Universität Paderborn
Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
D-33098 Paderborn

Durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen mittels UML

Diplomarbeit
für den integrierten Studiengang Wirtschaftsinformatik
im Rahmen des Hauptstudiums II

Matthias Tichy
Buchholzer Weg 4
40472 Düsseldorf
Matr.-Nr.: 3595185

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Gregor Engels

betreut von
Dr. Holger Giese

Paderborn, im Juli 2002

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer, als der angegebenen, Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

1 Einführung

Große Softwaresysteme bilden das Rückgrad jedes Unternehmens und sind entscheidend für dessen Erfolg. Sie bestehen aus mehreren Millionen Zeilen Quelltext und werden dezentral von großen Teams entwickelt. Heutzutage bestehen diese Softwaresysteme nicht mehr aus großen monolithischen Blöcken, sondern aus vielen einzelnen, räumlich und logisch verteilten Systemen. Es ist notwendig, dass die einzelnen Teilsysteme eines verteilten Systems einfach integriert werden können. Für die erfolgreiche Nutzung dieser Softwaresysteme sind einige Eigenschaften vonnöten. Das System muss einfach und kostengünstig um neue Fähigkeiten erweitert und an andere Umgebungsbedingungen angepasst werden können. Zudem muss die Wahrscheinlichkeit von Ausfällen minimal sein.

Diese Probleme – Wartbarkeit und Fehlertoleranz – werden in dieser Arbeit angesprochen und ein Ansatz zur Lösung vorgeschlagen.

1.1 Wartbarkeit

In der langen Einsatzzeit der Softwaresysteme müssen diese an viele Neuerungen angepasst werden. Diese Änderungen sind zum Teil durch die Umwelt vorgegeben, wie zum Beispiel die Umstellung auf den Euro oder die jährlichen Änderungen in der Steuergesetzgebung. Andere Änderungen werden vom Unternehmen durchgeführt, um zum Beispiel Produkte über das Internet anzubieten oder durch Verknüpfung der Bestellsysteme mit den Informationstechnologiesystemen der Zulieferer Kosten zu sparen und schneller auf Marktveränderungen zu reagieren. Die Softwaresysteme unterliegen deshalb einer dauernden Veränderung bzw. Evolution.

Die Möglichkeit, einfach, schnell und kostengünstig das Softwaresystem zu ändern, das System also zu warten, hat mehrere Aspekte. Ein Aspekt ist die Lesbarkeit des geschriebenen Quelltextes bzw. dessen Kommentierung. Des Weiteren ist die Spezifikation bzw. Dokumentation und deren Übereinstimmung mit der Implementierung wichtig für die Wartbarkeit eines Softwaresystems.

Ein wichtiger Aspekt der Wartbarkeit resultiert aus Abhängigkeiten zwischen den einzelnen Teilen des Softwaresystems. Wenn diese Abhängigkeiten nicht erkennbar oder in der Dokumentation schlecht bzw. gar nicht spezifiziert sind, stellen den Entwickler schon einfachste Änderungen vor die Frage, was im restlichen

System passiert, wenn ein bestimmter Teil der Software leicht geändert wird.

Um dieses Problem der Abhängigkeiten zu lösen, wurden verschiedene Konzepte entwickelt. In Modulen werden zusammengehörige Teile eines Systems gruppiert, so dass Abhängigkeiten vor allem innerhalb eines Moduls auftreten und weniger zwischen den Modulen. In der objektorientierten Welt wird das Konzept der Kapselung genutzt, um von der eigentlichen Implementierung zu abstrahieren und eine Schnittstelle zur Benutzung eines Objektes anzubieten. Jedes Objekt kann hier nur über diese Schnittstelle, bestehend aus bestimmten Methoden, angesprochen werden. Der Zugriff auf interne Attribute und Methoden ist verboten. Da in einem größeren Softwaresystem allerdings eine Vielzahl von Klassen mit ihren Abhängigkeiten existieren und die Schnittstellen der Objekte nur syntaktische Überprüfung zulassen, sind weitergehende Konzepte nötig.

Komponenten bieten für die Schnittstellen einen Kontraktbegriff, der es erlaubt, genauer zu überprüfen, ob bestimmte Teilsysteme zusammenpassen bzw. welche Änderungen an einer Komponente durchgeführt werden dürfen, ohne die Nutzungsmöglichkeiten dieser Komponente unerwartet zu ändern. Für Komponenten und ihre Schnittstellen können des Weiteren die Abhängigkeiten zwischen den Schnittstellen spezifiziert werden. Diese Abhängigkeiten können zur Laufzeit überprüft werden, um zu bestimmen, ob eine bestimmte Kommunikation zwischen Komponenten erlaubt ist. In Abschnitt 2.1 werden die Grundbegriffe von Komponenten näher erläutert.

Das Problem der Wartbarkeit betrifft auch den Betrieb der Softwaresysteme. Die verwendeten Hard- und Softwarearchitekturen müssen sehr flexibel sein, um geänderte Erwartungen an das Softwaresystem zu unterstützen und zu erfüllen.

Aufgrund der oben genannten Gründe ist es wichtig, den kompletten Lebenszyklus eines immer wieder zu ändernden Softwaresystems, also Entwurf, Implementierung und Betrieb, durchgängig zu unterstützen.

1.2 Fehlertoleranz

Wie oben beschrieben ist die Verfügbarkeit der Systeme sehr wichtig. In zentralisierten, nicht verteilten Systemarchitekturen ist daher die Verfügbarkeit der zentralen Systeme äußerst wichtig. Hier muss viel Aufwand betrieben werden, damit diese Systeme jederzeit verfügbar sind. Da auch der Ausfall einzelner Teile der Hardware eines zentralen Systems zum kompletten Versagen führt, sind häufig alle Teile redundant in Hardware ausgeführt. Dem Datenverlust beim Ausfall einer Festplatte wird zum Beispiel durch automatische Spiegelung der Daten auf einer Ersatzfestplatte vorgebeugt. Diese redundanten Teile steigern die Kosten des Systems, erhöhen aber nicht die Geschwindigkeit oder den Durchsatz. Für extreme Verfügbarkeit werden Backup-Systeme eingesetzt, die beim Ausfall des primären Systems nahtlos die Aufgaben übernehmen können, allerdings im normalen Betrieb nicht genutzt werden.

Im Gegensatz dazu besteht bei verteilten Systemen die Möglichkeit, auch bei einem Teilausfall das Funktionieren des Gesamtsystems sicher zu stellen. Dies muss allerdings bereits beim Entwurf des Systems berücksichtigt werden. Bei Ausfällen von einzelnen Rechenknoten besteht zum Beispiel die Möglichkeit, die auf diesem Rechenknoten ausgeführten Komponenten des Softwaresystems auf anderen Rechenknoten neu zu starten. Algorithmen für verteilte Systeme ermöglichen, den Ausfall von Teilen des Systems zu maskieren und dadurch Datenverlust zu vermeiden sowie die ausgefallenen Teile parallel im Betrieb zu ersetzen. Die Funktionalität des verteilten Systems ist durch den Ausfall von Teilen nicht beeinflusst, allerdings ist die Geschwindigkeit und der Durchsatz im Fehlerfall niedriger.

Die Fehlerarten in einem verteilten System beschränken sich nicht nur auf den oben beschriebenen Ausfall von Rechenknoten, sondern auch das genutzte Kommunikationsmedium kann Ursprung von Fehlern sein. Diese zusätzlich auftretenden Fehler werden in Abschnitt 2.2.2 dargestellt. In Kapitel 2.3 werden Konsistenzmodelle angesprochen und in den Abschnitten 5.1.2 und 5.5 zwei Algorithmen vorgestellt, welche die Konsistenz von Daten in einem verteilten System auch bei diversen Fehlern sicher stellen.

1.3 Ziele der Arbeit

Ziel dieser Arbeit ist es, den kompletten Lebenslauf von Komponenten in offenen Softwarearchitekturen zu unterstützen. Offene Softwarearchitekturen charakterisieren sich durch die Möglichkeit der Integration von beim Entwurf unbekannten (Teil-)Systemen und einem dauernden Wandel des gesamten Systems. Um die während der Entwurfszeit unbekannten Systeme zur Laufzeit in die Architektur einbinden zu können, müssen explizit festgelegte Protokolle und Standards unterstützt und genutzt werden. Der Lebenslauf der Komponenten beinhaltet, wie im Titel dieser Arbeit bereits ausgesagt, den Entwurf und die Implementierung von Komponenten sowie deren Betrieb in einem verteilten System.

Aus den Erläuterungen der ersten beiden Abschnitte lassen sich die folgenden Anforderungen an eine Lösung extrahieren. Allgemein muss eine graphische Spezifikationssprache während des gesamten Lebenszyklus der Komponenten unterstützt werden. Vorteil einer graphischen Spezifikationssprache ist es, alle Aspekte eines Systems in Hinsicht auf Struktur, Verhalten, Kommunikation, etc. aus einem Diagramm einfach ersehen zu können. Speziell bei Entwurf und Implementierung sollte es möglich sein, aus der graphischen Spezifikation direkt Quelltext der Komponenten zu generieren, um die Differenz zwischen Spezifikation und Implementierung zu minimieren. Des Weiteren ist es erforderlich, den Entwickler einer Komponente von Standardaufgaben, wie zum Beispiel die Integration in eine bestimmte Softwarearchitektur und die Integration mit anderen Komponenten, zu entlasten. Das zu entwickelnde System soll prinzipiell verteilt

arbeiten, um die im obigen Abschnitt dargestellte Möglichkeit der Fehlertoleranz zu erreichen. Diese Verteilung darf vor dem Entwickler nicht komplett verborgen werden, um eine explizite Reaktion auf Probleme verteilter Systeme zu ermöglichen. In Hinsicht auf Fehlertoleranz ist besonders darauf zu achten, dass keine zentralen Teile existieren, deren Verfügbarkeit Auswirkungen auf das gesamte System hat (*Single Point of Failure*).

1.4 Lösungsansatz

Im Folgenden wird der für diese Arbeit gewählte Lösungsansatz grob skizziert. In allen Bereichen (Entwurf, Implementierung und Betrieb von Komponenten) wird die Unified Modelling Language (UML) als graphische Notation genutzt. Die für diese Arbeit genutzten Diagrammarten und deren Erweiterungen werden in Kapitel 4 angesprochen. Durch die Nutzung von Komponenten und deren Spezifikation durch graphische Diagramme wird das in Abschnitt 1.1 vorgestellte Problem der Wartbarkeit von Softwaresystemen adressiert. Die Komponenten werden danach durch eine Laufzeitumgebung in einem verteilten System betrieben. Diese Laufzeitumgebung bietet die in Abschnitt 1.2 angesprochene Fehlertoleranz.

Abbildung 1.1 zeigt den Lebenszyklus der Komponenten. Für die einzelne Komponente wird in den Bereichen Entwurf und Implementierung die Unified Modelling Language genutzt, um mit den verschiedenen Struktur- und Verhaltensdiagrammen Komponenten zu spezifizieren und zu implementieren. Hierbei wird das UML Case-Tool Fujaba [FNT98] genutzt, um die Diagramme zu erstellen und danach aus diesen Diagrammen Java-Quelltext zu generieren. Dieser Quelltext wird danach kompiliert und die resultierenden binären Klassen zu mehreren Java-Archiven zusammengefasst sowie eine XML-Datei generiert, welche die Komponente mit ihren Schnittstellen beschreibt.

Mittels der aus der XML-Datei extrahierten Informationen kann das Gesamtsystem, bestehend aus den einzelnen Komponenten, geplant werden. Die Komposition des Systems geschieht mittels Komponentendiagrammen. Hierbei werden die benötigten und genutzten Schnittstellen der einzelnen Komponenten miteinander verknüpft und die Komponenten zu Kompositionskomponenten zusammengesetzt. Für den Betrieb des Systems werden dann die Komponenten in einem Verteilungsdiagramm auf Rechenknoten verteilt, die durch Bedingungen genauer spezifiziert werden. Aus diesen beiden Diagrammarten werden Verteilungsbeschreibungen generiert, die von der Laufzeitumgebung gelesen werden.

Nachdem der Betrieb des Systems geplant worden ist, wird die Laufzeitumgebung genutzt, um das System zu betreiben. Diese Laufzeitumgebung garantiert, dass jede Komponente des Systems auf passenden Rechenknoten, sofern vorhanden, ausgeführt wird. Die Laufzeitumgebung basiert auf einer verteilten Datenhaltung, welche die oben angesprochenen Verteilungsbeschreibungen speichert. Die Daten werden verteilt gespeichert, um das System fehlertolerant gegenüber

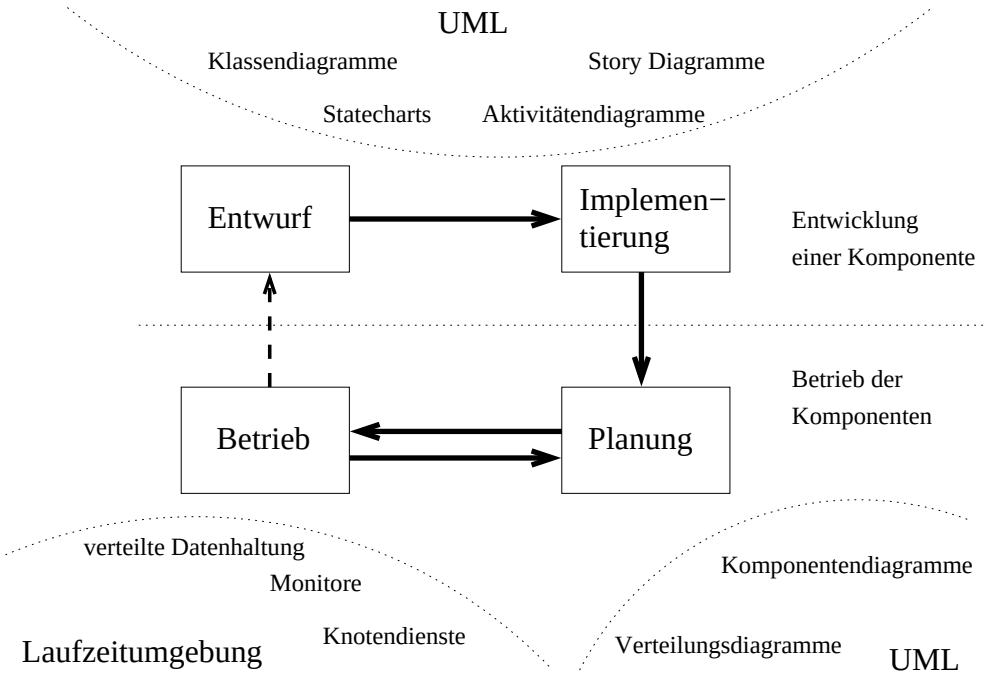

Abbildung 1.1: Gesamtübersicht

Ausfällen zu machen.

Damit die Laufzeitumgebung die Möglichkeit hat, die einzelnen Komponenten auf einem bestimmten Rechenknoten zu starten, muss auf jedem dieser Rechenknoten ein Knotendienst ausgeführt werden. Dieser Knotendienst startet diese Komponenten und gewährleistet die korrekte Verknüpfung der Komponentenschnittstellen. Zusätzlich stellt der Knotendienst Auslastungsdaten zur Verfügung, um die Auslastung des Systems darstellen und Daten für die Einleitung geeigneter Gegenmaßnahmen anbieten zu können.

Ein Monitor stellt sicher, dass eine zu verteilende Komponente auf jeden Fall auf einem Rechenknoten des Systems ausgeführt wird. Damit das System auch bei einem Ausfall eines Monitors zuverlässig funktioniert, wird das Konzept von Mietdauern benutzt. Hierbei ist jeder Monitor nur für eine gewisse Zeitdauer für die Verteilung einer Komponente verantwortlich. Der Monitor muss diese Mietdauer während seiner Laufzeit periodisch verlängern. Wird die Mietdauer nicht verlängert und läuft aus, da der Monitor zum Beispiel ausgefallen ist, kann ein anderer Monitor die Aufgaben des ausgefallenen Monitors übernehmen. Diese Mietdauern werden auch verteilt gespeichert, allerdings mit einer geringeren Konsistenz als die oben angesprochenen Verteilungsbeschreibungen, um eine höhere Verfügbarkeit zu erreichen.

Das letzte Element der Laufzeitumgebung ist eine Administrationskonsole, welche die aktuelle Situation im System anzeigt und die Möglichkeit bietet, neue Verteilungsbeschreibungen in das System einzupflegen.

1.5 Beispielszenario

Die in dieser Arbeit betrachteten Fragestellungen werden anhand eines durchgängigen Beispiels illustriert. Als Beispiel wird das Distributed Software Development (DSD) Projekt genutzt. Im DSD Projekt [GGT01, GNZ01] wurde ein komponentenbasiertes Versionierungs- und Konfigurationssystem entwickelt. Aufbauend auf dem Concurrent Versions System (CVS) [Ced93] existieren einzelne Softwarekomponenten für die verschiedenen Arbeitsaufgaben im Rahmen des Versions- und Konfigurationsmanagements. Dies sind zum Beispiel eine Komponente für die Übernahme der Änderungen anderer Entwickler in die eigenen Daten (*update*), eine Komponente für das Einpflegen der eigenen Änderungen in das Gesamtsystem (*checkin*) sowie Komponenten für die Anzeige der Versionshistorie einer Datei (*log*) und der Differenz zwischen verschiedenen Versionen einer Datei (*diff*).

