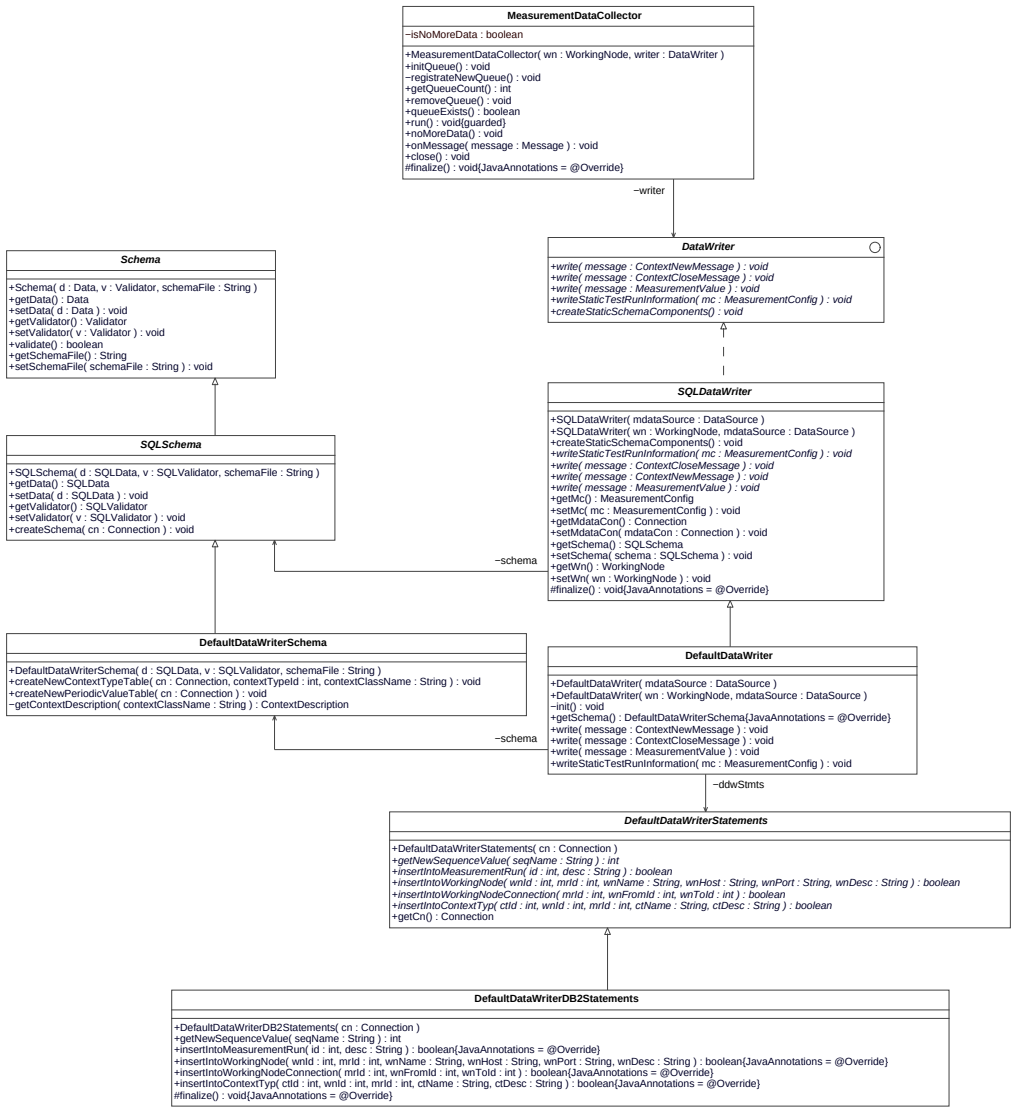

Abbildung 4.17: Als DataWriter wird der DefaultDataWriter verwendet, der das DefaultDataWriterSchema verwendet.

- `MeasurementRun: MR(mrId, Startzeitpunkt, Beschreibung)`
  mit: `mrId` als Primärschlüssel

- `WorkingNode: WN(wnId, mrId, Name, Host, Port, Beschreibung)`
  mit: `(wnId, mrId)` als Primärschlüssel und
  `mrId` als Fremdschlüssel auf `MR`

- `WorkingNodeConnection: WNC(mrId, wnFromId, wnToId)`
  mit: `(mrId, wnFromId, wnToId)` als Primärschlüssel,
  `(mrId, wnFromId)` als Fremdschlüssel auf `WN` und
  `(mrId, wnToId)` als Fremdschlüssel auf `WN`

- `ExecutionContextTyp: CT (ctId, wnId, mrId, Name, Beschreibung)`
  mit: `ctId` als Primärschlüssel,
  `wnId` als Fremdschlüssel auf `WN` und
  `mrId` als Fremdschlüssel auf `MR`

Bevor die Messung gestartet wird, werden die Informationen über den aktuell ablaufenden Messablauf in das Schema aufgenommen. Dabei wird dem gerade ausgeführten Messlauf von der Datenbank eine eindeutiger Identifikator zugewiesen, die in der Klasse `MeasurementConfig` im Attribut `currentRunIdentification` hinterlegt wird (s. Abbildung 4.2).

Die in der Messkonfiguration hinterlegten Arbeitsknoten sowie ihre Beziehungen untereinander werden in die Datenbank mit aufgenommen, um die Struktur des verteilten Systems während einer Auswertung reproduzieren zu können. Darüber hinaus werden die von den Arbeitsknoten anlegbaren Ausführungskontexttypen hinterlegt.

Für die Messwerte, die über die verschiedenen Ausführungskontexttypen erfasst werden, werden Tabellen erzeugt, die dynamisch je nach übermitteltem Ausführungskontexttyp angelegt werden.

Dynamische Schemakomponenten

Die dynamischen erzeugten Schemakomponenten erfassen die Messinformationen, die während der Messung an die Messumgebung übermittelt werden. Im Nachfolgenden wird betrachtet, welche Informationen jeweils für die einzelnen Nachrichtentypen abgespeichert werden.

Übermittelt ein Messdatensender eine `ContextNewMessage`, muss für den darin enthaltenen Ausführungskontexttyp eine Tabelle bestehen, in welche die Werte, die mit nachfolgenden `MeasurementValueMessages` übermittelt werden, abgespeichert werden können.

Liegt für den Kontexttyp noch keine Tabelle im Schema vor, wird sie angelegt. Dazu wird anhand der Beschreibung des Kontexttyps (vgl. Code-Beispiel aus Abschnitt 4.4.2) für jeden `MeasurementValue` eine neue Spalte erzeugt. Für einen `PeriodicMeasurementValue` wird auch eine eigene Spalte in der Tabelle erzeugt, jedoch lässt sich hier unterscheiden, ob die periodisch erfassten Werte gebündelt in dieser Spalte oder in einer neu anzulegenden Tabelle über eine Primär-Fremdschlüsselbeziehung abgespeichert werden. Diese Unterscheidung wird anhand des Attributes `joinedStorage` vorgenommen. Hat das Attribut den Wert `false`, so wird eine neue Tabelle zur Aufnahme der periodischen Werte erzeugt. Ist das Attribut auf `true` gesetzt, werden die Werte in eine einzelne Spalte geschrieben und mit dem in Attribut `separator` ausgezeichneten Trennzeichen voneinander getrennt.

Sobald die Tabelle im Schema vorhanden ist, wird für den in der `ContextNewMessage` übermittelten Ausführungskontexttyp ein neues Tupel in der zugehörigen Tabelle angelegt. Dabei wird für den übermittelten Ausführungskontext ein neuer Primärschlüsselwert von der Datenbank vergeben, die über eine Datenbanksequenz erzeugt wird. Der Datenerfasser merkt sich die Zuordnung `executionContextId`→*Datenbankidentifikator*, um nachfolgende Messwerte richtig zuordnen zu können.

Werden nach der Übertragung einer `ContextNewMessage` Messwerte mit Hilfe von `MeasurementValueMessages` übermittelt, werden diese in den zugehörigen Spalten des entsprechenden vorher angelegten Tupels, mit Hilfe von Update-Statements, gespeichert. Bei ausgelagerter Speicherung von periodisch erfassten Werten, werden die Teilergebinisse durch Insert-Statements in die Detailtabelle geschrieben.

Sobald für einen Ausführungskontext eine `ContextCloseMessage` übertragen wird, verwirft der Datenschreiber die Zuordnung `executionContextId`→*Datenbankidentifikator*.

Die verwendeten Templates zur Erstellung der dynamisch erzeugten Schemaobjekte können im Abschnitt B.2 eingesehen werden.

5. Implementierung der Frameworkkomponenten für den ACCache

In diesem Kapitel wird anhand des ACCache-Systems gezeigt, wie das Framework dazu genutzt werden kann, verteilte Teilanwendungen zu vermessen. Zu diesem Zweck werden, aufbauend auf der Implementierung des Frameworks, die zur Vermessung verschiedener Teilanwendungen notwendigen Implementierungsschritte aufgezeigt.

Das Funktionieren des Messablaufes wird dabei an einem kleinen Szenario mit nur einem simulierten Client gezeigt, welcher die Arbeitslast für ein ACCache-System definiert. Dabei wird auf dem ACCache nur die Bearbeitungszeit der Anfragen erfasst, an die Messumgebung übermittelt und abgespeichert.