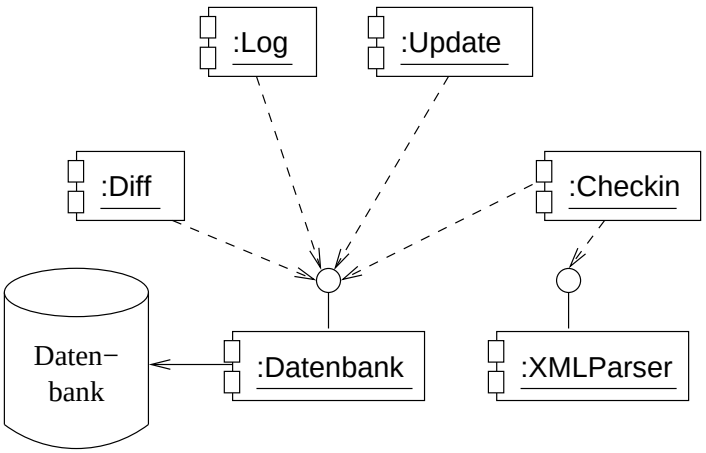

Abbildung 1.2: Beispiel DSD

Unterstützend für den Entwicklungsprozess werden die einzelnen Änderungen der Entwickler in einer Datenbank protokolliert und die Zugehörigkeit der Entwickler zu bestimmten Softwareprojekten in einer Datenbank hinterlegt. Für diese beiden Aufgaben existiert eine Datenbankkomponente, die Zugriff auf den Datenbankserver bereitstellt. Um bestimmte in XML kodierte Informationen zu verarbeiten, existiert des Weiteren eine Komponente für das Parsen von XML-Dokumenten. Abbildung 1.2 zeigt die vorgestellten Komponenten[1].

Es existieren noch weitere Komponenten, diese werden allerdings nicht für die Beispiele in dieser Arbeit benötigt und aus diesem Grund hier nicht weiter dargestellt.

[1] Der dafür verwendete Diagrammtyp sowie dessen Syntax und Semantik wird in Abschnitt 4.4 genau erläutert.

1.6 Gliederung der Arbeit

Diese Arbeit ist in sechs verschiedene Kapitel gegliedert. Das aktuelle Kapitel bietet eine Einführung in die Thematik und eine Erläuterung der Ziele dieser Arbeit. Kapitel 2 erläutert die für diese Arbeit relevanten Grundlagen in Hinsicht auf komponentenbasierte Systeme (Abschnitt 2.1), verteilte Systeme (Abschnitt 2.2) sowie Konsistenz von Daten in verteilten Systemen (Abschnitt 2.3) und verschiedene Konsistenzmodelle (Abschnitt 2.3.1).

Kapitel 3 beschreibt die relevanten Technologien für verteilte Systeme und einige verwandte Ansätze im Kontext dieser Arbeit, insbesondere die Jini-Technologie in Abschnitt 3.3. In Kapitel 4 werden die Ansätze für den Entwurf und die Implementierung einer Komponente sowie für die Planung des Betriebs dieser Komponente vorgestellt. Dies beinhaltet die Erläuterung der für diese Arbeit relevanten UML-Diagrammtypen und deren Erweiterungen. Kapitel 5 stellt die für den Betrieb der Komponenten genutzte Laufzeitumgebung vor und beinhaltet eine Implementierung für zwei Konsistenzmodelle (Abschnitt 5.1.2 und 5.5). Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 6.

2 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die für das Verständnis dieser Arbeit notwendig sind. Im ersten Abschnitt werden komponentenbasierte Systeme angesprochen. Der zweite Abschnitt erläutert verteilte Systeme, deren Herausforderungen und Fehlertypen. Im letzten Abschnitt werden Konsistenzmodelle im Bereich verteilter Systeme angesprochen.

Die Unified Modelling Language wird als bekannt vorausgesetzt und lediglich in Kapitel 4 vor der Erläuterung der speziellen UML-Diagrammtypen kurz eingeführt.

2.1 Komponentenbasierte Systeme

Komponentenbasierte Systeme wurden in Kapitel 1 kurz als ein möglicher Ansatz angesprochen, um die Wartbarkeit von Softwaresystemen zu verbessern. In diesem Abschnitt wird genauer auf den Komponentenbegriff und die einzelnen Konzepte eingegangen.

Szyperski definiert in [Szy98, S.34] eine Softwarekomponente folgendermaßen:

> ,,A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“

Der UML 1.4 Standard [OMG01, S.3/176] beinhaltet die folgende Definition:

> ,,A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.“

Beiden Definitionen ist gemeinsam, dass eine Komponente ein klar definierter Teil eines Systems ist, der über fest definierte Schnittstellen angesprochen wird und die Implementierung kapselt, sowie dass diese Komponente verteilt werden kann. Diese Kapselung der Implementierung wird genutzt, um eine Applikation in verschiedene Teile zu trennen. Diese Trennung einer Applikation in mehrere

möglichst unabhängige Teile wird bereits seit langem in Programmiersprachen unterstützt.

Um in prozeduralen Programmiersprachen Teile von Applikationen klar zu definieren, werden sie in Module ausgelagert. Die ersten Programmiersprachen, die Module unterstützten, sind Modula-2 [Wir83] und Ada [Uni82]. Module gruppieren mehrere Prozeduren und ermöglichen die getrennte Kompilierung der Module. In den objektorientierten Programmiersprachen gruppieren Module (in Java auch Pakete genannt) mehrere Klassen. Durch diese Module werden die Klassen allerdings nicht gekapselt. Es ist durchaus möglich, von Klassen aus anderen Modulen zu erben und so deren Funktionalität zu ändern oder zu erweitern. Durch diese Vererbung wird allerdings die Kapselung zerstört. Module bieten also nicht die benötigte Kapselung der Implementierung.

Im Gegensatz dazu sind die in Komponenten gekapselten Klassen nicht von außen ansprechbar, also weder durch direkte Aufrufe noch indem von ihnen geerbt werden kann. Komponenten sind eine ,,Black-Box“. Die Funktionalität einer Komponente ist nur über eine Schnittstelle ansprechbar, während die Implementierung unbekannt und nicht erreichbar ist. Meyer entwickelt in [Mey87] für diese fest definierten Schnittstellen den Begriff *Design by contract*. Mittels dieser ,,Verträge“ beschreibt eine Komponente, welche Bedingungen der Benutzer von Schnittstellen einhalten muss und welche Bedingungen die Komponente selber nach Abarbeitung der Aufgaben einhält. Diese Bedingungen werden zum Beispiel durch Vor- und Nachbedingungen definiert und überprüfen typischerweise Wertebereiche von Parametern und Rückgabewerte einer Methode. Eine Implementierung dieser Schnittstelle kann nun geändert werden, solange diese Bedingungen erfüllt bleiben bzw. die Vorbedingungen abgeschwächt und die Nachbedingungen verstärkt werden. Problematisch sind in dieser Hinsicht nicht-funktionale Bedingungen, wie zum Beispiel feste Antwortzeiten, Qualitätsmerkmale wie Mean-Time-Between-Failure (MTBF) oder Genauigkeit von mathematischen Fließkommaberechnungen, da diese nicht leicht überprüft werden können, sondern aufwendig beobachtet werden müssen. Weitere Informationen zu Kontrakten finden sich in [LRvV98, BJW99, Mey87].

Durch diese fest definierten Kontrakte ist es möglich, einzelne Komponenten getrennt und auch in verschiedenen Firmen zu entwickeln. Daraus ergibt sich die Möglichkeit, Komponenten von Dritt-Firmen einzukaufen und dann nach Bedarf diese Komponenten mittels ihrer Schnittstellen zu komplexen Applikationen zusammenzusetzen. Jede Komponente erwartet eine bestimmte aus Komponenten bestehende Umwelt. Eine Komponente kann nur dann ihre Funktion erfüllen, wenn Komponenten in der Umwelt zur Verfügung stehen, welche die von der Komponente benötigten Schnittstellen implementieren. Diese Abhängigkeiten werden *context-dependencies* genannt.

2.1.1 Komponentenverzeichnisse

Komponentenverzeichnisse verwalten Mengen von Komponenten mit ihren Schnittstellen bzw. Kontrakten und erlauben nach bestimmten Komponenten, Schnittstellen oder Kontrakten zu suchen. Da es typischerweise unwichtig ist zu wissen, um welche konkrete Komponente es sich handelt bzw. auf welchem konkreten Rechner die Komponente ausgeführt wird, kann aus einem Komponentenverzeichnis nach einer Komponente gesucht werden, die die gewünschten Fähigkeiten (z.B. Schnittstellen) und Attribute hat.

Die Suche kann zum einen über statische Namen geschehen. Dies wird zum Beispiel beim Domain Name Service Konzept (DNS) [Moc87] genutzt. Hier werden die einzelnen Server über einen Namen referenziert. Auch RMI bietet einen Naming Service (RMIRegistry [Sun98a, S.12–13]), in dem die Objekte über einen Namen angeboten und abgefragt werden können. Problematisch ist bei diesem statischen Konzept, dass der Name als Schlüssel für eine Funktionalität genutzt wird. Die durch den Namen angegebene Funktionalität kann nicht geändert werden, ohne dass auch allen Nutzern diese Änderung mitgeteilt wird, damit diese nicht die alte Funktionalität unter diesem Namen erwarten.

Besser ist es, die Komponenten über ihre angebotene Funktionalität zu identifizieren. Dieses Konzept wird zum Beispiel im CORBA Trader [ISO98] sowie bei der in Abschnitt 3.3 vorgestellten Jini-Technologie im Lookupdienst genutzt. Dadurch, dass zum Beispiel in dem Verzeichnis nach Schnittstellen gesucht werden kann, ist es möglich, den oben vorgestellten Vorteil von Komponenten, dass sich die Implementierung der Komponente ändern kann, besser zu nutzen. Außerdem besteht die Möglichkeit, den Namen und den Ort einer Komponente zu ändern, ohne dass die Funktionalität des Systems beeinträchtigt wird.

2.1.2 Service oder Server

In einem verteilten System werden *Dienste* (engl. *Service*) von *Servern* angeboten. Ein Dienst ist nach [CDK00, S.8] ein bestimmter Teil eines Computersystems, der eine Menge von Ressourcen verwaltet und deren Funktionalität für Benutzer und Applikationen anbietet. Ein Server ist ein Prozess auf einem vernetzten Computer, der Anfragen nach bestimmten Diensten von Prozessen auf anderen Computern erhält und passend beantwortet.

In diesen beiden Definitionen lässt sich der Unterschied zwischen serverbasierten und dienstbasierten Konzepten erkennen. Bei einem serverbasierten Konzept ist die Funktionalität an einen bestimmten Prozess auf einem bestimmten Computer gebunden, während das Angebot einer Funktionalität eines Dienstes unabhängig von der konkreten Realisierung durch einen Server-Prozess ist. An der im Beispielszenario angesprochenen XML-Parser-Komponente lässt sich der Unterschied deutlich machen. In Abbildung 2.1 wird die Situation des Beispiels dargestellt[1].

[1] Der dafür verwendete Diagrammtyp wird in Abschnitt 4.4 genau erläutert.

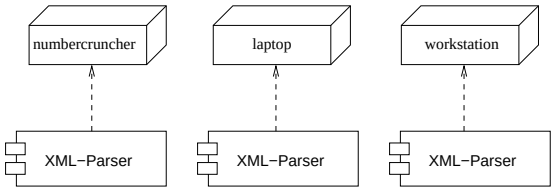

Abbildung 2.1: Service oder Server Beispiel

In einem Netz gibt es drei Rechner („numbercruncher“, „laptop“ und „workstation“) und auf jedem dieser Rechner läuft ein Server-Prozess, der die Funktionalität, ein XML-Dokument zu parsen, anbietet. Ein Nutzer möchte nun in diesem Szenario ein Dokument parsen. In einem serverbasierten Konzept schickt er Anfragen an die einzelnen Server-Prozesse (typischerweise ausgehend von dem Server, der ihm die höchste Geschwindigkeit verspricht). Je nach Verfügbarkeit erhält er eine Fehlermeldung oder eine korrekte Antwort. Bei einem dienstbasierten Konzept besteht die Möglichkeit, dass der Nutzer über ein Komponentenverzeichnis einen Dienst sucht, der die gewünschte Funktionalität bereitstellt. Solange ein Dienst im System existiert, wird der Nutzer eine Antwort bekommen und muss sich nicht mit der korrekten Behandlung von Fehlermeldungen und der konkreten Auswahl des Dienstes beschäftigen. Bei einem dienstbasierten Konzept besteht weiterhin die Möglichkeit, einfach Lastverteilung durchzuführen, indem die Aufgabe von einem Server erledigt wird, der, obwohl eigentlich langsamer, die Aufgabe durch eine geringere Auslastung schneller durchführen kann.

In einem dienstbasierten Konzept ist nicht wichtig, wer die Aufgabe durchführt und wo die Arbeit geleistet wird, sondern dass die Aufgabe korrekt durchgeführt wird. Der Nutzer muss im Endeffekt nicht wissen, wo der Server-Prozess ausgeführt wird, der die Aufgabe erledigt. Dadurch ist es möglich, dass trotz Ausfall einzelner Server der Dienst trotzdem noch angeboten werden kann.

2.2 Verteilte Systeme

In der Literatur finden sich viele Definitionen von verteilten Systemen. Diese Definitionen unterscheiden sich darin, welche Aspekte besonders hervorgehoben werden. Tanenbaum und van Steen geben in [TvS02, S. 2] eine allgemeine, aber sehr informelle Definition:

> „A distributed system is a collection of independent computers that appears to its users as a single coherent system.“

Diese Definition hebt den Aspekt heraus, dass die Natur und Eigenschaften eines verteilten Systems versteckt werden, so dass ein Benutzer den Eindruck hat, ein einziges System zu benutzen. Diese Maskierung der speziellen Eigenschaften verteilter Systeme führt zu einer Reihe von Problemen, die Waldo et al. in

[WWWK94] herausstellen. Es ist vor allem problematisch, dass die zusätzlichen Problemfälle verteilter Systeme gegenüber nicht-verteilten Systeme ausgeblendet werden und nicht angemessen darauf reagiert werden kann. Coulouris et al. definieren ein verteiltes System in [CDK00, S.1] als eine Menge von Komponenten, die miteinander kommunizieren und auf vernetzten Computern ausgeführt werden:

> ,,A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.“

Im Weiteren werden diese vernetzten Computer als Rechenknoten bezeichnet. Wie in der Einführung beschrieben, sind verteilte Systeme in der heutigen Zeit Stand der Technik. Die Systeme sind räumlich verteilt, es werden lokale Netzwerke genutzt und geographisch getrennte lokale Netze über große Entfernungen miteinander gekoppelt. Die einzelnen Teile des Systems verhalten sich reaktiv, reagieren also auf Anfragen von anderen Systemteilen, ohne dass es einen zentralen Koordinator geben muss.

Durch diese Verteilung ergeben sich zusätzliche Probleme bzw. Herausforderungen bei der Entwicklung von Softwaresystemen für diese verteilten Systeme.

2.2.1 Herausforderungen

Bei verteilten Systemen müssen im Gegensatz zu nicht verteilten Systemen einige zusätzliche Herausforderungen gelöst werden. Coulouris beschreibt in [CDK00, S.16-25] die grundsätzlichen Herausforderungen:

- Heterogenität:
  Heterogenität beschreibt die Zusammensetzung eines verteilten Systems in Hinsicht auf die unterschiedliche Hardware der einzelnen Rechenknoten sowie der verschiedenen Software (Betriebssystem und Netzwerksystem) auf diesen Rechenknoten. Durch den weltweiten Erfolg des Internets und der damit verbundenen Protokolle ist es problemlos möglich, Nachrichten und Daten zwischen verschiedenen Rechenknoten auszutauschen. Die Programmiersprache Java mit dem Konzept der virtuellen Maschine ermöglicht, dasselbe Programm auf verschiedenen Hardwarearchitekturen auszuführen. Mit Hilfe von *mobile Code* (Definition siehe [ATLLW96]) kann auch Software zwischen verschiedenen Rechenknoten ausgetauscht werden. Thorn gibt in [Tho97] eine Übersicht von Programmiersprachen, die mobilen Code unterstützen.

- Offenheit:
  Offenheit in verteilten Systemen drückt sich in den Möglichkeiten aus, wie neue Teile hinzugefügt und von verschiedenen Clients genutzt werden

können. Um Offenheit zu erreichen, müssen die Schnittstellen und Protokolle der einzelnen Systeme veröffentlicht sein, Kommunikation zwischen allen Teilen des Systems muss gewährleistet sein und es muss sichergestellt sein, dass die Schnittstellen und Protokolle korrekt erfüllt werden.

- Fehlerbehandlung:
  Während nicht verteilte Systeme durch (mit hohen Kosten verbundene) besondere Maßnahmen sehr ausfallsicher gestaltet werden können, ist es nicht möglich zu garantieren, dass einzelne Teile eines verteilten Systems nicht ausfallen. Allein durch die große Anzahl von Rechenknoten ist es sehr wahrscheinlich, dass mindestens ein Knoten zu einem bestimmten Zeitpunkt nicht verfügbar ist. Des Weiteren ergeben sich durch die Nutzung verteilter Systeme im Vergleich zu nicht verteilten Systemen neue Fehlerarten, die in Abschnitt 2.2.2 beschrieben werden. Diese Fehler müssen behandelt werden, um ein Funktionieren des Systems sicherzustellen.

- Nebenläufigkeit:
  Durch die gleichzeitige Verarbeitung auf verschiedenen Rechenknoten tritt das Problem der Nebenläufigkeit auf. Mehrere Clients wollen zum Beispiel eine Datei zur gleichen Zeit auf einem bestimmten System schreiben. Wenn beide Clients die Erlaubnis haben, diese Datei zu schreiben, kommt es zu fehlerhaften Ergebnissen. Durch die gleichzeitige Nutzung dürfen also keine inkonsistenten Ergebnisse entstehen.

- Transparenz:
  In den meisten Fällen soll den Nutzern des Systems der verteilte Charakter verborgen bleiben. Das verteilte System soll sich dem Benutzer als eine Einheit präsentieren und überwiegend die Besonderheiten verstecken. Die beiden wichtigsten Aspekte sind hier *Zugriffstransparenz* und *Ortstransparenz*. Zugriffstransparenz erlaubt, lokale und entfernte Ressourcen mit denselben Methoden zu benutzen. Ortstransparenz erlaubt, Ressourcen zu nutzen, ohne den Ort dieser Ressourcen kennen zu müssen. Zugriffstransparenz in einem verteilten System ist allerdings nicht immer zu erreichen und, wie von Waldo et al. in [WWWK94] beschrieben, nicht immer sinnvoll. Problematisch ist dabei, dass der Nutzer nicht auf Fehler reagieren kann, die durch die Nutzung von entfernten Ressourcen entstehen, da diese Informationen durch die gewünschte Zugriffstransparenz dem Nutzer nicht mitgeteilt werden. Eine mögliche Lösung bei einem Ausfall eines Systems, auf das zugegriffen werden soll, besteht darin, den Zugriff solange zu blockieren, bis das System wieder in Betrieb ist. Dieses Verhalten ist allerdings für die Mehrzahl der Anwendungsfälle nicht geeignet.