Der erste Implementierungsschritt besteht darin, dem Framework mitzuteilen, welche verschiedenen Arten von Teilanwendungen zu vermessen sind. Für diese wird ein neuer Arbeitsknoten angelegt.

5.1 Definition der Arbeitsknoten

Zur Vermessung des ACCache-Systems spielen nur 3 Teilanwendungen eine Rolle: die Client-Anwendung, das ACCache-System und das Backend-Datenbanksystem.

Die Client-Anwendungen sollen durch das Framework simuliert werden. Daher ist es nicht notwendig, einen Arbeitsknoten für eine bestimmte Client-Anwendung auszuzeichnen. Es genügt die Spezialisierung eines `SimulatedClient`-Arbeitsknotens. Die Klasse `SimulatedClient` könnte auch direkt verwendet werden, um die Client-Anwendung zu repräsentieren. Jedoch führt diese nur allgemeine Arbeitsschritte aus, was bedeutet, dass in ihr auch Arbeitschritte aufgenommen werden können, die ein ACCache-System oder eine Backend-Datenbank nicht ausführen kann. Um dies zu vermeiden, wurde die Klasse `JDBCClient` als Unterklasse von `SimulatedClient` implementiert (s. Abbildung 5.1). Sie kann nur JDBC-Transaktionen als Arbeitsschritte ausführen.

Für das ACCache-System wurde die Klasse `ACCacheWorkingNode` und für das DB2-Datenbanksystem der Arbeitsknoten `DB2WorkingNode` als Unterklasse von `WorkingNode` angelegt. Sie erweitern die Funktionalität der Klasse `WorkingNode` nicht. Ihre Definition ist aber notwendig, um dem Framework einen neuen Typ von Arbeitsknoten bekannt zu machen.

Alle vorgestellten speziellen Arbeitsknoten (s. Abbildung 5.1) wurden im Paket `mm.accache.wn.nodes` implementiert. Wenn in der Zukunft verschiedenste andere verteilte Systeme mit Hilfe des Frameworks implementiert werden, empfiehlt sich eventuell eine andere Aufteilung. So könnte z. B. die Klasse `JDBCClient` als fester Bestandteil des Frameworks aufgenommen werden, weil sie eventuell für mehrere Systeme einsetzbar ist und die Klasse `DB2WorkingNode` könnte in eine eigene Pakethierarchie, z. B. `mm.db2`, aufgenommen werden.

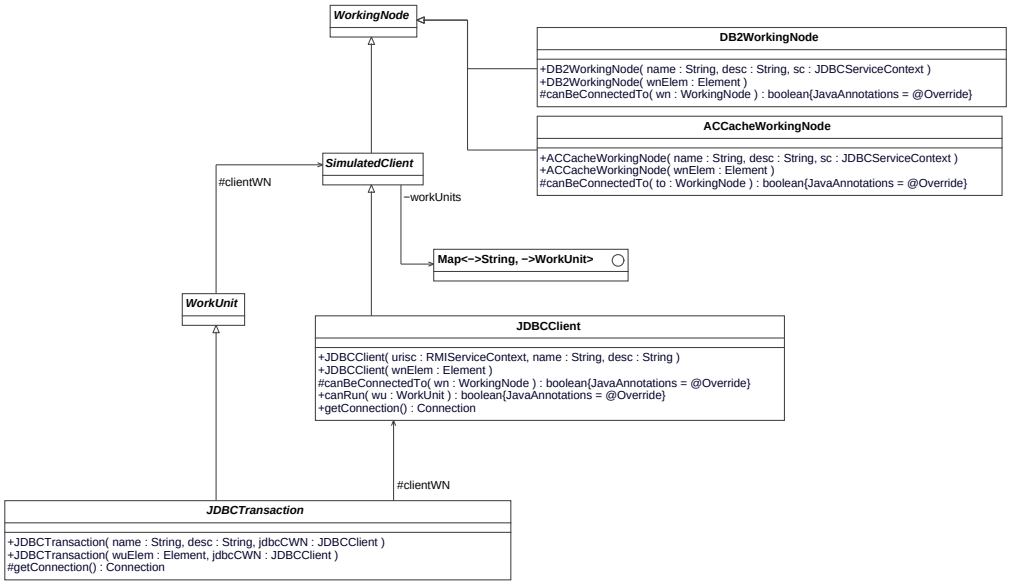

Abbildung 5.1: Die verschiedenen Spezialisierungen der Arbeitsknoten

Durch die Definition der Arbeitsknoten können diese nun in den Konfigurationsdateien verwendet werden. In der Folge wird anhand der Eigenschaften, die ein Arbeitsknoten abbildet, entschieden, ob Komponenten des Frameworks spezialisiert bzw. implementiert werden müssen.

Damit festgelegt werden kann, über welche Schnittstelle ein Arbeitsknoten aufgerufen werden kann, müssen diese nun definiert werden.

Aufrufschnittstellen

Der Arbeitsknoten `SimulatedClient` lässt sich über eine RMI-Schnittstelle ansprechen. Zurzeit wird dies mit Hilfe des Messdienstes gelöst, der einen `SimulatedClient` auf Wunsch instanziiert und mit Beginn der Messung startet. Da die Klasse `RMIServiceContext` (vgl. Abschnitt 4.1.1, Abbildung der Aufrufbarkeit) die notwendigen Auszeichnungen bereits zur Verfügung stellt, ist das Anlegen einer neuen Unterklasse von `ServiceContext` nicht notwendig. Ebenso verhält es sich für die Arbeitsknoten `DB2WorkingNode` und `ACCacheWorkingNode`, welche über JDBC (`JDBCServiceContext`) angesprochen werden.

Somit ist die Eigenschaft der Aufrufbarkeit überprüft und kann für alle Arbeitsknoten zufriedenstellend abgebildet werden.

Messdatenerfassung

Für die Erfassung bzw. Speicherung der Messdaten wird die Standardmethode verwendet. Es sind keine neuen Komponenten anzulegen. Allerdings bereitet es kein Problem, bei Bedarf zusätzliche Komponenten einzubinden. So wäre es z. B. möglich gewesen, einen speziellen Datenschreiber zu implementieren.

Beobachter

Für jeden der drei angelegten Arbeitsknotentypen können Beobachterkomponenten implementiert werden, für die dann auch zu entscheiden ist, welche Ausführungskontexte sie zur Erfassung ihrer Daten verwenden sollen. Dies stellt eine weitere sehr wichtige Designentscheidung dar. Im Rahmen der Diplomarbeit wurde nur eine Beobachterkomponente implementiert, welche die grundlegende Funktion der Messdatenerfassung sicherstellt (vgl. Abschnitt 5.2.4). Es ist jedoch später notwendig, für jeden der drei Arbeitsknoten Beobachterkomponenten zu definieren und zu implementieren.

Die Entscheidung, wie viele Beobachterkomponenten ausgezeichnet werden, sollte von der Beantwortung folgender Fragen abhängig gemacht werden:

1. Wie viele Bereiche einer Teilanwendung lassen sich unabhängig voneinander beobachten?

   Die Unabhängigkeit zweier Bereiche einer Teilanwendung lässt sich daran erkennen, dass jeder Teilbereich durch je einen Beobachter überwacht werden kann, ohne dass einer der Beobachter mit einem anderen Beobachter eine gemeinsam verwendete Ausführungskontextinstanz teilt. Für das ACCache-System könnte man z. B. einen Beobachter erstellen, der die Nachladevorgänge beobachtet und einen der die Durchführung der Sondierung beobachtet. Je nachdem, wie nun die Erfassung der Messdaten über die Ausführungskontexte gestaltet ist, können die beiden Beobachter unabhängig oder abhängig voneinander sein. Die Durchführung der Sondierung ist sicherlich im Kontext einer Anfrage zu betrachten. Eine Anfrage ist aber auch dafür verantwortlich, dass Nachladevorgänge stattfinden. Wird dies durch den Beobachter der Nachladevorgänge erfasst, muss er die Möglichkeit besitzen, die gleiche Ausführungskontextinstanz einer Anfrage zu benutzen. Wird ein Nachladevorgang unabhängig von der auslösenden Anfrage betrachtet, sind die beiden Beobachter unabhängig voneinander.

   Die Bildung von Beobachterkomponenten, die abhängig voneinander sind, sollte wenn möglich vermieden werden, da diese einen zusätzlichen Mechanismus benötigen, um gemeinsam auf gleiche Ausführungskontextinstanzen zuzugreifen.

2. Welche Teile der Beobachtung werden nur selten benötigt und welche sehr häufig?

Ein einzelner Beobachter lässt sich sehr leicht aktivieren bzw. deaktivieren (vgl. Abschnitt 4.1.1, Abbildung der Beobachtbarkeit). Daher ist es ein Vorteil, mehrere Beobachterkomponenten anzulegen, so dass sich Bereiche einer Teilanwendung von der Beobachtung ausblenden lassen. Damit die Beobachtung nicht zu sehr das Messergebnis beeinflusst, sollte man auch stets darauf achten, nur die Teile einer Anwendung zu beobachten, welche für die aktuell ablaufende Messung relevant sind.

3. Welche Bereiche der Teilanwendung sollen bzw. müssen für sich alleine betrachtet werden?

   Wenn bestimmte Bereiche einer Teilanwendung abgeschlossene Einheiten darstellen, die nacheinander vermessen werden können, ist es auch sinnvoll, mehrere Beobachterkomponenten für diese einzusetzen. Die Beobachterkomponenten müssen dann nicht unbedingt unabhängig voneinander sein, da sie nacheinander und nicht zusammen eingesetzt werden. Für das Beispiel, welches in Frage 1 vorgestellt wurde, bedeutet dies, dass die Sondierung nach der Beobachtung der Nachladevorgänge in einer erneut durchgeführten Messung stattfindet und beide Beobachter ihre Ergebnisse im Kontext einer Anfrage ablegen.

Automatische Konfigurationsschritte

Um die automatische Konfigurierbarkeit der einzelnen Arbeitsknoten sicherzustellen, muss analysiert werden, ob zusätzliche Konfigurationsschritte implementiert werden müssen.

Für den simulierten Client sind keine neuen Konfigurationsschritte ins Framework aufzunehmen. Er wird durch das Kommando `StartSimulatedClient` des MeasurementService zur Ausführung gebracht. In Abbildung 5.2 sind die Schnittstelle des MeasurementService und die beiden zur Zeit ausführbaren Kommandos `StartSimulatedClientCmd` und `StartMeasurementCmd` dargestellt (vgl. auch Abschnitt 4.4.1, Schritt 2 und 3).

Zum automatischen Konfigurieren des DB2-Arbeitsknotens sind die gleichen Schritte wie beim Starten des Messumgebung notwendig. Das Starten der Datenbankinstanz kann von der Implementierung der Klasse `StartDB` übernommen werden. Hierzu müssen die Konfigurationsinformationen für diesen Konfigurationsschritt in einer eigenen Datei zur Verfügung gestellt werden.

Das Anlegen einer neuen Datenbank und das Anlegen eines Schemas können ebenso durch die Funktionalität der bereits vorgestellten Konfigurationsschritte `CreateDB` und `CreateSchema` übernommen werden. Auch hier sind die Konfigurationsinformationen über eine andere Quelle zu beziehen.

Zum Starten eines ACCache-Systems sind dessen Startparameter und die zu verwendende Cache-Group-Definition zu übergeben. Damit das Starten automatisch durchgeführt werden kann, wurde ein eigenständiger Konfigurationsschritt `StartACCache` implementiert. Dieser startet das ACCache-System über die Ausführung eines SSH-Aufrufes.

Damit dies jedoch funktioniert, muss die Datenbank, die vom ACCache-System zur Ablage und Verwaltung der Cache Group eingesetzt wird, initialisiert sein. Dies

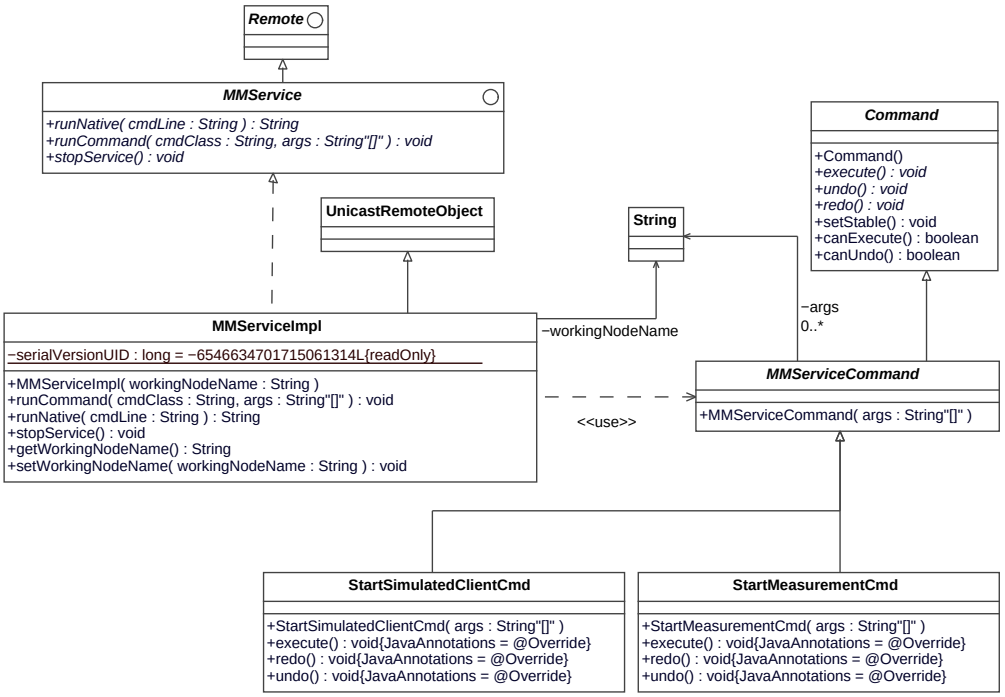

Abbildung 5.2: Der Measurement Service führt den SimulatedClient auf einem entfernten Host aus.

bedeutet wiederum, dass das Starten der Datenbankinstanz und das Anlegen einer Datenbank notwendig ist.

Im Rahmen der Diplomarbeit wurde eine automatische Konfigurierung der Arbeitsknoten noch nicht vollständig vorgenommen. Die benötigten Funktionalitäten stehen jedoch zur Verfügung. Die zusätzlichen Konfigurationsinformationen, die für jede Instanz und jeden Messlauf unterschiedlich sein können, sollten in die Konfigurationsdatei für Messkonfigurationen aufgenommen werden.

5.2 Exemplarischer Messlauf

Der nun im Folgenden kurz vorgestellte exemplarische Messlauf soll vor allem den Ablauf der Messdatenerfassung testen, wie er in Abschnitt 4.4.1 vorgestellt wurde.

Dazu werden zunächst der Aufbau unseres verteilten Testsystems und die dazu notwendigen Konfigurationsinformationen vorgestellt. Danach wird eine Arbeitslast, bestehend aus zwei verschiedenen Anfragen, definiert, die der Scheduler des simulierten Clients im Verhältnis 60% zu 40% einplant und über das ACCache-System ausführt. Dazu werden die Daten und das Schema der prototypischen Implementierung des ACCache-Systems von Christian Merker benutzt, welches Darsteller und Filme in einer Filmdatenbank speichert [Mer05].

Die Beobachtung des ACCache-Systems beschränkt sich auf die Erfassung der Bearbeitungszeit einer Anfrage auf dem ACCache-Systems. Der simulierte Client und die DB2-Datenbank werden nicht beobachtet. Aus diesem Grund ist die Zahl der zu generierenden Ausführungskontexte auf den Kontext *Anfrage* und einen initialen Kontext für das ACCache-System beschränkt.

Zum Schluss werden die erfassten Messwerte noch kurz dargestellt, welche aber nur sehr beschränkt aussagekräftig sind. Zumindest lässt sich hier eine deutliche Leistungssteigerung bei wiederholter Anfragestellung feststellen, da eine sich wiederholende Anfrage dann direkt auf dem Cache ausgeführt werden kann.

5.2.1 Aufbau des zu vermessenden verteilten Systems

Das für den exemplarischen Messlauf zu vermessende verteilte System besteht aus einem simulierten Client-Instanz, einer Instanz des ACCache-Systems und einer Datenbankinstanz. Der simulierte Client führt seine Arbeitsschritte über die JDBC-Schnittstelle des ACCache-Systems aus, welches wiederum auf die Backend-Datenbank zugreift (s. Abbildung 5.3). Beobachtet wird dabei nur das ACCache-System, auf dem die Bearbeitungszeit der ausgeführten Anfragen gemessen wird. Für den DB2-Arbeitsknoten wurde kein `MeasurementService` instanziiert, da er nicht beobachtet wird und auch keine entfernt zu instanziierenden Objekte benötigt.

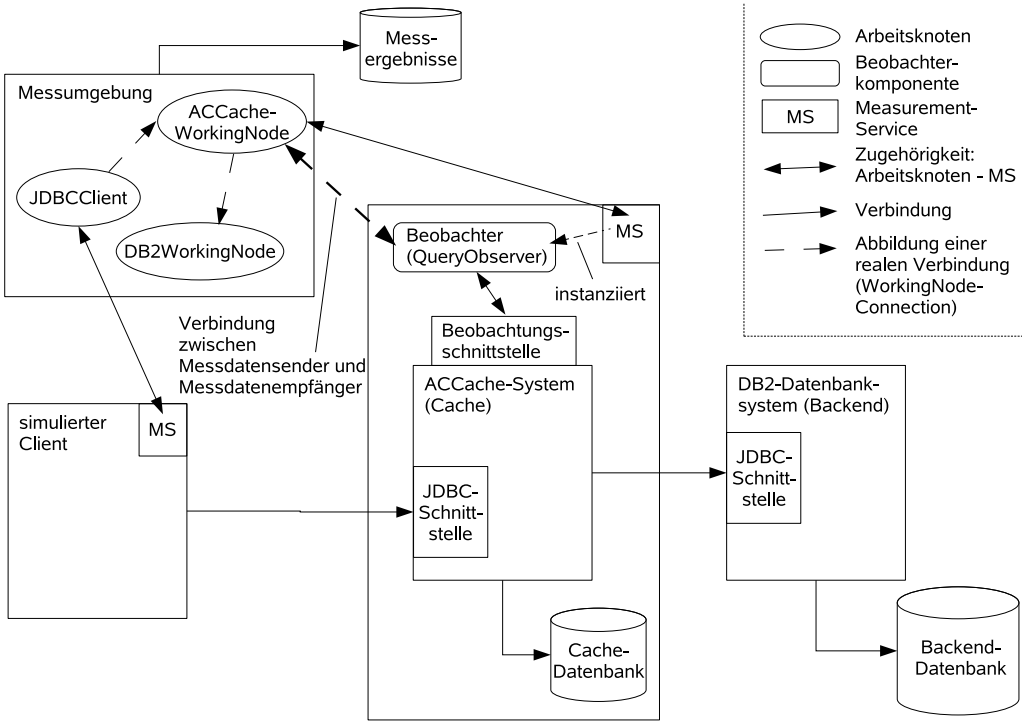

Abbildung 5.3: Der Aufbau des verteilten Systems für den exemplarischen Messlauf.