- Skalierbarkeit:
  Ein System ist skalierbar, wenn es auch bei einem Anwachsen der Nut-

zer und Ressourcen effizient arbeitet. Es kann damit jederzeit gesteigerten Anforderungen gerecht werden.

- Sicherheit:
  Sicherheit ist ein weiterer Aspekt von verteilten Systemen. Durch die Aufteilung der Informationen und deren Verarbeitung auf Rechenknoten, die eventuell sogar über ein potentiell unsicheres Kommunikationsmedium (z.B. Internet) verbunden sind, existieren sehr viel mehr Angriffspunkte als im nicht verteilten Fall.

- Konsistenzerhaltung:
  Konsistenzerhaltung in einem verteilten System ist von Coulouris nicht explizit in [CDK00, S.16-25] angesprochen worden, für diese Arbeit ist Konsistenzerhaltung in einem verteilten System allerdings ein Schlüsselthema. Konsistenzerhaltung bedeutet, dass die im Netz verteilten Daten so gespeichert werden, dass auch bei Ausfällen von Teilen des Systems jederzeit auf die Daten zugegriffen werden kann. Die Lese- und Schreiboperationen werden in einer durch ein Konsistenzmodell vorgegebenen Reihenfolge durchgeführt. Neben dem Ausfall von einzelnen Rechenknoten im verteilten System ist eine explizite Betrachtung der Konsistenzerhaltung in einem partitionierten Netzwerk nötig.

Für diese Arbeit ist das Problem der Heterogenität durch die Nutzung von Java und dem Virtual Machine Konzept größtenteils überwunden. Von den anderen Herausforderungen werden in dieser Arbeit vor allem Transparenz, Skalierbarkeit, Fehlerbehandlung, Nebenläufigkeit und Konsistenz thematisiert. Transparenz, vor allem Ortstransparenz, wird durch die in dieser Arbeit in Kapitel 5 beschriebene Laufzeitumgebung zum Betrieb von Komponenten in einem verteilten System gewährleistet. Skalierbarkeit wird dadurch erreicht, dass die zu betreibenden Komponenten auf das gesamte, durch die Laufzeitumgebung verwaltete System gemäß bestimmter Restriktionen verteilt werden und so die Arbeit einfach auf mehrere Rechenknoten verteilt wird. In dieser Arbeit werden in der Laufzeitumgebung zwei Konzepte (Abschnitt 5.1.2 und 5.5) vorgestellt, um verteilte Daten hinsichtlich zweier in Abschnitt 2.3.1 vorgestellten Konsistenzmodelle konsistent zu halten. Diese Konsistenz der Daten ist zwingend notwendig, um die Funktionalität des Systems zu gewährleisten.

2.2.2 Fehlertypen

Bei der Fehlerbehandlung müssen die in verteilten Systemen auftretenden Fehlerklassen beachtet werden. Tanenbaum [TvS02, S.364ff] und Birman [Bir96, S.29ff] unterscheiden die in Tabelle 2.2 dargestellten Fehlerklassen:

Ein *Crash failure* entsteht, wenn ein Rechenknoten, der sich bis dahin korrekt verhalten hat, aufhört zu arbeiten. Dieser Rechenknoten reagiert auf keinerlei

Fehlerklasse	Beschreibung
Crash failure	Ein Knoten stoppt, aber hat sich bis dahin korrekt verhalten.
Omission failure	Ein Knoten antwortet nicht auf eingehende Anfragen.
Timing failure	Die Antwort eines Knotens liegt außerhalb des spezifizierten Zeitintervalls.
Network Failures	Das Netzwerk verliert Nachrichten zwischen den Knoten.
Byzantine failure	Ein Knoten erzeugt willkürlich Nachrichten.

Tabelle 2.2: Fehlerklassen in verteilten Systemen

Anfragen mehr. Deshalb können andere Rechenknoten nur aus dem Ausbleiben von Antworten oder einer Zeitüberschreitung schließen, dass dieser Rechenknoten ausgefallen ist.

Ein *Omission failure* beschreibt die Situation, wenn ein funktionsfähiger Rechenknoten nicht auf Anfragen antwortet. Dies kann daran liegen, dass der Rechenknoten die Anfrage nicht bekommen hat (*Receive omission failure*) oder dass er die Anfrage korrekt erhalten und bearbeitet hat, er aber die Antwort nicht versenden konnte (*Send omission failure*). Diese Probleme treten bei dem angefragten Rechenknoten auf und sind nicht durch das Netzwerk bedingt.

Timing failures treten auf, wenn die Antwort eines Rechenknotens außerhalb des Zeitrahmens, der in einem Protokoll spezifiziert wurde, bei dem anfragenden Knoten eintrifft. Dieser Fall tritt zum Beispiel ein, wenn ein Knoten überlastet ist und die Anfragen deshalb erst sehr spät und langsam bearbeiten kann.

Ein *Network failure* beschreibt die Situation, wenn das Kommunikationsnetzwerk Nachrichten durch Ausfälle nicht korrekt an den Empfänger sendet. Ein Sonderfall ist eine Netzwerkpartition. Dieser Fall tritt ein, wenn ein zusammengehöriges Netzwerk durch Ausfall von Teilen des Netzwerks in mehrere Teile gespalten wird. Nachrichten innerhalb der Teilnetze kommen an, aber Nachrichten von einem Teilnetz in ein anderes Teilnetz nicht.

Ein *Byzantine failure* tritt auf, wenn ein Rechenknoten willkürlich Nachrichten verschickt, die er nicht senden darf. Diese Nachrichten sind eventuell gar nicht als fehlerhafte Nachrichten erkennbar oder werden sogar absichtlich verschickt, um Ergebnisse anderer Rechenknoten zu verfälschen.

Alle außer dem letzten Fehlertyp werden durch die in der Arbeit beschriebenen Konzepte behandelt.

2.2.3 Synchrone und Asynchrone Systeme

Verteilte Systeme können sich stark unterscheiden. In Hinsicht auf Prozessausführungsdauer, Nachrichtensendedauer und Zeitverschiebung haben Hadzilacos und Toueg in [HT94, S.8] *synchrone* und *asynchrone* verteilte Systeme definiert. Für ein synchrones verteiltes System sind die folgenden Schranken bekannt:

- Die Zeit, um einen bestimmten Schritt von einem Prozess auszuführen, hat eine bekannte untere und obere Schranke.
- Jede Nachricht, die über einen Kanal versandt wird, wird innerhalb einer bekannten Zeitschranke empfangen.
- Jeder Prozess hat eine lokale Zeit, deren Verschiebungsgeschwindigkeit zur korrekten Zeit eine bekannte Schranke hat.

Diese Anforderungen erlauben die Nutzung von *timeouts*, um auf den Ausfall von Rechenknoten zu schließen. Allerdings ist es sehr schwierig, ein verteiltes System zu entwerfen, für das die oben beschriebenen Schranken bekannt sind und deren Einhaltung garantiert werden kann. Systeme, welche die obigen Bedingungen nicht erfüllen, werden als asynchrone verteilte Systeme bezeichnet. In [FLP85] beweisen Fischer, Lynch und Paterson, dass es in einem asynchronen System keinen Algorithmus für die Erreichung von Konsens gibt, der einen unangekündigten Ausfall eines Rechenknotens toleriert.

Dies führt zur Definition der verschiedenen Konsistenzmodelle in verteilten Systemen im nächsten Abschnitt.

2.3 Konsistenz in verteilten Systemen

Wie in Abschnitt 2.2.1 dargestellt, entsteht durch die Nutzung eines verteilten Systems das Problem, gemeinsam genutzte Daten auf mehrere Rechenknoten zu replizieren und dabei konsistent zu halten.

Die oben genannten Ziele und die gewünschte Konsistenz im verteilten System beeinflussen sich gegenseitig. Je geringer die benötigte Konsistenz ist desto mehr lassen sich die Vorteile eines verteilten Systems nutzen, da weniger Leistung benötigt wird, um die gewünschte Konsistenz zu erhalten.

Der Brockhaus [Bro00] definiert Konsistenz als:

> „... Widerspruchsfreiheit eines axiomatischen Systems.“

Diese Widerspruchsfreiheit betrifft nach [TvS02, S.297] bei der Replikation von Daten die Ergebnisse von Lese- und Schreiboperationen auf diesen gemeinsam

genutzten Daten. Die Daten werden von einem Datensystem verwaltet. Konsistenzmodelle beschreiben eine Art von Vertrag zwischen Prozessen und dem Datensystem. Wenn der Prozess den Regeln des Konsistenzmodells folgt, dann garantiert das Datensystem ein korrektes Verhalten. Im Vergleich dazu betrachten Transaktionen in Datenbanksystemen nicht einzelne Operationen sondern Gruppen von Operationen.

Im nächsten Abschnitt werden verschiedene Konsistenzmodelle beschrieben. Umsetzungen zweier dieser Konsistenzmodelle werden in den Abschnitten 5.1.2 und 5.5 genau erläutert.

2.3.1 Konsistenzmodelle

Im einfachen Fall eines einzelnen Rechenknotens mit Daten, die nur einen Prozess auf diesem Rechner betreffen, ist strikte Konsistenz, wie sie später noch erläutert wird, der Normalfall und wird vom Programmierer auch erwartet. Bei mehreren Prozessen auf einem Einzelrechner ist es bereits schwieriger, strikte Konsistenz zu erreichen. In einem verteilten System ist strikte Konsistenz allerdings auf Grund des Fehlens einer globalen Zeit und der räumlichen Entfernung zwischen den Rechnern unmöglich [TvS02, S.300].

Im Folgenden werden verschiedene Konsistenzmodelle in Anlehnung an [TvS02, S.298] vorgestellt. Diese Konsistenzmodelle arbeiten ohne Synchronisierungsvariablen. Konsistenzmodelle mit Synchronisierungsvariablen sind *Weak Consistency, Release Consistency* und *Entry Consistency*. Diese Konsistenzmodelle werden im Folgenden nicht erläutert, da eine Synchronisierungsvariable, die alle lokale Kopien der Daten synchronisiert, in einem verteilten System, das nicht über einen gemeinsamen Hauptspeicher verfügt, nicht realisierbar ist. Erläuterungen zu diesen Konsistenzmodellen finden sich in [TvS02, S.308ff].

Tabelle 2.3 ordnet die im Folgenden beschriebenen Konsistenzmodelle absteigend nach den geforderten Restriktionen.

Strikte Konsistenz
Lineare Konsistenz
Sequentielle Konsistenz
Kausale Konsistenz
FIFO Konsistenz
Eventuelle Konsistenz

Tabelle 2.3: Konsistenzmodelle

Strikte Konsistenz

Unter *strikter Konsistenz* wird verstanden, dass ein Lesezugriff auf Datum x den Wert zurück liefert, der durch den letzten vorhergehenden Schreibzugriff ge-

setzt wurde. Diese Definition verlangt, dass es eine globale Zeit gibt, um genau herauszufinden, welches der letzte vorhergehende Schreibzugriff ist. In einem Ein-Prozessor System ist dieses Verhalten der Normalfall. Im verteilten System gibt es allerdings keine globale Zeit und durch die Entfernung der einzelnen Rechenknoten kommt es zu einer gewissen Laufzeit der Nachrichten. Dadurch ist eine strikte Konsistenz in einem verteilten System nicht erreichbar (für eine Erläuterung siehe [TvS02, S.299]).

Zusammenfassend kann gesagt werden, dass bei strikter Konsistenz alle Schreibergebnisse sofort für alle Prozesse sichtbar sind und eine Reihenfolge der Zugriffsoperationen in einer absoluten globalen Zeit existiert.

Sequentielle Konsistenz

Sequentielle Konsistenz, definiert von Lamport in [Lam79], ist etwas weniger restriktiv als strikte Konsistenz. Wenn die folgende Bedingung erfüllt ist, bezeichnet man ein Datensystem als sequentiell konsistent:

> ,,The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its programs.“

Diese Bedingung bedeutet, dass, wenn mehrere Prozesse nebenläufig auf möglicherweise verschiedenen Rechenknoten ausgeführt werden, jede erlaubte Reihenfolge der Operationen möglich ist, aber alle Prozesse *dieselbe* Reihenfolge der Operationen sehen müssen. Eine Reihenfolge ist erlaubt, wenn die einzelnen Operationen jedes Prozesses in derselben Reihenfolge vorkommen, wie sie der Prozess ausgelöst hat.

Lineare Konsistenz

Lineare Konsistenz [HW90] fügt der Bedingung der sequentiellen Konsistenz den Zusatz hinzu, dass die Reihenfolge der Operationen konsistent ist mit der Reihenfolge der Zeitpunkte, in der die Operationen ausgeführt wurden. Diese zusätzliche Bedingung erfordert, dass es eine globale Uhr mit einer endlichen Genauigkeit gibt bzw. die lokalen Uhren der einzelnen Rechenknoten innerhalb einer bestimmten Abweichung synchronisiert sind. Lineare Konsistenz ist stärker als sequentielle Konsistenz und schwächer als strikte Konsistenz.

Kausale Konsistenz

Hutto und Ahamad [HA90] (nach [TvS02, S.305]) definieren Datenverwaltungssysteme als *kausal konsistent*, wenn Schreiboperationen, die von anderen Operationen abhängig sind, in allen Prozessen in derselben Reihenfolge auftreten.

Schreiboperationen, die nicht abhängig sind[2], können bei jedem Prozess in einer beliebigen Reihenfolge auftreten. Abhängige Operationen sind dadurch charakterisiert, dass sie Daten bearbeiten, die vorher von einer anderen Operation geliefert werden. Wenn Daten erst gelesen, dann bearbeitet und zuletzt zurückgeschrieben werden, ist die Schreiboperation von der Leseoperation abhängig. Diese Leseoperation muss dann in allen Prozessen vor der Schreiboperation abgearbeitet werden. Kausale Konsistenz ist schwächer als sequentielle Konsistenz, da unabhängige Operationen beliebig verschachtelt abgearbeitet werden können.

Kausale Konsistenz ist vergleichbar mit serialisierbaren Schedules bei der Abarbeitung von Transaktionen in Datenbanksystemen. Dort können alle Lese- und Schreibzugriffe, die nicht dieselben Daten betreffen und damit voneinander unabhängig sind, innerhalb der Transaktion in einer beliebigen Reihenfolge ausgeführt werden.

FIFO Konsistenz

Im Gegensatz zu kausaler Konsistenz können bei *FIFO Konsistenz* auch kausal abhängige Schreiboperationen, die von verschiedenen Prozessen versandt werden, bei allen Prozessen in unterschiedlicher Reihenfolge eintreffen. Lediglich die Schreiboperationen eines Prozesses müssen von allen Prozessen in der gleichen Reihenfolge gesehen werden. Dieses von Lipton und Sandberg in [LS88] (nach [TvS02, S.306]) in der Domäne Shared-Memory-Systeme als *PRAM Consistency* beschriebene Konsistenzmodell ist recht einfach zu implementieren, da jeder Prozess seine Schreibzugriffe einfach mit einer fortlaufenden Nummer versehen kann und jeder andere Prozess die Schreibzugriffe dieses Prozesses anhand der Nummern in der richtigen Reihenfolge ausführen kann.

Eventuelle Konsistenz

Eventuelle Konsistenz hat etwas andere Ziele und eine andere Definition als die bisher behandelten Konsistenzmodelle. Die oben dargestellten Konsistenzmodelle sind auch in den weniger restriktiven Varianten immer noch sehr einschränkend. Viele verteilte Systeme benötigen nicht so eine starke Konsistenz, das heißt die Änderungen müssen nicht immer direkt oder sehr schnell bei den einzelnen Rechenknoten des verteilten Systems ankommen. In diesem Fall reicht es, wenn die Änderungen erst mit Verzögerung bei den einzelnen Rechenknoten eintreffen.

Tanenbaum beschreibt in [TvS02, S.317], dass bei eventueller Konsistenz über einen gewissen Zeitraum, wenn keine Änderungen vorkommen, alle Repliken schrittweise konsistent werden. Diese Form der Konsistenz wird durch stabilisierende Algorithmen, erstmals beschrieben von Dijkstra in [Dij74], erreicht. Dolev

[2] Unabhängige Operationen werden nebenläufig genannt.

erläutert in [Dol00] die Grundlagen und das Design von selbst-stabilisierenden Algorithmen.

3 Technologien für verteilte Systeme und verwandte Ansätze

In diesem Kapitel werden in den ersten drei Absätzen Technologien für Softwaresysteme im Bereich verteilter Systeme thematisiert. Im ersten Abschnitt werden Web-Services als eine Möglichkeit für die Kommunikation zwischen heterogenen Systemen dargestellt. Enterprise JavaBeans, vorgestellt in Abschnitt 3.2, ist ein Beispiel für eine zentralisierte Serverarchitektur, die den kompletten Lebenszyklus von Objekten in einem verteilten System abdeckt. Der darauf folgende Abschnitt erklärt die Jini-Technologie für ein verteiltes System, das grundlegende Design sowie die verschiedenen Infrastrukturdienste. In den folgenden Abschnitten werden aufbauend auf der Jini-Technologie verwandte Arbeiten für die Unterstützung bei der Entwicklung und dem Betrieb von Softwaresystemen erläutert. In Abschnitt 3.3.1 wird das Jini-Plugin für die NetBeans Entwicklungsumgebung und in Abschnitt 3.3.2 das RIO Toolkit beschrieben. Das Kapitel schließt mit einem Fazit, in dem die Einschränkungen der vorgestellten Ansätze in Bezug auf die Ziele dieser Arbeit thematisiert werden.