Alle Konfigurationen, die zur Durchführung des exemplarischen Messlaufes notwendig waren, sind in Abschnitt C aufgelistet. Sie umfassen nur einen beschränkten Teil der Möglichkeiten, die in Abschnitt A beispielhaft und zum Testen des Frameworks aufgeführt sind. Der in Abbildung 5.3 dargestellte Aufbau wird durch die in Abschnitt C.2 vorgestellten Definitionen spezifiziert.

5.2.2 Definition der auszuführenden Arbeitslast

Die vom `JDBCClient` auszuführende Arbeitslast wird durch die beiden JDBC-Transaktionen `SelectAnActor` und `SelectAllDirectors` abgebildet (s. Abbildung 5.4). Beide Klassen führen in ihrer Transaktion nur eine Anfrage aus.

Die Klasse `SelectAnActor` selektiert einen Darsteller aus der Tabelle aller Darsteller durch die Anfrage „`select * from darsteller where id = ?`“. Welcher Darsteller selektiert werden soll, kann über die Methode `setActorId()` festgelegt werden. Für den exemplarischen Messlauf wird pro Ausführung eine zufällige Zahl zwischen 1 und 100 frei gewählt, da die Beispieldatenbank genau 100 Darsteller verwaltet.

Die Klasse `SelectAllDirectors` führt stets die Anfrage „`select * from regisseur`“ aus, was zu Beginn der Messung zu einem hohen Nachladeaufkommen führen soll. Danach sollte die Anfrage stets auf dem Cache durchführbar sein.

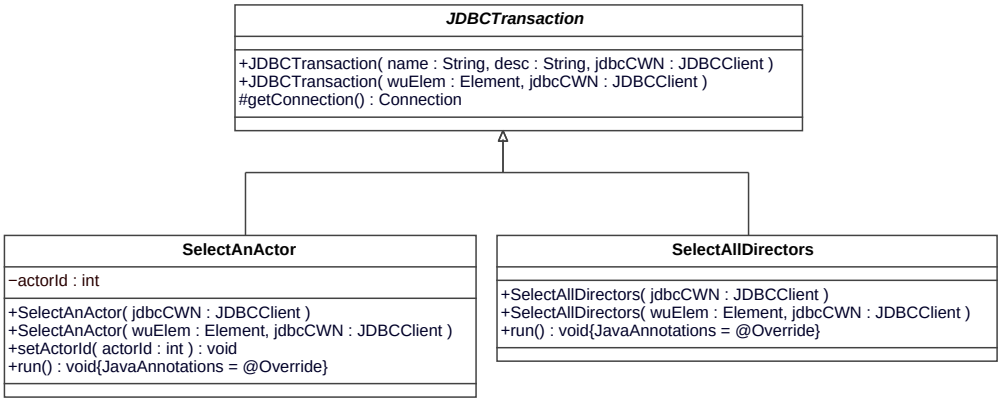

Abbildung 5.4: Die beiden Anfragen werden durch zwei JDBC-Transaktionen definiert.

5.2.3 Definition und Generierung von Ausführungskontexten

Wie bereits in Abschnitt 5.2 kurz erwähnt benötigen wir für den exemplarischen Messlauf nur zwei Ausführungskontexte. Da jede Messung einen initialen Kontext braucht, muss auch für für den exemplarischen Messlauf ein initialer Kontext ausgezeichnet werden. Dieser soll jedoch nur Anfragen als Unterkontexte zulassen, welche den zweiten zu definierenden Kontext darstellen. Der Ausführungskontext *Anfrage* soll den Anfragetext und die Bearbeitungszeit einer durchgeführten Anfrage erfassen. Abbildung 5.5 zeigt die für den exemplarischen Messlauf vorgenommene Modellierung.

Damit die Messumgebung die benötigten Kontexte generieren kann, muss deren Beschreibung vorliegen. In Abschnitt C.1 wurden die Beschreibungen der Ausführungskontexte für den ACCache-Arbeitsknoten hinterlegt. Nach dem Starten der Generierung für Ausführungskontexte wurden die beiden folgenden Klassen erzeugt:

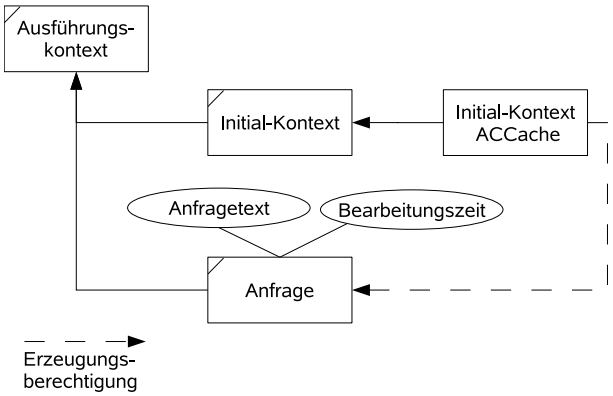

Abbildung 5.5: Die Modellierung der Ausführungskontexte für den exemplarischen Messlauf

- Der initiale Kontext:

```
package mm.accache.wn.context;

import mm.self.wn.context.*;
import mm.self.wn.context.msg.*;

import mm.exceptions.InitializationException;

import java.util.*;

public class ACCacheWorkingNodeInitialContext
       extends InitialExecutionContext {

   public ACCacheWorkingNodeInitialContext(String wnName)
      throws InitializationException {
      super(wnName);
   }

   public ACCacheWorkingNodeInitialContext(int msgCountToSend,
                                           String wnName)
      throws InitializationException {
      super(msgCountToSend, wnName);
   }

   public ACCacheWorkingNodeQuery createACCacheWorkingNodeQuery() {
      return new ACCacheWorkingNodeQuery(this);
   }
}
```

- Der Anfragekontext:

```
package mm.accache.wn.context;

import mm.self.wn.context.*;
import mm.self.wn.context.msg.*;
import java.util.*;

public class ACCacheWorkingNodeQuery extends ExecutionContext {
   public ACCacheWorkingNodeQuery(ExecutionContext parent) {
      super (parent);
   }

   public void gatherWorkingTime(float workingTime) {
      this.getInitContext().addMeasurementMessage(
         new MeasurementValue (this.getExecutionContextId(),
                               "workingTime",
                               workingTime));
   }

   public void gatherQueryText(String queryText) {
      this.getInitContext().addMeasurementMessage(
         new MeasurementValue (this.getExecutionContextId(),
                               "queryText",
                               queryText));
   }
}
```

Nachdem dem Beobachter vom Messdatendient der `ACCacheWorkingNodeInitialContext` übergeben wurde, kann über diesen lediglich ein `ACCacheWorkingNodeQuery`-Kontext angelegt werden. Dieser wiederum kann keine Unterkontexte anlegen. Er verfügt lediglich über die Funktionen `gatherWorkingTime` und `gatherQueryText`, um die Messdaten zu erfassen. Das Erfassen eines Messwertes führt dazu, dass eine neue `MeasurementMessage` mit dem erfassten Messwert im initialen Kontext erfasst wird.

5.2.4 Implementierung eines Beobachters

Für den ACCache-Arbeitsknoten wurde der Beobachter `ACCacheQueryObserver` in der Konfiguration bekannt gemacht (vgl. Abschnitt C.1). Der Messdatendienst wird eine Instanz dieses Beobachters beim Starten der Messung anlegen. Damit der Beobachter jedoch vom ACCache-System Informationen über den Status der abgearbeiteten Anfragen erhält, muss er sich als solcher beim ACCache-System registrieren.

Das ACCache-System sieht vor einzelne Teilebereiche der Anwendung, mit Hilfe von `JobObserver`-Implementierungen, beobachten zu können. Es verwaltet hierzu verschiedene Job-Typen, an denen sich Job-Beobachter registrieren können, um den Ablauf des Jobs zu überwachen. Darüber hinaus lassen sich `JobObserver` auch direkt

an einzelnen Komponenten des ACCache-Systems registrieren, so auch an `QueryWorker`- und `QueryWorkerManager`-Instanzen, welche für die Anfragebeantwortung verantwortlich sind.

Die Möglichkeit, das ACCache-System über `JobObserver` zu beobachten, wurde durch Implementierung eines `RemoteOperatingJobObserver` ausgenutzt. Mit Hilfe einer ihm übergebenen RMI-URL verbindet er sich mit einem entfernt ablaufenden `RemoteObserver`, indem er eine Referenz auf diesen erzeugt.

Damit der `ACCacheQueryObserver` Informationen von einem `RemoteOperatingJobObserver` erhalten kann, muss er einen `RemoteObserver` zur Verfügung stellen. Die Klasse `ACCacheQueryObserver` kann diese Aufgabe selbst nicht übernehmen, da sie von der Klasse `MeasurementObserver` erbt und jede `RemoteObserver`-Implementierung von der Klasse `UnicastRemoteObject` erben muss (RMI-Voraussetzung). Aus diesem Grund wurde die Klasse `ACCacheObserverHelper` erzeugt, die den benötigten `RemoteObserver` darstellt.

Eine Instanz der Klasse `ACCacheObserverHelper` registriert sich bei der Erzeugung an einer RMI-Registry und mit Hilfe der Funktion `registrateRemoteObserver` der Klasse `DriverManager` am ACCache-System. Beim Aufruf der Funktion `registrateRemoteObserver` wird die RMI-URL zum Erreichen des `QueryManager` und die RMI-URL zum Erreichen des `RemoteObserver` übergeben. Dies erlaubt es dem `DriverManager`, über die Funktion `addJobObserver(String rmiUrlToJobObs)` des `QueryWorkerManager`, einen `RemoteOperationJobObserver` zu erzeugen.

Der über diesen Weg instanziierte `RemoteOperationJobObserver` wird bei jeder Anforderung eines `QueryWorker` über die JDBC-Schnittstelle vom Manager übergeben, so dass es möglich ist, eine Anfragebearbeitung zu beobachten.

Die vom `RemoteOperationJobObserver` gesammelten Informationen werden, über die zur Verfügung gestellten Update-Operationen der verschiedenen Beobachter, an den `ACCacheQueryObserver` übertragen.

Abbildung 5.6 gibt einen Überblick über die an der Beobachtung des ACCache-System beteiligten Klassen.

Nachdem der `ACCacheQueryObserver` Messwerte vom `RemoteOperationJobObserver` erhalten hat, legt er, je nach übergebener Nachricht, neue `ACCacheWorkingNodeQuery`-Kontexte an, um die Messwerte darüber zu erfassen. Der `RemoteOperationJobObserver` versendet dazu stets einen eindeutigen Identifikationswert für den beobachteten `QueryWorker`, den Namen des erfassten Messwertes und den Messwert selbst. Sobald von einem `QueryWorker` die zwei erwarteten Messwerte `workingTime` und `queryText` vorliegen, wird der Anfragekontext wieder geschlossen und die Messwerte sind erfasst.

5.2.5 Die ersten Messergebnisse

Der implementierte Beobachter misst die Bearbeitungszeit einer Anfrage einfach mittels zweimaligem Abfragen der Systemzeit, die jeweils vor Beginn der Durchführung und nach Ablauf der Durchführung abgefragt wird. Aus diesem und aus mehreren anderen Gründen sind die ermittelten Werte nicht aussagekräftig. Es kam bei der Messung nur darauf an zu beweisen, dass die Messdatenerfassung funktioniert. Es

wurden zwei Messdurchläufe durchgeführt. Im ersten Durchlauf wurde das ACCache-System neu initialisiert. Die Ergebnisse dazu sind in Abbildung 5.7 abgebildet. Im zweiten Messdurchlauf wurde die Messung bei „warmgelaufenem“ Cache nochmals wiederholt. Die Ergebnisse sind in Abbildung 5.8 abgebildet.

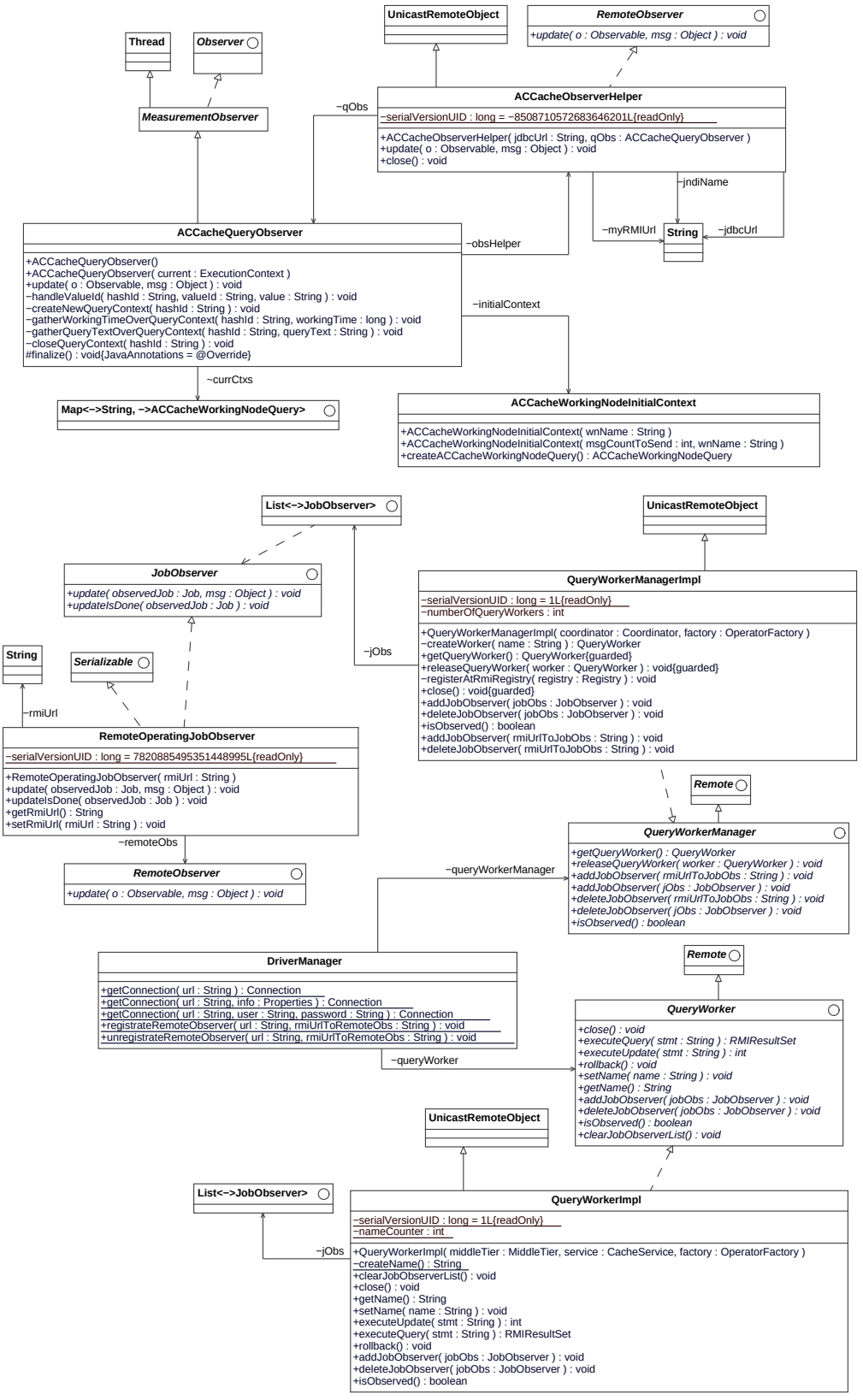

Abbildung 5.6: Die an der Beobachtung des ACCache-System beteiligten Klassen.