3.1 Web-Services

Web-Services sind ein aktuelles Schlagwort in der Software-Entwicklung. Vaughan-Nichols erläutert in [VN02] den Begriff Web-Service und die zugehörigen Technologien. Der Begriff Web-Service hat nichts mit dem World Wide Web (WWW) zu tun, sondern Web wird in diesem Zusammenhang als Synonym für das Internet bzw. im Allgemeinen für Netzwerk genutzt. Eine genaue Definition für Web-Services gibt es derzeit noch nicht, allerdings kristallisieren sich einige Eigenschaften aus den verschiedenen Beschreibungen heraus. Grundlegende Infrastruktur für Web-Services ist das TCP/IP Protokoll. Aufbauend auf dieser Grundlage können die Dienste über das Internet angeboten sowie in einem lokalen Intranet verwendet werden. Für die Kommunikation der Dienste wird ein Remote-Procedure-Call (RPC) Verfahren auf Basis von XML genutzt. Dieses Verfahren heißt Simple-Object-Access-Protocol (SOAP) [BEK+00]. Bei diesem Verfahren werden die Parameter in XML-Daten konvertiert und als Nachricht an den Empfänger verschickt. Dieser konvertiert die Parameter wieder zurück, bear-

beitet die Anfrage und schickt den in XML konvertierten Rückgabewert zurück. Durch das Konvertieren der Daten in XML lassen sich so plattform- und programmiersprachenübergreifende verteilte Applikationen entwickeln. Dieses Verfahren ähnelt dem CORBA Ansatz der Intermediate Description Language (IDL) [OMG99]. Die Dienste werden in einem (oder mehreren) zentralen Verzeichnissen verwaltet, die über das ,,Universal Description, Discovery and Integration“ UDDI-Protokoll [ARI00] angesprochen werden.

Basierend auf diesen Standardtechnologien gibt es zur Zeit zwei umfassende Umgebungen: .NET der Firma Microsoft [NET00] und Sun ONE [ONE01] der Firma Sun Microsystems. Beide Infrastrukturen benutzen dieselben Technologien (XML, SOAP etc.). Sun ONE nutzt die Programmiersprache Java [GJS96], Microsoft im Wesentlichen die Programmiersprache C#. Diese beiden Programmiersprachen ähneln sich sehr. Sogar die Syntax ist auf Grund der gemeinsamen Ahnen C und C++ sehr ähnlich. Auch werden die in den beiden Programmiersprachen entwickelten Programme nicht in Maschinensprache übersetzt, sondern in einen plattformunabhängigen *Byte-Code* [LY99] transferiert. Bei der Ausführung wird der entstehende Byte-Code dann durch eine virtuelle Maschine interpretiert. Durch dieses Konzept können die Programme ohne Änderung auf allen Plattformen ausgeführt werden, für die eine passende virtuelle Maschine existiert. Dies wird mit dem Schlagwort ,,Write once, run everywhere!“ ausgedrückt. Der einzige große Unterschied zwischen den beiden Virtual-Machine-Konzepten ist die explizite Unterstützung von unterschiedlichen Programmiersprachen bei der Common Language Runtime (CLR) der Microsoft .NET-Plattform. Hier besteht die Möglichkeit, dass in verschiedenen Programmiersprachen entwickelte Objekte sich innerhalb eines Programms gegenseitig aufrufen können und Klassen von Klassen erben können, die in einer anderen Programmiersprache geschrieben sind. Beim Design der Java virtuellen Maschine wurde diese Möglichkeit nicht direkt vorgesehen, allerdings gibt es einige Projekte, die für verschiedene Programmiersprachen Compiler entwickeln, die Byte-Code für die Java virtuelle Maschine erzeugen. [Tol02] enthält eine Übersicht.

Ein weiterer großer Unterschied besteht in den von den Unternehmen bereitgestellten Diensten. Beide Umgebungen wollen dem Nutzer von Web-Services eine Online-Identität geben, so dass potentielle Nutzer sich nicht bei jedem genutzten Dienst neu authentifizieren müssen, sondern die Authentifizierung bei einer zentralen Instanz durchführen. Die beiden Konzepte unterscheiden sich dahingehend, wer diese zentrale Authentifizierungsstelle betreibt. Beim .NET-Framework soll dies der Passport-Dienst von Microsoft sein, bei Sun ONE soll diese Aufgabe dezentral bei den Mitgliedern der ,,Liberty Alliance” durchgeführt werden.

Die Web-Service-Technologie ermöglicht die Kommunikation zwischen heterogenen Systemen. Es gibt allerdings keine besondere Unterstützung für Fehlertoleranz und Wartbarkeit. Für das Erreichen der Ziele dieser Arbeit sind Web-Services in der derzeitigen rudimentären Form nicht geeignet.

3.2 Enterprise JavaBeans

Enterprise JavaBeans (EJB) ist eine serverseitige Komponentenarchitektur zur Entwicklung verteilter, transaktionsorientierter Anwendungen in Java. Entwurfsziele der EJB-Architektur sind Eignung für den unternehmensweiten Einsatz und die Steuerung unternehmenskritischer Geschäftsprozesse. Grundlage für die folgenden Erläuterungen ist die Spezifikation in der Version 2.0 [DYK01].

Basis dieser Architektur sind Datenbanken, welche die Daten für Geschäftsprozesse persistent speichern. Aus diesen Daten werden vom Enterprise JavaBeans Container (innerhalb eines speziellen Application Server) Objekte erstellt, die von anderen Programmen im verteilten System genutzt werden können. Diese Objekte haben einen Zustand und werden in der EJB Nomenklatur ,,Entity Beans“ genannt. Auf diesen Entity Beans können grundlegende Aktionen aufgerufen werden, um die zugehörigen Daten zu bearbeiten. Da diese Objekte in einer Datenbank gespeichert werden, müssen sie zum Beispiel über einen Primärschlüssel verfügen. Zusätzlich gibt es ,,Session Beans“, welche die Vorgänge, die mehrere Entity Beans betreffen, bündeln.

Beispiele für Entity Beans sind Zimmer in einem Hotel, Plätze in einem Flugzeug etc. Ein Beispiel für ein Session Bean ist, die Aufgabe eine Reise zu buchen. Im Verlauf dieses Vorgangs werden Hotelzimmer belegt und Plätze im Flugzeug reserviert.

Entity Beans existieren nur in diesem Enterprise JavaBeans Container. Um Programmen im Netzwerk Zugriff auf diese Daten zu geben, werden für jeden Beantyp Schnittstellen sowie Stubs und Skeletons generiert. Es gibt zwei verschiedene Arten von Schnittstellen. Die ,,Remote“-Schnittstelle bietet Zugriff auf die Attribute des Beans, also zum Beispiel die Möglichkeit, das Geburtsdatum eines Angestellten zu ändern. Die ,,Home“-Schnittstelle betrifft den Lebenszyklus eines Beans. Mit Hilfe dieser Schnittstelle können Beans erzeugt und gelöscht werden sowie Entity Beans, die bestimmte Attributbedingungen erfüllen, gesucht werden.

Der Enterprise JavaBeans Container stellt grundlegende Funktionalität für die Verbindung von Objekten und Datenbank zur Verfügung. Er ist für das Mapping der relationalen Daten in Objekte zuständig und muss die Persistenz der Objekte in der Datenbank sicherstellen. Des Weiteren muss er die Transaktionen der Datenbank unterstützen und erweitern. Zusammengenommen ist er für den gesamten Lebenszyklus der Objekte zuständig. Diese Aufgaben werden transparent durch den Container durchgeführt, die einzelnen Beans können also keinerlei Einfluss darauf nehmen. Die Beans erhalten auch keinerlei Informationen über den verteilten Charakter des Systems. Die durch die Verteilung entstehenden Probleme werden alle über den Container bearbeitet und maskiert. Das einzelne Bean hat keinerlei Möglichkeiten, auf die Bearbeitung dieser Probleme einzuwirken.

Bei Enterprise JavaBeans werden nur die Programme verteilt, die auf die Datenobjekte im zentralen Enterprise JavaBeans Container zugreifen. Die Appli-

kationen können überall in einem verteilten System ausgeführt werden, allerdings müssen sie auf die Daten immer durch den Enterprise JavaBeans Container zugreifen. Durch diesen Ansatz muss in jedem System ein großer Applikationsserver existieren. Eigentlich ist die Enterprise JavaBeans Architektur eine normale n-tier-Architektur, basierend auf einem relationalen Datenbankserver und einer mächtigen objektorientierten Zugriffsschicht. Dadurch sind der Datenbankserver und der JavaBeans Container problematisch in Hinsicht auf Fehlertoleranz und Skalierbarkeit. Die Geschwindigkeiten des Datenbankservers und des JavaBeans Containers bilden eine obere Grenze für die Geschwindigkeit des Gesamtsystems. Bei einem Ausfall eines der beiden genannten Systeme ist das gesamte verteilte System nicht mehr funktionsfähig.

Des Weiteren ist durch das, zwar den Entwicklungsprozess erleichternde, automatische Generieren von Stubs und Skeletons die Flexibilität stark eingeschränkt. So besteht zum Beispiel nicht die Möglichkeit, anstatt entfernter Zugriffe auf die Daten, durch die Nutzung von mobilem Code, Zugriffe auf die Daten lokal auszuführen. Bei einem Benutzer-Entity-Bean gibt es zum Beispiel die Methoden `getBirthday()` und `getAge()`. Bei EJB's müssen beide Methoden Anfragen an den Server stellen, die Methode `getAge()` könnte allerdings problemlos auf dem lokalen Rechner aus der zwischengespeicherten Abfrage einer früheren `getBirthday()`-Abfrage das Alter des Benutzers berechnen. Allgemein ist es nicht möglich, das im nächsten Abschnitt vorgestellte Konzept eines Smartproxies zu verwirklichen. Diese Einschränkung führt zu einer starken Festlegung auf das Konzept der Enterprise JavaBeans, ohne die Möglichkeit zu haben, für bestimmte Probleme eine elegantere Lösung zu nutzen.

Die Enterprise JavaBeans Technologie bietet mehr Möglichkeiten als die im vorherigen Abschnitt beschriebenen Web-Services. EJB's basieren allerdings auf einer n-tier-Architektur, die Probleme hinsichtlich Fehlertoleranz und Skalierbarkeit aufweist. Zudem ist es nicht möglich mobilen Code zu nutzen. Aus diesen Gründen sind EJB's für diese Arbeit nicht geeignet.

3.3 Jini

In [WWWK94] beschreiben Waldo et al. die Probleme verteilter Systeme und deren inadäquate Lösung in verschiedenen Konzepten. Ein vielen Lösungen gemeinsames Problem ist der Versuch, Zugriffstransparenz zu erreichen. Es wird damit versucht, dem Programmierer vorzuspiegeln, dass er keine verteilten Systeme nutzt. Diese Maskierung der Details von verteilten Systemen birgt allerdings viele Gefahren, da der Programmierer nicht auf die gegenüber nicht-verteilter Software zusätzlichen Probleme (z.B. Ausfälle einzelner Rechenknoten oder Abstürze von Teilkomponenten) eingehen kann, da er keine Informationen über diese Fehler erhält.

In der folgenden Zeit entwickelten die Autoren aufgrund der oben dargestell-

ten Probleme für die Programmiersprache Java die Jini-Technologie [AOS+99, Wal99, Sun01a]. Die vorliegende Arbeit basiert auf der Spezifikation 1.2.1 der Jini-Technologie. Die Jini-Technologie geht explizit auf die oben genannten Probleme ein und zwingt auch den Entwickler von Komponenten (in Jini *Dienste* genannt), auf diese Probleme zu achten. Grundlegendes Konzept ist, dass die vom Nutzer eines Jini-Dienstes genutzte Schnittstelle nicht gleich der Netzwerkschnittstelle des Dienstes ist. Durch dieses Konzept besteht die Möglichkeit, zusätzliche Operationen durchzuführen, die Fehler der Netzwerkkommunikation maskieren oder einen Teil der Aufgaben bereits erledigen, bevor die Netzwerkkommunikation durchgeführt wird.

Ein Komponentenverzeichnis (der Lookupdienst) verwaltet die verschiedenen Dienste im System. Die Kommunikation innerhalb des Netzes basiert auf Multicast-Nachrichten [Wyb90]. Multicast-Nachrichten bieten eine effiziente 1:n-Kommunikation im lokalen Netz. Die einzelnen Prozesse in einem lokalen Netz können sich an einer bestimmten IP-Adresse und Portnummer anmelden und erhalten alle Nachrichten, die von anderen Prozessen an diese IP-Adresse geschickt werden. Durch die Nutzung der Multicast-Nachrichten ist es einfach und kostengünstig, mehrere Prozesse im lokalen Netz gleichzeitig zu benachrichtigen.

Jeder Lookupdienst verschickt regelmäßig Multicast-Nachrichten über sein Vorhandensein ins Netz, so dass interessierte Programme auf diese Nachrichten reagieren können. Grundlegendes Konzept ist die Benutzung von Mietdauern (Leasing). Jede Komponente, die im System ihre Dienste anbieten will, kann diese Dienste nur für eine gewisse Zeitdauer an den Lookupdiensten anmelden. Dienste, die noch bei keinem Lookupdienst angemeldet waren, bekommen eine eindeutige Identifikationsnummer (ServiceID), die in der weiteren Lebenszeit genutzt wird. Vor Ablauf der Mietdauer müssen die Dienste diese Mieten verlängern, um weiter an den Lookupdiensten angemeldet zu bleiben. Ansonsten vermuten die Lookupdienste, dass die Dienste ausgefallen sind oder nicht mehr erreicht werden können. Aus dieser Vermutung resultiert eine Entfernung dieser Dienste aus den internen Tabellen der Lookupdienste, so dass bei Anfragen nach einem bestimmten Diensttyp diese ausgefallenen Dienste nicht mehr in der Antwort enthalten sind. Wenn ein Dienst, der vom Netz getrennt wurde, wieder Teil des Netzes wird, trägt er sich wieder bei den Lookupdiensten ein.

Die Verwaltung der Mietdauern und deren passende Verlängerung können die Dienste an eine Hilfsklasse übertragen, die im selben Prozess wie der Dienst ausgeführt wird. Alternativ gibt es die Möglichkeit, diese Mietverlängerung an einen anderen Dienst zu delegieren, um zum Beispiel Diensten auf besonders ressourceneingeschränkten Systemen die Möglichkeit zu geben, diese Aufgabe nicht selber durchführen zu müssen. Wenn die Mietverlängerung an einen anderen Dienst delegiert worden ist, muss sich der Dienst auch periodisch bei diesem anderen Dienst melden, allerdings können die Mietdauern dabei länger sein. Problematisch ist bei dieser Möglichkeit, dass bei einem Ausfall die Zeitdauer größer ist, bis der Lookupdienst den entsprechenden Eintrag aus seinen internen Tabellen

entfernt.

Es besteht die Möglichkeit, Dienste in Gruppen einzuteilen. Gruppen identifizieren sich bei Jini über eine Zeichenkette. Es gibt zwei Sondergruppen. Eine Sondergruppe ist die öffentliche Gruppe (*public*). Die andere Sondergruppe umfasst alle anderen Gruppen (*all*). Jeder Lookupdienst wird für die Gruppen, die er verwalten soll, konfiguriert und reagiert entsprechend nur auf Anfragen (Anmeldung/Suche) für diese Gruppen.

Ein Dienst für ein Jini-Netzwerk besteht typischerweise aus zwei Teilen: Der *Proxy* ist der Teil des Dienstes, der auf dem Client ausgeführt wird, und mit dem *Backend*, das auf einem Server ausgeführt wird, kommuniziert.

Proxy: Jeder Proxy ist eine Instanz einer serialisierbaren Klasse und implementiert eine (oder mehrere) dienst-spezifische Schnittstellen[1]. Dieser Proxy wird in serialisierter Form [Sun98b] zusammen mit der oben beschriebenen Mietdauer und zusätzlichen Attributen[2] bei den Lookupdiensten angemeldet. Der Proxy wird auf dem Client ausgeführt und kann die Aufgaben direkt auf dem Client erfüllen oder alternativ die Anfragen an sein Backend zur Durchführung delegieren.
Wenn ein Proxy eine nicht-triviale Funktionalität erfüllt, also zum Beispiel auf dem Client direkt einen Teil der Berechnung ausführt oder ein Backend unter mehreren Backends auswählt, wird in dieser Arbeit von einem Smartproxy gesprochen. Ledru beschreibt zum Beispiel in [Led02] einen Smartproxy eines Druckdienstes, der bei Ausfall eines Druckers den Druckjob auf einem anderen passenden Drucker ausgibt.

Backend: Das Backend muss nicht in Java programmiert sein. Es verarbeitet auf Veranlassung durch einen seiner Proxies ankommende Anfragen. Um die korrekte Verarbeitung mehrerer gleichzeitig ankommender Anfragen der Proxies muss sich das Backend selber (durch Synchronisierung oder mittels Sperrvariablen) kümmern. Wenn das Backend ein Java-Programm ist, wird die Kommunikation zwischen Proxy und Backend typischerweise durch Remote Method Invocation (RMI) [WRW96, Sun98a] realisiert. Die Kommunikation kann allerdings auch problemlos über einfache TCP/IP-Verbindungen verlaufen. Es ist auch möglich, dass der Proxy über CORBA mit seinem Backend kommuniziert.

In den Lookupdiensten kann nach einem Dienst bzgl. der Gleichheit der ServiceID, der Attribute und nach einer zu erfüllenden Schnittstelle gesucht werden. Durch die Suche nach einer Schnittstelle besteht, wie in Abschnitt 2.1 für Komponenten bereits erläutert, die Möglichkeit, die Implementierung des Dienstes zu ändern, ohne dass andere Dienste durch diese Änderung betroffen sind. Wenn

[1] vgl. komponentenbasierte Systeme in Abschnitt 2.1
[2] Diese Attribute müssen von net.jini.core.entry.Entry abgeleitet sein.

ein Dienst nun eine Antwort auf eine Anfrage von den Lookupdiensten erhält, wird unter anderem, falls vorhanden, ein passender serialisierter Proxy zurückgeschickt. Wenn dem nutzenden Dienst Klassen fehlen, um den Proxy zu deserialisieren, werden diese Klassen über den RMI-ClassLoading Mechanismus von einem Webserver geladen. Diese Klassendateien müssen vorher von einem Administrator auf diesen Webserver kopiert werden. Bei weiteren Anfragen müssen dann diese Klassen aufgrund des Klassen-Caching der Java-Virtual-Machine nicht mehr geladen werden.