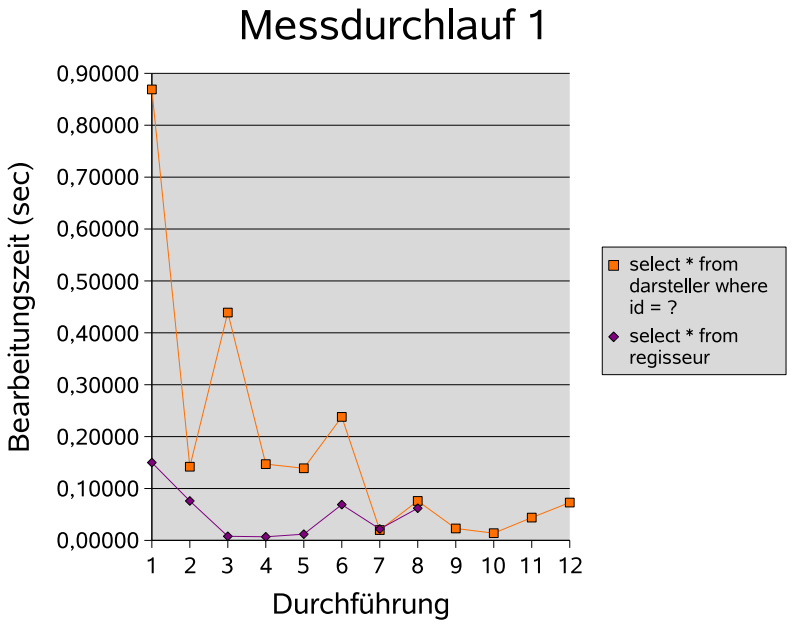

Abbildung 5.7: Die Messergebnisse des ersten Messdurchlaufes

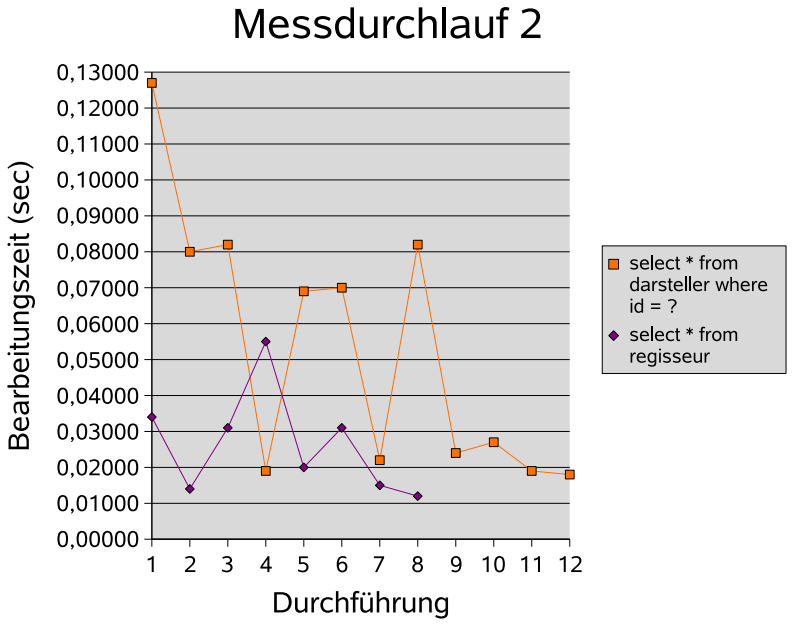

Abbildung 5.8: Die Messergebnisse des zweiten Messdurchlaufes

6. Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde ein Framework entwickelt, das es ermöglicht, Messdaten verteilter Systeme zu erfassen. Es erlaubt, die genaue Struktur eines verteilten Systems abzubilden, und hilft dabei, die einzelnen Teilanwendungen über zu definierende Konfigurationsschritte automatisch zu steuern.

Die Art und Weise, wie Messdaten erfasst, übermittelt und gespeichert werden, wurde festgeschrieben und läuft für den Entwickler komplett transparent ab. Hierbei ist vor allem die automatische Generierung von Ausführungskontexten eine große Hilfe. Sie verhindert durch das Generieren Implementierungsfehler und erleichtert es dem Entwickler, über angepasste Methoden, Beobachter zu implementieren. Das Framework implementiert eine Standardmethode, übermittelte Messwerte abzuspeichern, die ein sich selbst erweiterndes Datenbankschema benutzt. Es stellt bei einer großen Menge erfasster Messdaten sicher, dass die Werte anhand ihrer Ausführungkontexte, über die sie erfasst wurden, ordentlich strukturiert gespeichert werden. Somit konnte ein einfacher AOW-Ansatz vermieden werden, der die Auswertbarkeit und Weiterverarbeitung der Messdaten sehr behindert hätte.

Darüber hinaus wurde die Möglichkeit geschaffen, das Verhalten eines oder mehrerer Benutzer zu simulieren. Dabei erlaubt die Definition von auszuführenden Arbeitschritten die Auszeichnung beliebiger Arbeitslasten. In der Zukunft ist es daher möglich, die Anfragelasten eine TCP-C- oder TPC-W-Benchmarks zu implementieren, um das ACCache-System genau zu vermessen. Es ist bereits vorgesehen, das Framework mit einer Komponente zu erweitern, die es erlaubt, anhand von Regeln verschiedene Anfragelasten sowie Daten zu generieren, so dass die Entwicklung verschiedener Messkonfigurationen deutlich erleichtert und beschleunigt wird.

Da es zur Zeit notwendig ist, alle Konfigurationen über Konfigurationsdateien vorzunehmen, ist es ratsam, in der Zukunft eine graphische Oberfläche für die Messumgebung zu implementieren. Diese würde es erlauben, die Konfigurationen über Wizards und Dialoge zu steuern.

Die durch das Framework definierten Komponenten sind nicht auf bestimmte Architekturen festgelegt und sind somit beliebig erweiterbar.

Die Implementierungen zur Vermessung des ACCache-Systems haben gezeigt, dass die Vermessung mit Hilfe des Frameworks möglich ist. Da im Rahmen dieser Arbeit noch keine aussagekräftigen Messergebnisse erzeugt wurden, ist es notwendig, die Implementierung entsprechend zu erweitern.

Die Durchführung von Transaktionen, die auch Update-Operationen beeinhalten, wird aber erst dann möglich, wenn eine Update-Strategie für das ACCache-System implementiert wurde, was eine der wichtigsten Anforderungen an die zukünftige Forschung darstellt.

A. Konfigurationsdateien

A.1 Konfigurationsdatei der Messumgebung

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
   <comment>
    Diese Datei enthält alle Konfigurationsparameter,
    die für den MeasurementManager direkt von Bedeutung sind.

    Konfigurationswerte für das Schreiben und Lesen
    der Messwerte
    optional: Das Anlegen und damit auch Löschen der Datenbank
              automatisieren
    optional: Das Starten der Datenabank Instanz automatisieren
   </comment>

   <entry key="MData:DBHost">localhost</entry>
   <entry key="MData:DBName">mdata</entry>
   <entry key="MData:JDBCUrl">
     jdbc:db2://localhost:50001/mdata</entry>
   <entry key="MData:Driver">com.ibm.db2.jcc.DB2Driver</entry>
   <entry key="MData:Username">mm_usr</entry>
   <entry key="MData:Password">********</entry>
   <entry key="MData:DataWriterClass">
      mm.self.wn.datacolector.DefaultDataWriter</entry>
   <entry key="MData:doCreateSchema">false</entry>

   <entry key="MData:doCreateDB">false</entry>
   <entry key="MData:CreateDB:Username">db2inst1</entry>
   <entry key="MData:CreateDB:Password">********</entry>
   <entry key="MData:CreateDB:CreateScriptTplFile">
      config/manager/sql/CreateDBMeasurementManagerScript.tpl
```

```
   </entry>

   <entry key="MData:doStartDB">false</entry>
   <entry key="MData:StartDB:Username">db2inst1</entry>
   <entry key="MData:StartDB:Password">********</entry>
   <entry key="MData:StartDB:StartCommand">db2start</entry>
   <entry key="MData:StartDB:StopCommand">db2stop</entry>
</properties>
```

A.1.1 Verwendetes Skripttemplate zum Anlegen der Datenbank

Die Namen der Parameter aus der Konfigurationsdatei können direkt dazu verwendet werden die Werte im Template zu referenzieren.

```
-- CreateDB Datenbankscript-Template für den MeasurementManager

-- Anlegen der Datenbank
db2 create database <MData:DBName>

-- Dem Writer-User die Rechte an der Datenbank geben
db2 grant connect, createtab on database to <MData:Username>
```

A.2 Konfigurationsdatei für Arbeitsknotentypen

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE WorkingNodeConfig SYSTEM "WorkingNodeConfig.dtd">
<WorkingNodeConfig>
   <!-- wenn hier definiert, gilt dieser Wert für alle WorkingNodes
   <MeasurementMessagesCountToSend>0</MeasurementMessagesCountToSend>
   -->

   <WorkingNode id="JDBCClient" class="JDBCClient">
      <!--
      MeasurementMessagesCountToSend wird in der Projekt-Config
      für jede WorkingNode Instanz einzeln festgelegt.
      Wenn hier definiert, überschreibt dies den Wert für
      alle Instanzen von JDBCClient
      <MeasurementMessagesCountToSend>0</MeasurementMessagesCountToSend>
     -->

      <!-- Die Observer des JDBCClients -->
      <Observer id="JDBCClientObs1"
                class="JDBCClientTransactionObserver" />
      <Observer id="JDBCClientObs2"
                class="JDBCClientQueryObserver" />
```

```
      <Observer id="JDBCClientObs3"
                class="JDBCClientProcessObserver" />

      <!-- Die Kontexte des JDBCClients -->
      <InitialContext id="JDBCClientContext0"
                      generateClass="false"
                      class="InitialContext">
         <AllowableSubContexts>
            <ContextPermission idref="JDBCClientContext1"/>
         </AllowableSubContexts>
      </InitialContext>

      <ContextDef id="JDBCClientContext1"
                  name="Transaction"
                  generateClass="true">
         <MeasurementValue name="workingTime" javatype="float"
                           sqltype="decimal(10,2)"/>
         <MeasurementValue name="checkTime" javatype="java.util.Date"
                           sqltype="date"/>
         <PeriodicMeasurementValue name="cpuUsage"
                                   javatype="String" sqltype="clob"
                                   joinedStorage="true" separator=","/>
         <AllowableSubContexts>
            <ContextPermission idref="JDBCClientContext2"/>
         </AllowableSubContexts>
      </ContextDef>

      <ContextDef id="JDBCClientContext2" name="Query"
                  generateClass="true">
         <MeasurementValue name="workingTime" javatype="float"
                           sqltype="decimal(10.2)"/>
         <AllowableSubContexts></AllowableSubContexts>
      </ContextDef>
   </WorkingNode>

<WorkingNode id="ACCacheWorkingNode" class="ACCacheWorkingNode">
    <!-- Daten erst am Ende der Messung übertragen -->
    <!--
     MeasurementMessagesCountToSend wird in der Projekt-Config
     für jede WorkingNode Instanz einzeln festgelegt.
     Wenn hier definiert, überschreibt dies den Wert für
     alle Instanzen von ACCacheWorkingNode:
     <MeasurementMessagesCountToSend>0</MeasurementMessagesCountToSend>
    -->

    <!-- Die Observer des ACCacheWorkingNode! -->
    <Observer id="obs4"
     class="ACCacheTransactionObserver" />
    <Observer id="obs5"
```

```
 class="ACCacheQueryObserver" />

<!-- Die Kontexte des ACCacheWorkingNode-->

<InitialContext id="ACCacheContext0"
                generateClass="true"
                name="InitialContext">
   <AllowableSubContexts>
      <ContextPermission idref="ACCacheContext1"/>
      <ContextPermission
          class="mm.accache.wn.context.ACCacheWorkingNodeQuery"/>
   </AllowableSubContexts>
</InitialContext>

<ContextDef id="ACCacheContext1" name="Transaction"
           generateClass="true">
   <MeasurementValue name="workingTime" javatype="float"
                     sqltype="decimal(10,2)"/>
   <MeasurementValue name="aborted" javatype="boolean"
                     sqltype="char(1)"/>
   <AllowableSubContexts>
      <ContextPermission idref="ACCacheContext2"/>
   </AllowableSubContexts>
</ContextDef>

<ContextDef id="ACCacheContext2" name="Query"
           generateClass="true">
   <MeasurementValue name="workingTime" javatype="float"
                     sqltype="decimal(10,2)"/>
   <AllowableSubContexts>
      <ContextPermission idref="ACCacheContext3"/>
      <ContextPermission idref="ACCacheContext4"/>
      </AllowableSubContexts>
   </ContextDef>

   <ContextDef id="ACCacheContext3" name="BackendQuery"
               generateClass="true">
      <MeasurementValue name="workingTime" javatype="float"
                        sqltype="decimal(10,2)"/>
      <AllowableSubContexts></AllowableSubContexts>
   </ContextDef>

   <ContextDef id="ACCacheContext4" name="FillAction"
               generateClass="true">
      <MeasurementValue name="value" javatype="String"
                        sqltype="varchar(255)"/>
      <MeasurementValue name="column" javatype="String"
                        sqltype="varchar(255)"/>
      <AllowableSubContexts></AllowableSubContexts>
```

```
      </ContextDef>

   </WorkingNode>

   <WorkingNode id="DB2WorkingNode" class="DB2WorkingNode">
      <!-- keine Messung auf diesem Koten -->
      <messageCountToSend>-1</messageCountToSend>
   </WorkingNode>

</WorkingNodeConfig>
```

A.3 Konfigurationsdatei für Messkonfigurationen

Konfiguration von 2 simulierten Client-Arbeitsknoten, einem ACCache-System und einer DB2-Datenbank von IBM.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MM-ProjectDef SYSTEM "ProjektConfig.dtd">
<MM-ProjectDef>
<project
 desc="Dies ist eine Beschreibung fuer: Projekt1"
 name="Projekt1">

<config
 desc="Dies ist eine Beschreibung fuer: Konfiguration1"
 name="Konfiguration1">

   <workingnode
    class="mm.accache.wn.nodes.JDBCClient"
    desc="Dies ist eine Beschreibung fuer: ClientWN1"
    name="ClientWN1">
      <servicecontext
       class="mm.self.sc.URIServiceContext"
       className="ClientWN1"
       host="localhost"
       port="1099"/>

      <workunit
       class="mm.accache.wu.SelectAllDirectors"
       desc="Selektiert alle Direktoren"
       name="SelAllDirectors">
         <schedule maxExecutions="noLimit" overallPercent="30"/>
      </workunit>
      <workunit
       class="mm.accache.wu.SelectAnActor"
       desc="Selektiert einen Darsteller"
       name="SelDarst">
         <schedule maxExecutions="noLimit" overallPercent="70"/>
```

```
    </workunit>

    <mmServiceParameter
     rmihost="localhost"
     jndiport="1099"
     jndiname="ClientWN1"
     username="mms_usr"
     password="mms_service"
    >
       <classpathEntry>$MS_HOME/bin/</classpathEntry>
       <classpathEntry>
          $MS_HOME/lib/log4j-1.2.13.jar
       </classpathEntry>
       <classpathEntry>
          $MS_HOME/lib/commons-cli-1.0.jar
       </classpathEntry>
    </mmServiceParameter>

    <messageCountToSend>0</messageCountToSend>
    <observer class="JDBCClientTransactionObserver" active="true"/>
    <observer class="JDBCClientQueryObserver" active="true"/>
    <observer class="JDBCClientProcessObserver" active="true"/>
</workingnode>

<workingnode
 class="mm.accache.wn.nodes.JDBCClient"
 desc="Dies ist eine Beschreibung fuer: ClientWN2"
 name="ClientWN2">
    <servicecontext
     class="mm.self.sc.URIServiceContext"
     className="ClientWN2"
     host="localhost"
     port="1099"/>
    <workunit
     class="mm.accache.wu.SelectAllDirectors"
     desc="Selektiert alle Direktoren"
     name="SelAllDirectors">
       <schedule maxExecutions="40" overallPercent="100"/>
    </workunit>

    <mmServiceParameter
     rmihost="localhost"
     jndiport="1099"
     jndiname="ClientWN2"
     username="mms_usr"
     password="mms_service"
    >
       <classpathEntry>$MS_HOME/bin/</classpathEntry>
       <classpathEntry>$MS_HOME/lib/log4j-1.2.13.jar</classpathEntry>
```

```
      <classpathEntry>$MS_HOME/lib/commons-cli-1.0.jar</classpathEntry>
   </mmServiceParameter>

   <messageCountToSend>0</messageCountToSend>
   <observer class="JDBCClientTransactionObserver" active="true"/>
   <observer class="JDBCClientQueryObserver" active="true"/>
   <observer class="JDBCClientProcessObserver" active="true"/>
</workingnode>

<workingnode
 class="mm.accache.wn.nodes.ACCacheWorkingNode"
 desc="Dies ist eine Beschreibung fuer: ACCacheWN1"
 name="ACCacheWN1">
   <servicecontext
    class="mm.accache.sc.JDBCServiceContext"
    databaseServiceName=""
    driver="ACCache"
    host="localhost"
    username="accache"
    password="********"
    port="1099"
    url="rmi://localhost:1099/ACCache"/>
   <mmServiceParameter
    rmihost="localhost"
    jndiport="1099"
    jndiname="ACCacheWN1"
    username="mms_usr"
    password="mms_service"
   >
      <classpathEntry>$MS_HOME/bin/</classpathEntry>
      <classpathEntry>$MS_HOME/lib/log4j-1.2.13.jar</classpathEntry>
      <classpathEntry>$MS_HOME/lib/commons-cli-1.0.jar</classpathEntry>
   </mmServiceParameter>

   <messageCountToSend>0</messageCountToSend>
   <observer class="ACCacheTransactionObserver" active="true"/>
   <observer class="ACCacheQueryObserver" active="true"/>
</workingnode>

<workingnode
 class="mm.accache.wn.nodes.DB2WorkingNode"
 desc="Dies ist eine Beschreibung fuer: DB2WN1"
 name="DB2WN1">
   <servicecontext
    class="mm.accache.sc.JDBCServiceContext"
    databaseServiceName=""
    driver="com.ibm.db2.jcc.DB2Driver"
    host="localhost" password="********" port="50001"
    url="jdbc:db2://192.168.1.128:50001/backend"
```

```
        username="backend"/>
          <messageCountToSend>-1</messageCountToSend>
    </workingnode>

    <workingnodeconnection wnRefFrom="ACCacheWN1" wnRefTo="DB2WN1"/>
    <workingnodeconnection wnRefFrom="ClientWN1" wnRefTo="ACCacheWN1"/>
    <workingnodeconnection wnRefFrom="ClientWN2" wnRefTo="ACCacheWN1"/>
</config>
</project>
</MM-ProjectDef>
```

B. DDL-Definitionen des DefaultDataWriterSchemas

B.1 Die statischen Elemente des Schemas

```
-- Defaut DataWriter SQL Schema Beschreibung

create table MeasurementRun (
   mr_id        integer  not null primary key,
   mr_startdate date     not null,
   mr_desc      clob
);

create sequence mr;

create table WorkingNode (
   wn_id     integer      not null,
   wn_mr_id  integer      not null
      references MeasurementRun(mr_id)
      on delete cascade,
   wn_name  varchar(100)  not null,
   wn_host  varchar (50)  not null,
   wn_port  char(6)       not null,
   wn_desc  clob,
   unique (wn_name, wn_mr_id),
   primary key (wn_id, wn_mr_id)
);

create sequence wn;
```

```
create table WorkingNodeConnection (
   wnc_mr_id      integer not null,
   wnc_wn_from_id integer not null,
   wnc_wn_to_id   integer not null,
   primary key (wnc_mr_id, wnc_wn_from_id, wnc_wn_to_id),
   foreign key (wnc_mr_id, wnc_wn_from_id)
     references WorkingNode(wn_id, wn_mr_id)
     on delete cascade,
   foreign key (wnc_mr_id, wnc_wn_to_id)
   references WorkingNode(wn_id, wn_mr_id)
      on delete cascade
);

create table ExecutionContextTyp (
   ct_id    integer      not null primary key,
   ct_wn_id integer      not null,
   ct_mr_id integer      not null
      references MeasurementRun(mr_id),
   ct_name  varchar(100) not null,
   ct_desc clob,
   foreign key (ct_wn_id, ct_mr_id)
      references WorkingNode(wn_id, wn_mr_id)
);

create sequence ct;
```

B.2 Die dynamischen Elemente des Schemas

Da die nachfolgenden Tabellen und Sequenzen erst dynamisch während der Messung, je nach übermitteltem Ausführungskontext, angelegt werden, sind sie hier in einer Template-Schreibweise aufgeführt. Die mit spitzen Klammern gekennzeichneten Bezeichner werden während des Messablaufes durch die dann gültigen Werte ersetzt.