Abbildung 3.1 beschreibt das Zusammenwirken der verschiedenen Teilsysteme. Als Beispiel dient der XML-Parser-Dienst des Beispielszenarios. Das Backend des Dienstes wird in diesem Beispiel vom Proxy über RMI angesprochen. Das Backend liest eine Datei von dem Rechner, auf dem das Backend ausgeführt wird, und parst diese Datei dann. Zu Anfang trägt der XML-Parser-Dienst die Proxy-Instanz in den Lookupdienst ein. Wenn nun ein Client im zweiten Schritt in dem Lookupdienst nach diesem Dienst bzw. der Schnittstelle sucht, erhält der Client im dritten Schritt diese Instanz. Da ihm die zur Deserialisierung nötige Klasse `XMLProxy.class` fehlt, lädt er sie vom Webserver und erhält nach der Deserialisierung die Instanz des Proxies. Nach Aufruf von Methoden auf dem Proxy leitet dieser die Anfragen an das Backend weiter und erhält von dort die Antwort, die er an den Client weiterleitet.

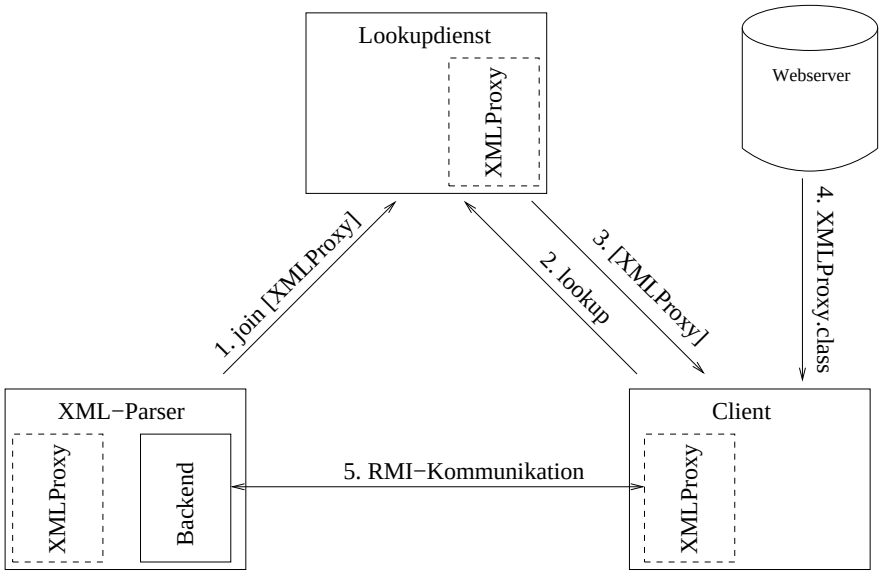

Abbildung 3.1: Jini Beispiel

Im Jini-Paket werden die oben bereits angesprochenen Dienste (Lookupdienst, Mietverlängerungsdienst) mitgeliefert. Weiters werden ein Dienst für die Zwischenspeicherung von Remote-Events und ein Dienst, an den die Suche nach

Diensten delegiert werden kann, mitgeliefert. Zwei weitere Dienste betreffen die Unterstützung von verteilten Transaktionen sowie die tupelbasierte Speicherung von Daten.

Für die Verwaltung von verteilten Transaktionen über das von Gray in [Gra78] beschriebene Two-Phase-Commit-Protokoll (2PC) wird ein Transaktions-Manager bereitgestellt. Alle Dienste und deren Operationen, die an dieser verteilten Transaktion teilnehmen möchten, müssen die Schnittstelle `TransactionParticipant` implementieren. Der Manager verwaltet dann die Einzeltransaktionen und ruft die entsprechenden Methoden (`prepare`, `commit`, `abort`) auf. Die gesamte Transaktion wird mit einer Mietdauer versehen, um das Blockierungs-Problem zu vermeiden.

Ein Dienst stellt eine JavaSpaces Realisierung bereit. JavaSpaces lehnen sich an das Tupel-Konzept der Programmiersprache Linda [CG90] an. Für eine nähere Erläuterung von JavaSpaces siehe [FHA99, Sun99]. Es gibt eine transiente Variante dieses Dienstes. Beim Beenden des Dienstes gehen alle enthaltenen Daten verloren. Alternativ gibt es eine Variante, welche die Daten persistent in einer objektorientierten Datenbank[3] speichert, so dass auch nach einem Absturz und Neustart noch alle Daten vorhanden sind.

Des Weiteren werden einige Zusatzwerkzeuge mitgeliefert. Diese Werkzeuge beinhalten zum Beispiel einen Browser, um sich die an Lookupdiensten angemeldeten Jini-Dienste anzeigen zu lassen und zu administrieren. Abbildung 3.2 zeigt den Browser, der über Attributbedingungen ausgewählte Dienste anzeigt.

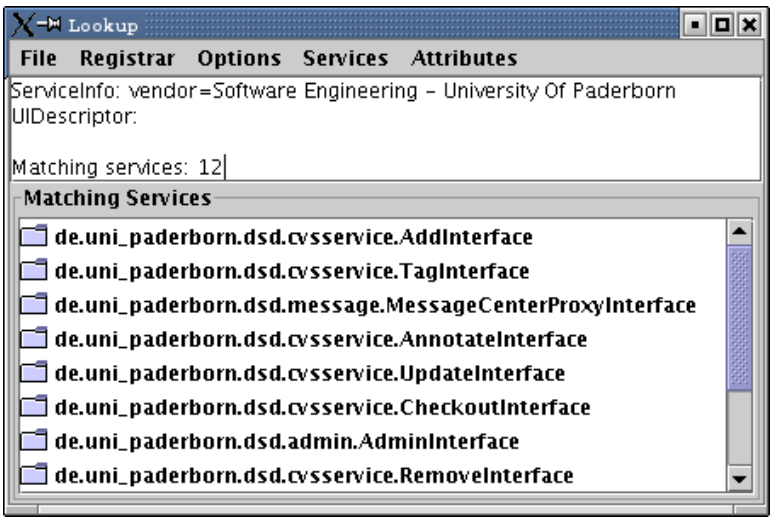

Abbildung 3.2: Der Jini-Browser

Der benötigte Webserver stellt einen kritischen Punkt hinsichtlich der Verfügbarkeit des Systems dar, da sein Ausfall die Nutzung des Dienstes

[3]Der persistente JavaSpace-Dienst nutzt die objektorientierte Datenbank PSE Pro. http://www.exln.com/products/psepro/

unmöglich macht. Das Jini-Paket enthält einen kleinen Web-Server, der innerhalb des Dienstprozesses läuft; der Dienst ist dadurch unabhängig von einem externen Webserver. Um existierende Webserver fehlertolerant zu machen, können die in [DKMT96] und [AT01] vorgeschlagenen Lösungen genutzt werden.

Die Jini-Architektur bietet eine sehr flexible Möglichkeit, um verteilte Systeme zu entwickeln. Besonders die Fähigkeit mit einem Smartproxy nicht auf eine normale Client-Server-Kommunikation festgelegt zu sein, um dynamisch auf Fehler zu reagieren, und das Konzept der Mietdauern ermöglicht die Entwicklung fehlertoleranter Systeme. Des Weiteren ist das Konzept nicht serverbasiert, sondern ermöglicht durch die Nutzung von Multicast-Nachrichten eine skalierbare Lösung. Durch die Trennung von Schnittstelle, Proxy und Backend wird der Komponentenbegriff aus den Abschnitten 1.1 und 2.1 unterstützt und dadurch die Wartbarkeit des Systems erhöht. Aus diesen Gründen wird die Jini-Technologie für die Lösung der Aufgabenstellung dieser Arbeit genutzt.

Es gibt allerdings in der Jini-Technologie keine Unterstützung für die Verwaltung eines kompletten Systems bestehend aus mehreren Diensten. Im Bezug auf Fehlertoleranz fehlt Jini die Fähigkeit, ausgefallene Dienste neu zu starten, so dass das System funktionsfähig bleibt. Außerdem muss während des Betriebes jeder Dienst sich selber an den Lookupdiensten anmelden sowie die von ihm zu nutzenden Dienste finden.

Im Folgenden werden zwei Ansätze angesprochen, die eine teilweise Realisierung der, in Abschnitt 1.3 angesprochenen, gewünschten Unterstützung für die Entwicklung und den Betrieb von Komponenten bieten. Beide Lösungen setzen auf der in diesem Abschnitt dargestellten Jini-Technologie auf. Im ersten Abschnitt wird eine Unterstützung für die Implementierung von Jini-Diensten in einer integrierten Entwicklungumgebung vorgestellt. Der zweite Abschnitt enthält eine Beschreibung einer Laufzeitumgebung für Jini-Dienste.

3.3.1 NetBeans

Für die integrierte Entwicklungsumgebung NetBeans [Net02] der Firma Sun gibt es ein Plugin, das eine rudimentäre Unterstützung für die Entwicklung von Jini-Diensten bietet. Dieses Plugin erweitert NetBeans um zwei Fähigkeiten:

- Man kann textuelle Vorlagen für die Entwicklung von Jini-Diensten nutzen. Mittels dieser Vorlagen kann zum Beispiel ein Dienst erstellt werden, der einen Proxy und ein RMI-Backend enthält. Der Proxy delegiert einfach alle Anfragen an das RMI-Backend.

- Es besteht die Möglichkeit, ähnlich dem in Abschnitt 3.3 angesprochenen Browser, die in der Jini-Umgebung vorhandenen Dienste anzuzeigen. Die Abbildung 3.3 zeigt diesen Browser.

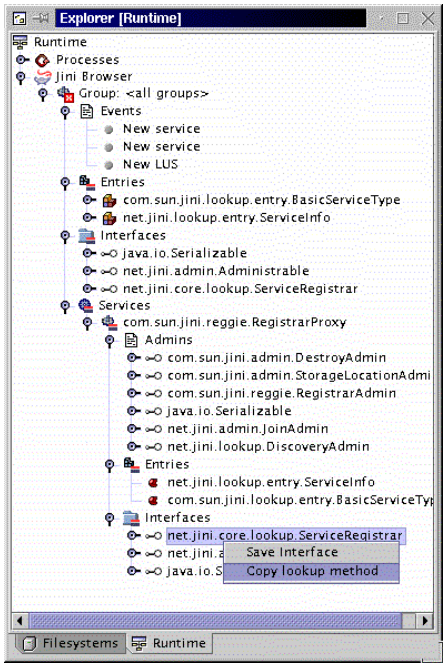

Abbildung 3.3: NetBeans-Jini-Browser

Die Unterstützung für die Entwicklung von Jini-Diensten in NetBeans ist für die Zwecke dieser Arbeit nicht ausreichend. Eine graphische Entwicklung oder eine Unterstützung durch UML-Diagramme ist nicht vorhanden. Es fehlt eine Generierungsunterstützung für die Anmeldung der Dienste an den Lookupdiensten und die Fähigkeit, automatisch mehrere Dienste miteinander zu verbinden und dadurch zu einer größeren Applikation zusammenzusetzen.

3.3.2 RIO

Das RIO-Projekt[4] [Sun01b] des Sun Pervasive Java Center for implementation assistance setzt sich zum Ziel, die Entwicklung von Jini-Diensten und deren Betrieb in einem lokalen Netz einfach und fehlertolerant zu gestalten. Abbildung 3.4 zeigt die einzelnen Teile des Gesamtsystems. Die wichtigsten Teile des Systems werden im Folgenden vorgestellt:

Jini Service Beans sind das grundlegende Konzept von RIO. Ein Jini Service Bean besteht aus einer Menge von Klassen, welche die Funktionalität des Dienstes bereitstellen, aber auch bestimmte Schnittstellen für den Betrieb des Jini-Dienstes implementieren. Ein Dynamic Container, in RIO *CyberNode* genannt, wird auf jedem Rechenknoten gestartet. Dieser CyberNode bietet die Möglichkeit, einzelne Service Beans auf dem Rechenknoten zu instanziieren. Als *Dynamic Provisioning* wird eine Technik bezeichnet, die mittels einer XML-Spezifikation („Operational

[4] http://developer.jini.org/exchange/projects/rio/index.html

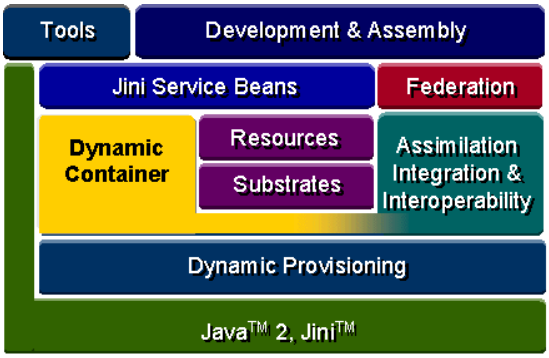

Abbildung 3.4: Übersicht des RIO-Projektes

String“ genannt) die dort eingetragenen Service Beans auf verschiedene CyberNodes verteilt. Diese Verteilung wird durch einen Monitor durchgeführt, der die XML-Spezifikation enthält. Dieser Monitor sucht einen bestimmten CyberNode anhand von bestimmten *Quality of Service* Bedingungen aus, startet den Service Bean auf diesem Rechenknoten und überprüft periodisch, ob der Bean noch funktioniert oder eventuell abgestürzt ist. Im letzteren Fall startet er den Bean auf einem anderen CyberNode neu.

RIO bietet Möglichkeiten, verschiedene, durch den Entwickler definierte, Daten zur Messung zu sammeln und anzuzeigen. Durch diese *Watches*, die Teil der *Ressourcen* sind, besteht die Möglichkeit, zum Beispiel Auslastungsdaten der verteilten Applikation zu sammeln und in einer Darstellungsapplikation anzuzeigen. Im ,,Federation“-Teil des Projektes wird die Möglichkeit geboten, die von den Jini-Diensten versandten Multicast-Nachrichten, die normalerweise nicht über Netzwerkgrenzen hinweg exportiert werden, in weiter entfernte Netzwerke zu senden.

Des Weiteren werden einige Werkzeuge für die Entwicklung und den Betrieb der Service Beans mitgeliefert. Für die Erstellung der Java-Archive werden eine erweiterte Version des BuildTool-Werkzeuges (ein Teil des *OutOfTheBox*-Projektes [Pro02b]) und einige Erweiterungen für das Werkzeug ANT [Pro02c] angeboten. Um die aktuelle Situation des Systems zu visualisieren, kann der ,,Operational String Monitor“ genutzt werden. Der ,,Viewer“ stellt die im System gefundenen Jini-Dienste dar und ermöglicht die Nutzung der graphischen Benutzerschnittstellen der Dienste.

Die oben dargestellten Programme und Werkzeuge sind zum Teil auch ohne das restliche System nutzbar. So können ,,Federation“ und ,,Viewer“ problemlos auch in normalen Jini-Netzen genutzt werden.

Prinzipiell ist das Ziel von RIO ähnlich zu dem Ziel dieser Diplomarbeit, Fehlertoleranz für Jini-Umgebungen zu erreichen und die Entwicklung der Dienste einfacher zu gestalten. Die Fehlertoleranz ist in RIO allerdings nicht so ausgeprägt. Zwar wird bei einem Absturz eines Service Bean dieses durch den Monitor

neu gestartet, allerdings ist durch die Einheit von Monitor und Verteilungsbeschreibung („Operational String“) Fehlertoleranz bei Ausfall des Monitors nicht gewährleistet, da mit dem Absturz des Monitors auch die Information über die zu verteilenden Beans verloren geht. Es kann kein anderer Monitor die nötigen Informationen erhalten und die Aufgaben übernehmen.

Die im Ziel dieser Arbeit (Abschnitt 1.3) angesprochenen Möglichkeiten der Komponentenintegration durch die automatische Verknüpfung von angebotenen und benutzten Schnittstellen mit Berücksichtigung von verschiedenen Restriktionen werden in RIO überhaupt nicht angeboten. Hierfür ist jeder Dienst selbst zuständig.

Ein zweiter grundlegender Unterschied ist das Ziel dieser Arbeit, eine durchgehende graphische Unterstützung mittels der UML sowohl in Entwurf und Implementierung von Komponenten als auch im Betrieb und dessen Planung zu bieten. Für den Entwurf und die Implementierung der Dienste stellt RIO nur Bibliotheken, aber keine durchgängige Unterstützung zur Verfügung. Auch während des Betriebes oder dessen Planung werden von RIO keine UML-Diagramme genutzt. Die oben angesprochenen „Viewer“ und „Operational String Monitor“ zeigen das System in einer eigenen Visualisierung an.

Eine Behebung der Schwächen und eine Erweiterung von RIO würde sich anbieten, es ist aber zu RIO kein Quelltext der Infrastrukturdienste verfügbar, so dass eine Erweiterung nicht möglich ist. Des Weiteren war zu Beginn dieser Diplomarbeit nicht sicher, ob an einer Weiterentwicklung von RIO gearbeitet wird. Bis zur Abgabe dieser Arbeit wurden drei weitere Versionen von RIO freigegeben, allerdings auch ohne Quelltext und ohne Änderung der Dokumentation. Die oben angesprochenen Probleme gelten für RIO also auch weiterhin.

3.4 Fazit

Die anfangs vorgestellten Konzepte von Web-Services und Enterprise JavaBeans bieten nicht die benötigte Skalierbarkeit oder Fehlertoleranz, da sie auf einem rein serverbasierten Konzept aufsetzen. Die Jini-Technologie ermöglicht durch das Konzept der Mietdauern und der dezentralen Struktur, fehlertolerante, skalierbare und wartbare Systeme zu entwerfen.

Nachfolgend wurden zwei Konzepte für die Entwicklung und den Betrieb von Jini-Diensten erläutert, die Unterstützung für die Entwicklung von Jini-Diensten in der NetBeans-Entwicklungsumgebung ist nur äußerst rudimentär. RIO erhöht die Fehlertoleranz einer Jini-Umgebung durch die Überwachung der Verfügbarkeit der Jini-Service-Beans. Des Weiteren bietet RIO einige zusätzliche Werkzeuge, wie zum Beispiel den Versand von Multicast-Nachrichten über Netzwerkgrenzen hinweg, die für den Betrieb der Jini-Umgebung nützlich sind. Problematisch ist die fehlende Fehlertoleranz bei Ausfall eines Monitors. Da auch der Quelltext von RIO nicht verfügbar ist, lassen sich die Probleme nicht durch Anpassungen

von RIO beheben. Aus diesen Gründen wird in dieser Arbeit in Kapitel 5 eine neue Laufzeitumgebung für Jini-Dienste entworfen, die explizit die oben angesprochenen Probleme löst. Nach einer Freigabe der Quelltexte von RIO ist eine Integration der Konzepte dieser Arbeit in RIO denkbar.

4 Entwurf, Implementierung und Planung des Betriebs von Komponenten

Abbildung 4.1 zeigt die Einordnung dieses Kapitels in den in Abbildung 1.1 dargestellten Gesamtablauf. Im ersten Teil dieses Kapitels wird die Unterstützung für die Entwicklung eines Dienstes, Entwurf und Implementierung, dargelegt, im zweiten Teil die Planung des Betriebs. Die Erläuterungen werden durch Beispiele unterstützt. In diesen Beispielen wird der Checkin-Dienst aus dem Beispielszenario in Abschnitt 1.5 zuerst entworfen, später in ein Komponentendiagramm eingebettet und als letztes eine Planung des Betriebs des Checkin-Dienstes durchgeführt.

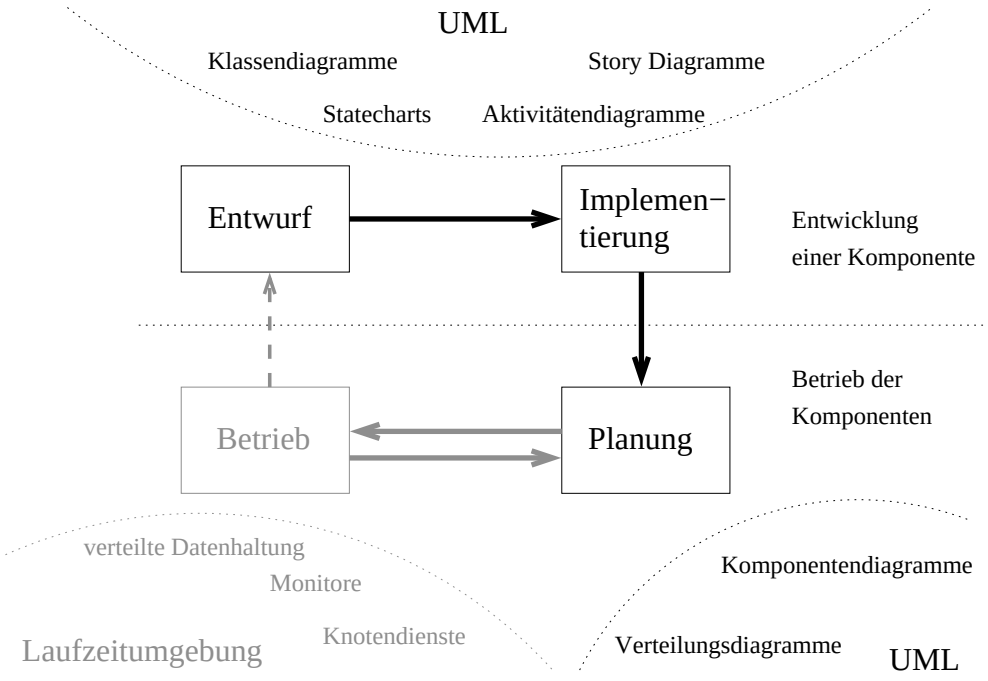

Abbildung 4.1: Einordnung in das Gesamtsystem

Im Bereich des Entwurfs und der Implementierung eines einzelnen Dienstes

kann durch die Nutzung von UML-Diagrammen die Wartbarkeit eines Softwaresystems erhöht werden, da die Struktur und das Verhalten der Softwaresysteme auf einer höheren Abstraktionsebene spezifiziert werden. Diese höhere Abstraktionsebene erlaubt dem Entwickler, die Software einfacher zu überblicken und die Zusammenhänge zwischen den einzelnen Teilen zu verstehen. Durch die Möglichkeit aus den verschiedenen Struktur- und Verhaltensdiagrammen direkt Java-Quelltext zu generieren, kann bei fehlerfreier Generierung die Korrektheit der Implementierung in Hinsicht auf die Spezifikation garantiert werden.

Im Bereich der Planung des Betriebs der Dienste werden Diagramme genutzt, um zum einen verschiedene Dienste zu einem größeren Softwaresystem zusammenzusetzen und zum anderen die Verteilung der einzelnen Dienste auf die einzelnen Rechenknoten zu spezifizieren. Durch die genaue Spezifikation der Planung ist die Wartbarkeit des Gesamtsystems für einen Administrator höher, da in den Diagrammen genau festgelegt wird, welche Dienste miteinander kommunizieren und wo diese Dienste ausgeführt werden können. Durch die Spezifikation der Dienstverteilung auf die Rechenknoten wird Einfluss auf die Skalierbarkeit des Gesamtsystems genommen, da durch eine restriktivere Spezifikation der Verteilung Einfluss auf die Güte der erreichbaren Lastverteilung genommen wird.

Im nächsten Abschnitt wird zuerst auf die für diese Arbeit relevanten Teile der Unified Modelling Language eingegangen. In Abschnitt 4.2 wird dann die Unterstützung von Entwurf und Implementierung der einzelnen Dienste erläutert. In Abschnitt 4.3 werden die für die Komposition der Dienste genutzten Komponentendiagramme beschrieben. In Abschnitt 4.4 werden die Verteilungsdiagramme für die Planung des Betriebs des verteilten Systems dargestellt.

4.1 Die Unified Modelling Language (UML)

Zur Modellierung und Dokumentation von Programmen gibt es seit langem graphische Notationen im Bereich der Software-Entwicklung. Für die Notation von imperativen bzw. prozeduralen Programmiersprachen werden zum Beispiel Nassi-Shneiderman-Diagramme [NS73] für die Darstellung des Kontrollflusses genutzt. Für die Darstellung der Tabellen und Relationen in relationalen Datenbanken sind Entity-Relationship-Diagramme (ER) [Che76] weit verbreitet. Zur formalen Darstellung von komplexen nebenläufigen Systemen werden Petri-Netze [Pet62] genutzt.

Seit dem Aufkommen von objektorientierter Analyse und Entwurf wurden verschiedene Notationen für die Modellierung der Programme erarbeitet. Unter diesen Ansätzen sind besonders die Ansätze von Booch [Boo93], Rumbaugh [RBP+91] und Jacobson [JCJO96] hervorzuheben. Da jede dieser Methoden eigene Stärken hat, wurden diese drei Ansätze zur Unified Modelling Language (UML) vereinigt und weiterentwickelt. Diese graphische Sprache beinhaltet viele verschiedene Diagrammtypen zur Modellierung von Softwaresystemen.

[OMG01] beinhaltet die der Arbeit zugrunde gelegte Spezifikation 1.4 der UML. Diese Spezifikation wird von der Object Management Group (OMG) erarbeitet und herausgegeben.

Die UML enthält Diagrammarten für den kompletten Lebenszyklus einer Software. Für die Anforderungsdefinition werden Use-Case-Diagramme angeboten. Für den Entwurf und die Implementierung des Systems stehen für die Spezifikation der Struktur Klassen- und Objektdiagramme und für die Spezifikation des Verhaltens Aktivitäts-, Zustands-, Sequenz- und Kollaborationsdiagramme zur Verfügung. Für die Planung des Betriebs einer komponentenbasierten Applikation können Komponenten- und Verteilungsdiagramme genutzt werden. Verteilungsdiagramme werden auch zur Visualisierung der konkreten Situation während des Betriebs genutzt.

Komponenten- und Verteilungsdiagramme bieten in der aktuellen Spezifikation nicht die für diese Arbeit nötigen Ausdrucksmittel. In [AW99] wurden Defizite bei Verteilungsdiagrammen festgestellt, die bereits teilweise in der Spezifikation 1.3 der UML verbessert wurden. Die für diese Arbeit notwendigen Erweiterungen werden für Komponentendiagramme in Abschnitt 4.3.1 und für Verteilungsdiagramme in Abschnitt 4.4.1 dargestellt.

Das UML-Case Tool Fujaba wird in dieser Arbeit für die Erstellung der UML-Diagramme genutzt. Klassen-, Zustands- und Aktivitätendiagramme werden von Fujaba bereits unterstützt. Diese Diagrammarten werden unverändert genutzt. Für diese Diagrammarten besteht die Möglichkeit, mittels Forward-Engineering Java-Quelltext zu generieren. Die Konzepte für diese Quelltextgenerierung werden für Klassen- und Aktivitätsdiagramme in [FNT98, S. 137-214] und für Zustandsdiagramme in [Köh99, S.25ff] und [KNNZ99] dargelegt. Die Unterstützung für Komponenten- und Verteilungsdiagramme wurde im Rahmen dieser Arbeit als Plugin für Fujaba neu implementiert.

4.2 Entwurf und Implementierung

Für die Entwicklung einer Komponente müssen die folgenden Schritte durchgeführt werden: Zuerst wird die Komponente mit UML-Diagrammen entworfen und implementiert. Danach müssen einige zusätzliche Informationen über diese Komponente für den Betrieb innerhalb der Laufzeitumgebung angegeben werden. Als letztes wird aus den Diagrammen Java-Quelltext generiert. Nachdem dieser kompiliert und gepackt wurde, wird eine XML-Datei mit den zusätzlichen Informationen erzeugt.

Für die Entwicklung der Komponente werden die folgenden Diagrammarten zur Modellierung der Struktur und des Verhaltens eines Programms genutzt:

Klassendiagramme: Klassendiagramme ähneln ER-Diagrammen und modellieren die Struktur des Softwaresystems durch Klassen und deren Beziehungen untereinander.

Aktivitätendiagramme: Dieser Diagrammtyp erlaubt die Modellierung der Kontrollflüsse von Methoden. Hiermit wird das Verhalten in einem bestimmten Zustand beschrieben.

Zustandsdiagramme: Zustandsdiagramme basieren auf der von Harel in [Har87] entwickelten Statechart-Notation. Zustandsdiagramme spezifizieren das Verhalten reaktiver Objekte, indem Zustände des Objektes und die Regeln für die Übergänge zwischen diesen Zuständen spezifiziert werden.

Mittels dieser Diagrammarten kann ein Dienst vollständig graphisch modelliert werden. So sind die in Kapitel 5 vorgestellten Laufzeitumgebungsdienste komplett in diesen Diagrammarten spezifiziert worden. Durch die automatische Quelltextgenerierung des Fujaba-Werkzeugs wird die Implementierung direkt aus der Spezifikation erzeugt.

Mit Klassendiagrammen wird die Struktur des Dienstes modelliert. Dies betrifft im Besonderen die Schnittstellen, die der Dienst anbieten soll, sowie die Schnittstellen, die der Dienst nutzt. Des Weiteren enthält das Strukturmodell Klassen für die Implementierung der Dienstlogik sowie, bei Nutzung von RMI, für die Kommunikation zwischen Proxy und Backend.

Die Schnittstellen müssen durch Stereotypen gekennzeichnet sein. Dies betrifft die von anderen Jini-Diensten angebotenen Schnittstellen (Stereotyp: `UsedServiceInterface`) sowie die von der Komponente angebotenen Schnittstellen (Stereotyp: `ProvidedServiceInterface`). Diese Informationen werden später bei der Generierung der Daten für die Laufzeitumgebung benötigt.

Abbildung 4.2 zeigt das Klassendiagramm des Checkin-Dienstes. Die Checkin-Schnittstelle wird angeboten und die Datenbank- und XML-Parser-Schnittstellen werden benötigt. Der Dienst kommuniziert direkt mit dem CVS-Serverprozess. Diese Kommunikation wird in der Superklasse `CVSAction` des Proxies durchgeführt.

Um einen Dienst nicht immer von Grund auf neu modellieren zu müssen, werden im Folgenden zwei direkt funktionsfähige Vorlagen vorgestellt, die zwei typische Jini-Diensttypen darstellen. Aufbauend auf diesen Vorlagen kann der Entwickler dann die Applikationslogik mit Hilfe geeigneter Diagramme erstellen. Diese Vorlagen sind in Form einer Fujaba-Projektdatei vorhanden, so dass der Entwickler diese Vorlagen direkt verwenden und anpassen kann.

Vorlage für einen reinen Proxy-Dienst

Diese Vorlage enthält den Entwurf und die Spezifikation eines Dienstes, dessen Proxy alle Aufgaben bereits auf dem Client erledigt. Abbildung 4.3 zeigt das Klassendiagramm des Dienstes. Die Klasse `Proxy` realisiert die Funktionalität des Dienstes. Sie erbt von der Schnittstelle `ProxyInterface`, nach der in den Lookupdiensten gesucht werden kann, und von der Schnittstelle `Serializable`, um die Serialisierbarkeit der Klasse zu ermöglichen. Eine Implementierung der

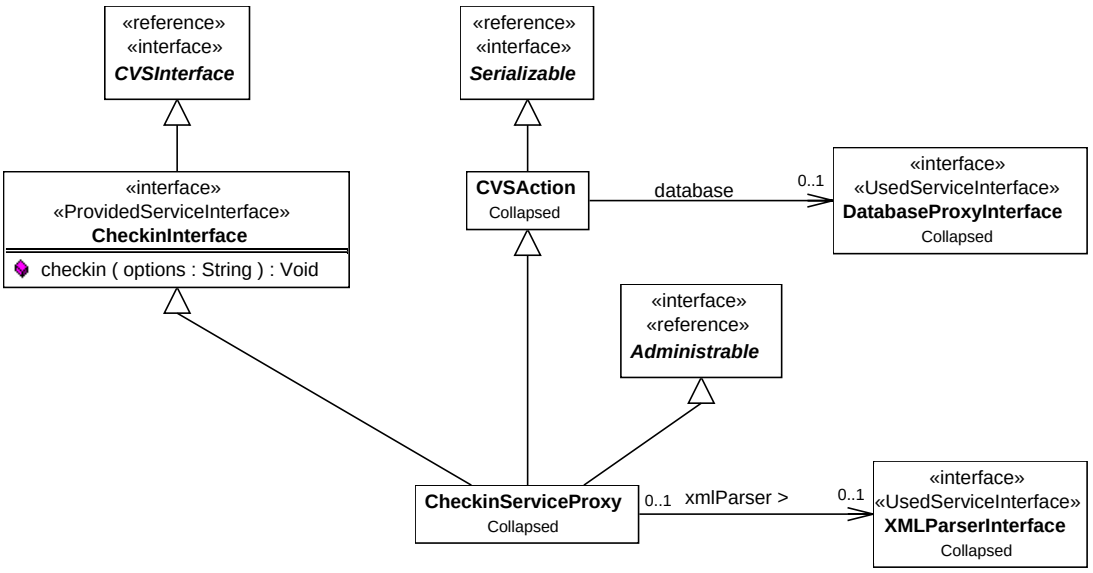

Abbildung 4.2: Klassendiagramm des Checkin-Dienstes

`FrameWorkInterface`-Schnittstelle wird von der im nächsten Kapitel beschriebenen Laufzeitumgebung benötigt, um den Dienst im System zu starten.

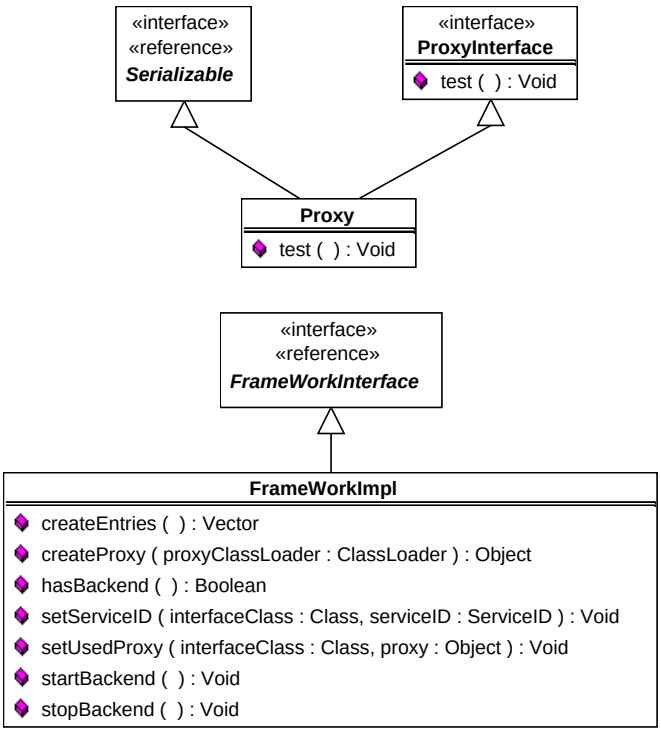

Abbildung 4.3: Klassendiagramm der Vorlage für einen reinen Proxy-Dienst

Vorlage für einen Dienst mit RMI-Backend

Diese Vorlage enthält die Spezifikation für einen Dienst, dessen Proxy über RMI [Sun98a] mit einem Backend kommuniziert. Die einzige Aufgabe des Proxies ist, die Anfragen an das Backend weiterzureichen und die Antwort an das anfragende Programm zurückzureichen. Abbildung 4.4 zeigt das Klassendiagramm des Dienstes mit einem RMI-Backend. Der Dienst besteht in diesem Klassendiagramm aus zwei Teilen. Links ist die `RMIProxy`-Klasse dargestellt. Diese Klasse erbt analog zur `Proxy`-Klasse in der obigen Abbildung von `Serializable` und einer Schnittstellenklasse. Neu in diesem Klassendiagramm ist die Schnittstelle `RMIBackendInterface`. Diese Schnittstelle erbt von der Schnittstelle `Remote`, die als Markierung für Schnittstellen dient, von denen Klassen, die über RMI angesprochen werden sollen, erben. Die Methode `test()` in der Schnittstelle `RMIBackendInterface` wird hierdurch als *remote* markiert. Die Klasse `RMIBackend` erbt von `RMIBackendInterface` und stellt die Implementierung der Methode `test()` bereit. Die Klasse `RMIProxy` hat eine unidirektionale Assoziation zu `RMIBackendInterface`. Mittels dieser Assoziation kann der Proxy den Methodenaufruf zum Backend delegieren. Die Klasse für die Nutzung in der Laufzeitumgebung wurde in dieser Abbildung weggelassen, da die Deklaration der Klasse identisch zur Deklaration im Klassendiagramm des reinen Proxy-Dienstes ist.