```
-- JS steht für joinedStorage

create table MeasurementValues_<ct_id> (
   mv_id   integer   not null  primary key,
   mv_<MeasurementValueName_1> <MeasurementValueType_1>,
   ...                         ...
   mv_<MeasurementValueName_N> <MeasurementValueType_N>,
   mv_<PeriodicValueNameJS_1>  <PeriodicValueTypeJS_1>,
   ...                         ...
   mv_<PeriodicValueNameJS_N>  <PeriodicValueTypeJS_N>
);

create sequence mv_<ct_id>;
```

```
create table PeriodicValues_<ct_id>_<PeriodicValueName_1> (
   pv_mv_id  integer not null
      references MeasurementValues_<ct_id>(mv_id),
   pv_index  integer not null,
   pv_value  <PeriodicValueType_1>,
   primary key (pv_mv_id, pv_index)
);
...
create table PeriodicValues_<ct_id>_<PeriodicValueName_N> (
   pv_id     integer not null,
      references MeasurementValues_<ct_id>(mv_id),
   pv_index  integer not null,
   pv_value  <PeriodicValueType_N>,
   primary key (pv_mv_id, pv_index)
);
```

C. Konfigurationen für den exemplarischen Messlauf

C.1 Konfigurationsdatei für Arbeitsknotentypen

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE WorkingNodeConfig SYSTEM "WorkingNodeConfig.dtd">
<WorkingNodeConfig>
   <WorkingNode id="JDBCClient" class="JDBCClient">
      <!-- keine Messung auf diesem Koten -->
      <messageCountToSend>-1</messageCountToSend>
   </WorkingNode>

   <WorkingNode id="ACCacheWorkingNode" class="ACCacheWorkingNode">
      <Observer id="obs1"
       class="ACCacheQueryObserver" />

      <InitialContext id="ACCacheContext0"
                      generateClass="true"
                      name="ACCacheInitialContext">
         <AllowableSubContexts>
            <ContextPermission idref="ACCacheContext1"/>
         </AllowableSubContexts>
      </InitialContext>

      <ContextDef id="ACCacheContext1" name="Query"
                  generateClass="true">
         <MeasurementValue name="workingTime" javatype="float"
                           sqltype="decimal(10,2)"/>
         <MeasurementValue name="queryText" javatype="String"
                           sqltype="varchar(200)"/>
         <AllowableSubContexts>
         </AllowableSubContexts>
```

```
      </ContextDef>
   </WorkingNode>

   <WorkingNode id="DB2WorkingNode" class="DB2WorkingNode">
      <!-- keine Messung auf diesem Koten -->
      <messageCountToSend>-1</messageCountToSend>
   </WorkingNode>
</WorkingNodeConfig>
```

C.2 Konfigurationsdatei für Messkonfigurationen

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MM-ProjectDef SYSTEM "ProjektConfig.dtd">
<MM-ProjectDef>
<project
 desc="Projekt für den exemplarischen Messlauf"
 name="exempML">

<config
 desc="Messkonfiguration des exemplarischen Messlaufs"
 name="exempKonf">

   <workingnode
    class="mm.accache.wn.nodes.JDBCClient"
    desc="Der simulierte Client WorkingNode"
    name="JDBCClient">
      <servicecontext
       class="mm.self.sc.URIServiceContext"
       className="JDBCClient"
       host="localhost"
       port="1099"/>

      <workunit
       class="mm.accache.wu.SelectAllDirectors"
       desc="Selektiert alle Regisseure"
       name="SelAllDirectors">
         <schedule maxExecutions="8" overallPercent="40"/>
      </workunit>
      <workunit
       class="mm.accache.wu.SelectAnActor"
       desc="Selektiert einen Darsteller"
       name="SelDarst">
         <schedule maxExecutions="12" overallPercent="60"/>
      </workunit>

      <mmServiceParameter
       rmihost="localhost"
       jndiport="1099"
```

```
     jndiname="JDBCClientService"
     username="mms_usr"
     password="mms_service"
    >
       <classpathEntry>$MS_HOME/bin/</classpathEntry>
       <classpathEntry>
          $MS_HOME/lib/log4j-1.2.13.jar
       </classpathEntry>
       <classpathEntry>
          $MS_HOME/lib/commons-cli-1.0.jar
       </classpathEntry>
    </mmServiceParameter>

    <messageCountToSend>-1</messageCountToSend>
</workingnode>

<workingnode
 class="mm.accache.wn.nodes.ACCacheWorkingNode"
 desc="Der Arbeitsknoten repräsentiert ein ACCache-System"
 name="ACCache">
    <servicecontext
     class="mm.accache.sc.JDBCServiceContext"
     databaseServiceName=""
     driver="cacheManager.sql.DriverManager"
     host="localhost"
     username=""
     password=""
     port="1099"
     url="localhost:1099"/>

    <mmServiceParameter
     rmihost="localhost"
     jndiport="1099"
     jndiname="ACCacheWNService"
     username="mms_usr"
     password="mms_service"
    >
       <classpathEntry>$MS_HOME/bin/</classpathEntry>
       <classpathEntry>$MS_HOME/lib/log4j-1.2.13.jar</classpathEntry>
       <classpathEntry>$MS_HOME/lib/commons-cli-1.0.jar</classpathEntry>
    </mmServiceParameter>

    <messageCountToSend>1</messageCountToSend>
    <observer class="ACCacheQueryObserver" active="true"/>
</workingnode>

<workingnode
 class="mm.accache.wn.nodes.DB2WorkingNode"
 desc="Die Repräsentierung des DB2-Datenbanksystems"
```

```
    name="DB2">
      <servicecontext
       class="mm.accache.sc.JDBCServiceContext"
       databaseServiceName=""
       driver="com.ibm.db2.jcc.DB2Driver"
       host="localhost" password="********" port="50001"
       url="jdbc:db2://localhost:50001/backend"
       username="db2inst1"/>
      <messageCountToSend>-1</messageCountToSend>
   </workingnode>

   <workingnodeconnection wnRefFrom="ACCache" wnRefTo="DB2"/>
   <workingnodeconnection wnRefFrom="JDBCClient" wnRefTo="ACCache"/>
</config>
</project>
</MM-ProjectDef>
```

Literatur

[BD96] Beuter, T. ; Dadam, P.: Prinzipien der Replikationskontrollen in verteilten Datenbanksystemen. In: *Informatik - Forschung und Entwicklung*, 1996

[Büh05] Bühmann, Andreas: Einen Schritt zurück zum negativen Datenbank-Caching. In: *BTW 2005: Datenbanksysteme für Business, Technologie und Web, Gesellschaft für Informatik, Karlsruhe, Lecture Notes in Informatics*, 2005, S. 107–124

[Büh06] Bühmann, Andreas: Einen Schritt zurück zum negativen Datenbank-Caching. In: *Informatik - Forschung und Entwicklung 20:4*, 2006, S. 184–195

[Bur98] Buretta, Marie: *Data Replication*. John Wiley & Sons Inc, Dezember 1998. – ISBN 0–47115–754–6

[Dad96] Dadam, Peter: *Verteilte Datenbanken und Client/Server Systeme*. Springer Verlag, 1996

[HB07] Härder, Theo ; Bühmann, Andreas: Value Complete, Column Complete, Predicate Complete - Magic Words Driving the Design of Cache Groups. In: *VLDB Journal*, 2007. – (Zur Veröffentlichung bereits angenommen).

[Mer05] Merker, Christian. *Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache*. http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Mer05.DA.html. Oktober 2005

[Sch06] Scholl, Wolfgang. *Cache-Group-Optimierung zur Effizienzsteigerung von Datenbank-Caches*. http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Sch06.PA.pdf. April 2006

[TS03] Tanenbaum, Andrew S. ; Steen, Maarten v.: *Verteilte Systeme: Grundlagen und Paradigmen*. Pearson Studium, 2003. – ISBN 3–8273–7057–4