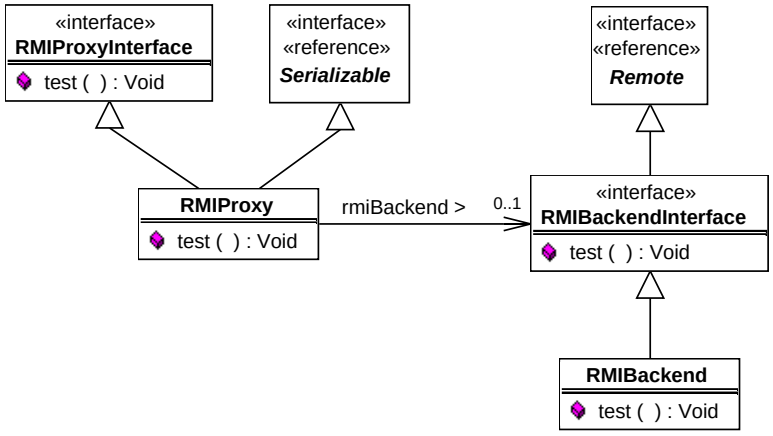

Abbildung 4.4: Klassendiagramm der Vorlage für einen Dienst mit RMI Backend

4.2.1 Kompilierung und Packen der Komponente

Um die Komponente nutzen zu können, müssen, wie oben angesprochen, einige Operationen durchgeführt werden. Zuerst muss der Java-Quelltext generiert werden. Dies geschieht mit Hilfe der Quelltextgenerierung von Fujaba. Danach wird der Quelltext mit Hilfe des BuildTool-Werkzeugs kompiliert, wenn nötig für RMI-Klassen Stubs erzeugt und schlussendlich ein oder mehrere Java-Archive

erzeugt. Die Klassen für einen Jini-Dienst werden typischerweise auf zwei Java-Archive aufgeteilt. Ein Java-Archiv enthält die Klassen, die später auf dem Client genutzt werden sollen, ein zweites Archiv enthält die Klassen, die auf dem Server ausgeführt werden. Als letzter Schritt wird die XML-Beschreibung eines Dienstes erzeugt, damit dieser in einem Komponenten- und Verteilungsdiagramm (siehe Abschnitte 4.3 und 4.4) genutzt werden kann.

Das Werkzeug BuildTool ist Teil des OutOfTheBox-Projekts [Pro02b]. Ziel dieses Projektes ist es, die Entwicklung von Jini-Diensten einfacher zu machen. Das Werkzeug BuildTool dient dazu, den Quelltext der Dienste zu kompilieren, den RMI-Compiler zu starten und in einem letzten Schritt die Java-Archive für den Betrieb des Dienstes zu erstellen. Für den letzten Schritt kann das ClassDep-Werkzeug aus dem Jini-Paket genutzt werden. Dieses Werkzeug analysiert die kompilierten Klassen und ihre Abhängigkeiten und erstellt so, ausgehend von bestimmten Wurzelklassen, einen Baum der Abhängigkeiten zwischen den Klassen. Mit diesem Werkzeug können einfach die Klassen identifiziert werden, welche die verschiedenen Java-Archive bilden.

Die für den Einsatz des BuildTools nötigen Dateien werden mit Hilfe der Diagramminformationen erzeugt. Das BuildTool benötigt drei Konfigurationsdateien. In der ersten Datei werden die zur Erledigung der Aufgaben nötigen Pfade und Klassenpfade, die Dateinamen des Java- und RMI-Compilers und deren zusätzliche Optionen sowie die Dateinamen für die restlichen Konfigurationsdateien angegeben. In einer zweiten Konfigurationsdatei werden die Dateien aufgelistet, die vom RMI-Compiler zur Generierung der benötigten Stubs und Skeletons nachbearbeitet werden sollen. Diese Klassen charakterisieren sich dadurch, dass sie von einer Schnittstelle erben, die von `java.lang.Remote` erbt. Die dritte Konfigurationsdatei enthält die Spezifikation für die Erstellung der Java-Archive.

Diese drei Dateien werden mit Hilfe der Informationen aus den UML-Diagrammen erzeugt. Die Datei mit den Klassen, die durch den RMI-Compiler bearbeitet werden müssen, wird bei jedem Aufruf der Generierungsfunktion erstellt, da die Informationen komplett aus den Klassendiagrammen extrahiert werden können. Die beiden anderen Konfigurationsdateien können durch Aufruf der entsprechenden Funktion auch generiert werden, allerdings ist es möglich, dass diese Dateien eine Nachbearbeitung durch den Benutzer benötigen und vom Benutzer um zusätzliche Informationen erweitert werden. Aus diesem Grund stellt die Generierung nur einen Vorschlag des Systems und eine Arbeitserleichterung für den Benutzer dar.

4.3 Komponentendiagramme

Komponentendiagramme beschreiben die Konfiguration der Komponenten. Diese Konfiguration beinhaltet die Verbindung der Komponenten zu einer zusam-

mengesetzten Komponente sowie die Integration der Komponenten über ihre Schnittstellen. Es werden die einzelnen Komponenten, ihre Schnittstellen sowie die Abhängigkeiten zwischen diesen Schnittstellen definiert. Es werden angebotene und benutzte Schnittstellen unterschieden. Ein Komponentendiagramm besteht aus einem Graph, dessen Knoten Schnittstellen und Komponenten und dessen Kanten die Linien von Komponenten zu angebotenen oder benutzten Schnittstellen sind.

Abbildung 4.5 zeigt das Aussehen einer Komponente. Der Rahmen mit den beiden kleinen Rechtecken enthält den Namen der Komponente. Die Komponente stellt die Checkinkomponente des Beispielszenarios dar. Schnittstellen werden als nicht ausgefüllter Kreis mit dem nebenliegenden Namen gezeichnet. Angebotene Schnittstellen werden durch eine Kante an die dazugehörige Komponente dargestellt. Eine angebotene Schnittstelle ist in diesem Beispiel das `CheckinInterface`; über diese Schnittstelle wird die eigentliche Funktionalität der Komponente angeboten. Die Administrable-Schnittstelle bietet die Möglichkeit, die Komponente zu konfigurieren. Die benutzten Schnittstellen werden durch einen gestrichelten Pfeil von der nutzenden Komponente zur Schnittstelle gezeichnet. Im Beispiel wird die bereits angesprochene XML-Parser-Komponente genutzt, um bestimmte generierte Daten einzulesen. Diese Daten werden mittels der Datenbankkomponente in eine Datenbank geschrieben.

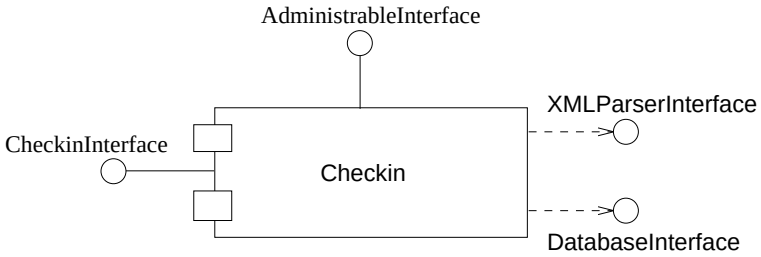

Abbildung 4.5: Aussehen einer Komponente

Komponenten können andere Komponenten beinhalten, dadurch wird eine Komposition dargestellt. Die angebotenen Schnittstellen der untergeordneten Komponenten sind dadurch gekapselt, allerdings besteht die Möglichkeit, die Schnittstellen auch außerhalb der Kompositionskomponente anzubieten. Um dies zu erreichen, muss ausgehend von der inneren Schnittstelle eine weitere Kante mit einem Schnittstellenkreis außerhalb der Komponente eingezeichnet werden. Abbildung 4.6 zeigt das Aussehen einer aus mehreren Komponenten zusammengesetzten Komponente.

4.3.1 Einschränkungen und Erweiterungen

Die in dieser Arbeit genutzten Komponentendiagramme unterstützen nur Komponenten und Schnittstellen. Die in der UML-Spezifikation vorgesehen Artefakte,

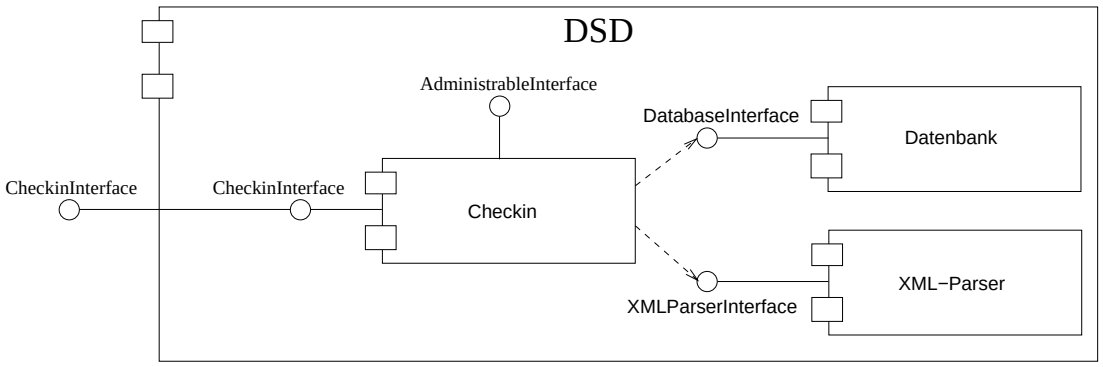

Abbildung 4.6: Aussehen einer zusammengesetzten Komponente

durch die Komponenten implementiert werden, werden nicht unterstützt. Im Weiteren wird zwischen zwei verschiedenen Typen von Komponenten unterschieden: Kompositionskomponenten und Dienste. Ein Dienst ist eine Komponente, die eine bestimmte Funktionalität über Schnittstellen anbietet und andere Dienste bzw. deren Schnittstellen nutzt. Eine Kompositionskomponente ist eine Komponente, die mehrere Komponenten beinhaltet. Die Schnittstellen der beinhalteten Komponenten können dann in der Komponente über Benutzungskanten verbunden werden. Diese Kompositionskomponenten sind beliebig schachtelbar.

Jeder Dienst enthält die folgenden Informationen: Name des Dienstes, eine Beschreibung, eine URL, die auf die für das Starten des Dienstes benötigten Dateien verweist, die Namen der Java-Archive von Backend und Proxy sowie der Name einer Klasse, die eine bestimmte Schnittstelle (siehe Abschnitt 5.3 und Abbildung 5.10) implementiert, welche von der Laufzeitumgebung genutzt wird, um den Dienst zu instanziieren.

Die benötigten Schnittstellen können durch Bedingungen näher beschrieben werden. Bedingungen bestehen aus einem Schlüssel, einem Wert[1] und einem Operator (*gleich*, *ungleich*, *kleiner)*. Diese Bedingungen müssen gelten, damit eine gefundene Schnittstelle als passend erkannt wird. Diese Bedingungen können bei den angebotenen Schnittstellen im Diagramm gesetzt werden. Hier wird dasselbe Schlüssel-Wert-Paar gesetzt und mit einem Zuweisungsoperator verknüpft. Beispiel für solche Bedingungen sind der Name eines Datenbank-Servers oder die Rechengenauigkeit einer Schnittstelle für mathematische Berechnungen.

Zwischen angebotenen und benutzten Schnittstellen kann eine Abhängigkeitsbeziehung existieren. Dies bedeutet, dass Operationen auf einer angebotenen Schnittstelle Operationen auf einer benutzten Schnittstelle auslösen. Diese Abhängigkeiten können genutzt werden, um zyklische Abhängigkeiten, die zum Beispiel zu Endlosschleifen führen können, in den Aufrufen von Komponenten zu erkennen und zu verhindern. Abbildung 4.7 zeigt eine Komponente mit zwei Abhängigkeitsbeziehungen.

[1] Schlüssel und Wert sind als linkes und rechtes Attribut modelliert.

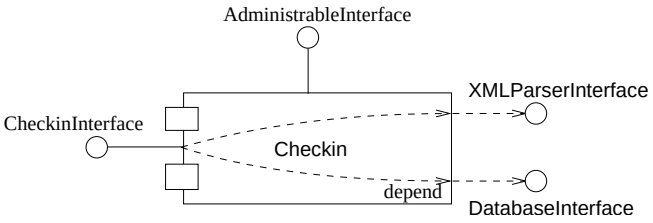

Abbildung 4.7: Beispiel für eine Komponente mit Abhängigkeiten

4.3.2 Meta-Modell

Im Folgenden wird das Meta-Modell für Komponentendiagramme vorgestellt. Abbildung 4.8 zeigt das Klassendiagramm, wie es im Fujaba-Werkzeug spezifiziert wurde. Die Java-Bean Stereotypen ändern die Quelltextgenerierung, damit das durch die Instanzen der Klassen des Meta-Modells erzeugte Komponentendiagramm am Bildschirm visualisiert werden kann. Für das Verständnis des Meta-Modells sind diese Stereotypen unwichtig.

`AbstractComponent` ist die abstrakte Basisklasse für Komponenten. Mit der Klasse `XMLComponent` bildet sie ein Composite-Pattern [GHJV94, S.163-174]. Mittels dieser Klasse können verschiedene Komponenten ineinander verschachtelt werden. Instanzen der Klasse `Service` stellen einen einzelnen Jini-Dienst im System dar. Ein Dienst besitzt mehrere angebotene Schnittstellen (Klasse `ProvidedInterface`) und mehrere benutzte Schnittstellen (Klasse `UsedInterface`). Jede dieser Schnittstellen existiert innerhalb einer bestimmten Kompositionskomponente. Dies bedeutet, dass nur Komponenten innerhalb dieser Kompositionskomponente auf diese Schnittstelle zugreifen können. Jede angebotene Schnittstelle besitzt mehrere `AttrExprPair`-Instanzen. Diese Instanzen enthalten die oben beschriebenen zusätzlichen Schlüssel-Werte-Paare.

Abbildung 4.9 beinhaltet das Klassendiagramm für die Restriktionen der Komponentenintegration. Es gibt zwei verschiedene Restriktionen, abgeleitet von der abstrakten Klasse `ConnectionRestrictions`, für die Verbindung von Komponenten. Instanzen der Klasse `AttributeRestriction` enthalten die für die Nutzung einer Schnittstelle zu erfüllenden Bedingungen. Mittels der Klasse `DependRestriction` werden die oben dargestellten Abhängigkeiten der angebotenen Schnittstellen mit benutzten Schnittstellen spezifiziert.

Abbildung 4.10 zeigt das Meta-Modell für die Erstellung von bool'schen Bedingungen. Ausgehend von der abstrakten Klasse `Expression` werden zwei verschiedene Arten von Ausdrücken modelliert. Die Klasse `AttrExprPair` modelliert eine einzelne Bedingung. Sie hat ein linkes und ein rechtes `Attribut` verbunden durch einen `Operator`. Jedes `Attribut` hat einen bestimmten Attributtyp, da zum Beispiel der `LessOperator` nur den Vergleich von Zahlen unterstützt. Auf der rechten Seite des Klassendiagramms findet sich die Definition von *nicht*-, *oder*- sowie *und*-verknüpften Ausdrücken. Die Schnittstelle `NodeAttributesIn-`

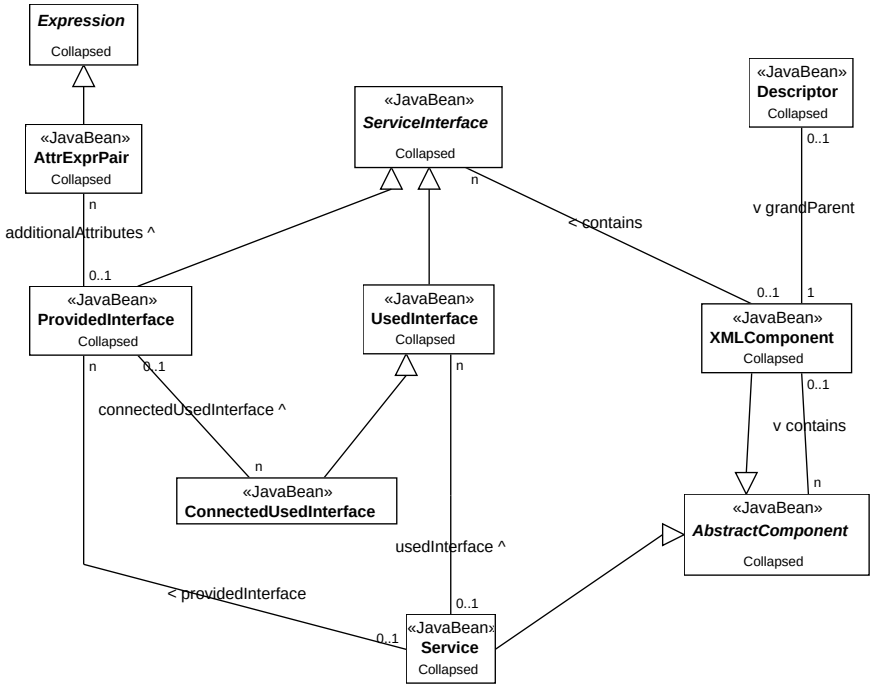

Abbildung 4.8: Meta-Modell des Komponentendiagramms

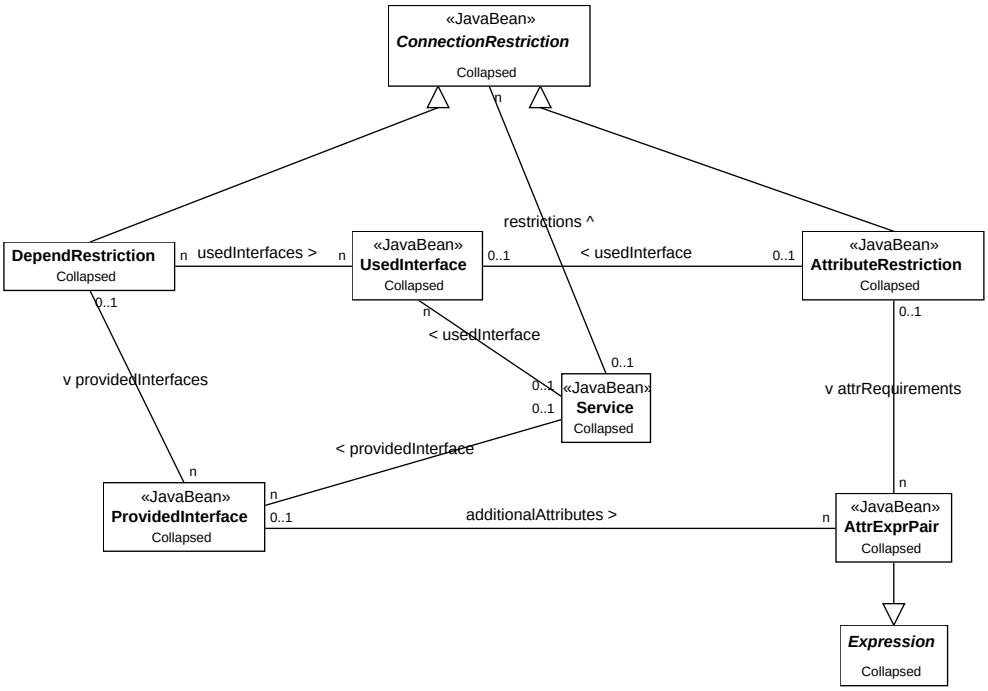

Abbildung 4.9: Komponentenrestriktionen

`terface` bietet die Möglichkeit, eine spezifische `NodeAttributeFactory` zu erhalten. Mittels dieser Klasse können Variablen in den Bedingungen durch ihren Wert und Typ ersetzt werden. Dies wird, während des Betriebs, für die Evaluation der Bedingungen für die Auswahl einer benötigten Schnittstelle genutzt.

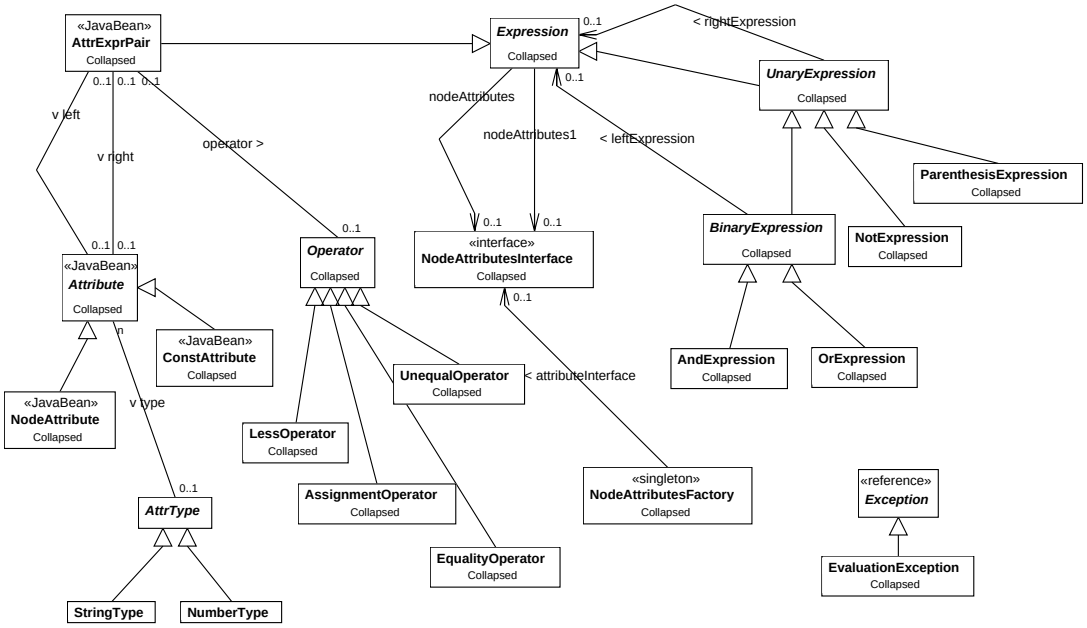

Abbildung 4.10: Meta-Modell der Bedingungen

4.4 Verteilungsdiagramme

In der Spezifikation der UML [OMG01, S.3/172] werden Verteilungsdiagramme als Konfigurationsdiagramme für Verarbeitungselemente (wie zum Beispiel Rechenknoten) und Softwarekomponenten, Prozesse und Objekte, die auf diesen Elementen ablaufen, beschrieben. Das Diagramm besteht aus einem (oder mehreren) Graphen, deren Knoten die Rechenknoten und deren Kanten die Kommunikationsverbindungen zwischen diesen Knoten sind. Die Rechenknoten haben einen Namen und einen Typ. Diese Knoten können dann Komponenten, Prozesse oder Objekte beinhalten, die auf diesem Rechenknoten zur Zeit ausgeführt werden. Komponenten, die Schnittstellen anderer Komponenten nutzen, können diese Benutzungsbeziehung analog wie bei Komponentendiagrammen durch einen gestrichelten Pfeil, der auf die genutzte Schnittstelle zeigt, darstellen. Abbildung 4.11 zeigt in dieser Darstellung die beiden Knoten „uther“ und „ginerva“, mit den jeweiligen Komponenten.

Die Spezifikation von Verteilungsdiagrammen erlaubt auch eine etwas andere Darstellung, die ebenfalls auf Graphen basiert. In dieser Darstellung werden

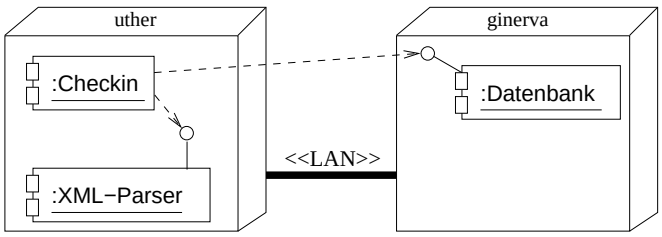

Abbildung 4.11: Darstellung der Knoten und Kommunikationsverbindungen

sowohl die Komponenten als auch die Rechenknoten als Knoten des Graphen dargestellt. Die Kanten zwischen Rechenknoten und Komponente werden durch die Bezeichnung ,,<<deploy>>“ annotiert, gestrichelt gezeichnet und haben einen Pfeil in Richtung des Knotens. Im Weiteren wird diese Kante als *Verteilungskante* bezeichnet. Eine Verteilungskante bedeutet, dass eine Instanz dieser Komponente auf den verbundenen Knoten ausgeführt werden *kann*. Wenn mehrere Kanten von einer Komponente ausgehen, dann sind die durch diese Kanten erreichten Knoten durch ein nicht-exklusives-Oder verbunden. Dies bedeutet, diese Komponente muss auf mindestens einem der Knoten laufen, aber kann durchaus auch auf mehreren der Knoten ausgeführt werden.

Diese Darstellung wird in dieser Arbeit genutzt, um sowohl die Verteilung eines Systems zu planen als auch die aktuelle Situation zu visualisieren.

4.4.1 Erweiterungen

Um genau zu spezifizieren, wie die einzelnen Komponenten auf die einzelnen Knoten verteilt werden sollen, ist es nicht ausreichend alleine den Namen des Knotens zu kennen. Typischerweise werden die Komponenten nicht auf Grund des Namens auf die Knoten verteilt, sondern eher auf Grund von Eigenschaften dieser Knoten. Durch diese Eigenschaften werden Klassen von Knoten definiert, die dieselben Eigenschaften aufweisen. In dieser Arbeit werden daher die Verteilungsdiagramme dahingehend erweitert, dass potentielle Knoten durch diese Eigenschaften genauer spezifiziert werden können.

Diese Eigenschaften können das Betriebssystem, dessen Version, die Version der Java-Virtual-Machine, die IP-Adresse des Knotens oder auch sein Name im lokalen Netz sein. Die Eigenschaften können dann durch verschiedene Operatoren mit Ausdrücken zu einer bool'schen Bedingung verknüpft werden. Die unterstützten Operatoren sind: *gleich*, *ungleich, kleiner* und *größer*.

Mittels dieser Bedingungen können dann aus der Menge der verfügbaren Knoten passende Knoten ausgewählt werden. Da einzelne Bedingungen einen Knoten nicht ausreichend beschreiben, müssen Kombinationen von mehreren Bedingungen unterstützt werden. Um dies zu erreichen, stehen die bool'schen Operatoren *und, oder* und *nicht* zur Verfügung. Mittels dieser Operatoren können dann

beliebige Ausdrücke erstellt werden. Diese komplexen Ausdrücke werden dann innerhalb des Rahmens des Knotens angezeigt.

Graphisch ist diese Darstellung allerdings weniger geeignet, da bei komplizierteren Ausdrücken nicht auf den ersten Blick ersichtlich ist, welche Klasse von Knoten durch diesen Ausdruck spezifiziert wird. Aus diesem Grund ist eine einfacher zu erfassende Visualisierung wünschenswert.

Wenn mehrere Bedingungen durch *und*-Operatoren verknüpft sind, können diese, in Anlehnung an die Darstellung von Attributbedingungen in StoryPattern in [FNT98, S.45], in einer Spalte untereinander angeordnet dargestellt werden. Wenn man mehrere dieser *und*-verknüpften Ausdrücke mit *oder* verknüpfen möchte, kann dies, wie oben bereits angedeutet, durch Kanten an mehrere Knoten ausgedrückt werden.

Die Abbildung 4.12 zeigt ein Verteilungsdiagramm. In diesem Verteilungsdiagramm wird die Komponente *Checkin* auf verschiedene Rechenknoten verteilt. Der linke Knoten enthält die einzelne Bedingung, dass die Komponente auf einem Knoten, der ,,uther“ heißt, ausgeführt werden kann. Der mittlere Knoten beschreibt die Bedingung, dass die Komponente auf einem Knoten ausgeführt werden kann, dessen Name nicht ,,uther“ ist und der ein Windows oder Irix Betriebssystem hat. Die letzte Bedingung besagt, dass der Knoten ein Linux-Betriebssystem nutzt und nicht die IP-Adresse 131.234.22.30 hat. Wie oben beschrieben sind diese drei Bedingungen durch *oder* verknüpft, daher bedeutet dieses Diagramm, dass die Komponente auf jedem Knoten ausgeführt werden kann, der die Bedingung

$$\begin{gathered}(hostname = uther) \\ \vee(hostname \neq uther \wedge (osname = windows \vee osname = irix)) \\ \vee(osname = linux \wedge ip \neq 131.234.22.30)\end{gathered}$$

erfüllt.

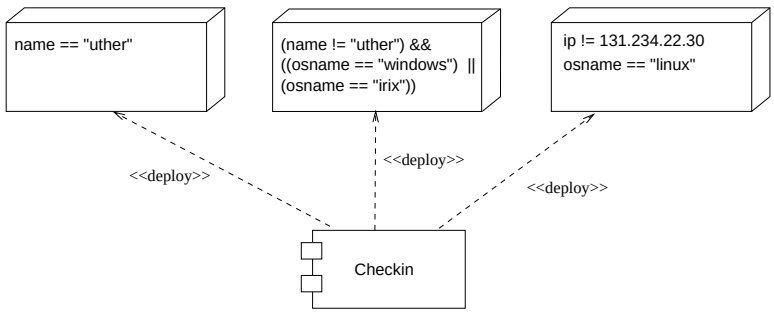

Abbildung 4.12: Darstellung der möglichen Komponentenverteilung

4.4.2 Visualisierung der aktuellen Verteilung

In der obigen Beschreibung der Verteilungsdiagramme wurde nur die statische Sicht in der Planung der Verteilung dargestellt. Wenn das verteilte System in Betrieb ist, ist es notwendig, die aktuelle Situation im System anzuzeigen. Im Gegensatz zur Darstellung der möglichen Komponentenverteilung in der Planung wird dann die realisierte Verteilung der Komponenten dargestellt. Dies beinhaltet, welche Knoten und mit welchen Eigenschaften zur Zeit in Betrieb sind und des Weiteren welche Komponenten auf welchen Knoten ausgeführt werden.

Für diese Anzeige werden auch Verteilungsdiagramme genutzt. Diese Verteilungsdiagramme werden angepasst, um darzustellen, dass es sich um eine Instanz-Sicht handelt. Alle Knoten werden mit ihren aktuellen Eigenschaften angezeigt, also Schlüssel und der Wert auf diesem Knoten, verbunden durch einen Zuweisungsoperator. Des Weiteren werden die Kanten im Diagramm mit `<<deployed>>` annotiert. Abbildung 5.22 zeigt die Administrationskonsole der Laufzeitumgebung mit einer Visualisierung der aktuellen Netzwerksituation.

4.4.3 Meta-Modell

Abbildung 4.13 enthält den für das Verteilungsdiagramm relevanten Ausschnitt des Meta-Modells. Die Meta-Modelle der Komponentendiagramme und Verteilungsdiagramme in den Abbildungen 4.8 und 4.13 sind Teil eines gemeinsamen Modells. Ein Verteilungsdiagramm besitzt deswegen immer ein zugehöriges Komponentendiagramm. In diesem Komponentendiagramm werden die Komponenten spezifiziert, die später im Verteilungsdiagramm auf die verschiedenen Rechenknoten verteilt werden. Das Verteilungsdiagramm und das zugehörige Komponentendiagramm besitzen jeweils eine Referenz auf die selbe Instanz der Klasse `Descriptor`.

Die Klasse `DeploymentComponent` ist für die Darstellung von Komponenten in einem Verteilungsdiagramm zuständig. Sie hat eine Beziehung zur abstrakten Klasse `AbstractComponent`, die bereits in Abschnitt 4.3.2 erläutert wurde. Diese Beziehung verbindet Komponenten- und Verteilungsdiagramme. Die Klasse `Deploy` beschreibt, auf welchen Knoten die Komponenten betrieben werden können. Diese Knoten werden mit den Klassen `AndPairNode` und `ExprNode` über die gemeinsame abstrakte Superklasse `DeploymentNode` modelliert. Mit einem `AndPairNode` wird die Verteilung auf einen Knoten im Netz über eine Menge von *und*-verknüpften Bedingungen modelliert. Wenn der Knoten mit einer allgemeinen bool'schen Bedingung ausgewählt werden soll, wird eine Instanz der Klasse `ExprNode` genutzt.

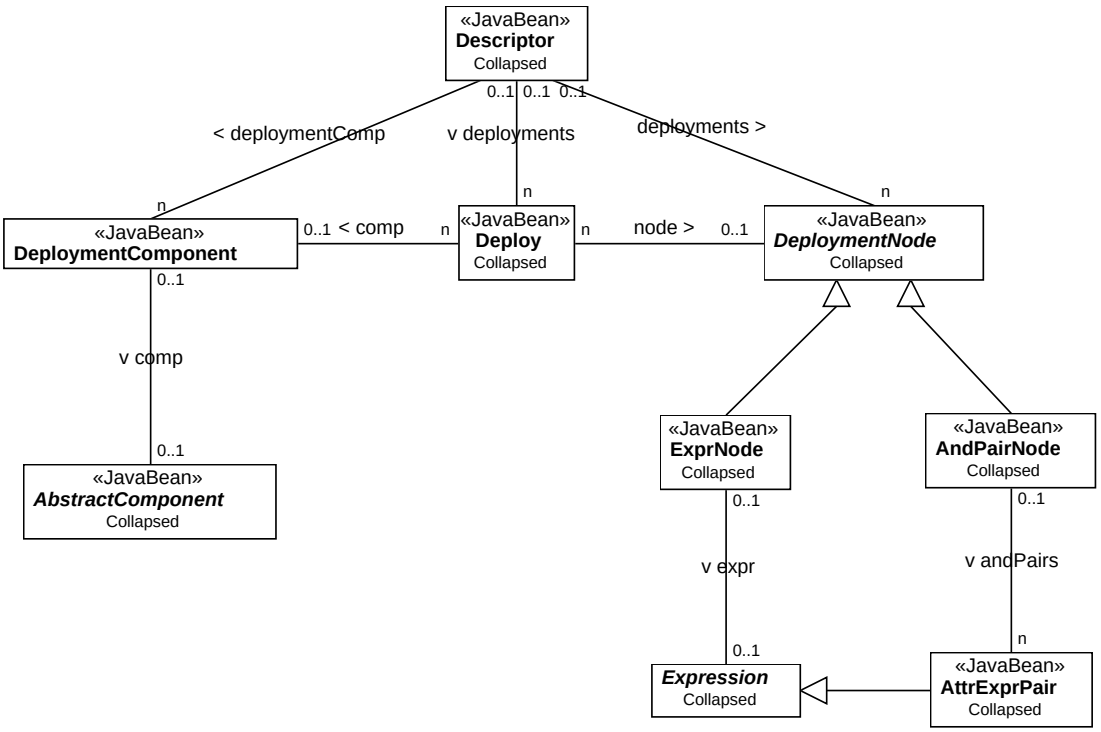

Abbildung 4.13: Meta-Modell des Verteilungsdiagramms

4.5 Zusammenfassung

Mittels der im ersten Teil dieses Kapitels vorgeschlagenen Unterstützung für den Entwurf und die Implementierung wird die Entwicklung wartbarer Dienste vereinfacht. Die Spezifikation der Struktur und des Verhaltens der Dienste auf hoher Abstraktionsebene und die Generierung von Java-Quelltext aus dieser Spezifikation vermeidet Inkonsistenzen zwischen Spezifikation und Implementierung, die ansonsten zu einer schwierigeren Anpassung und Erweiterung führen würden. Des Weiteren ist keine fehlerträchtige manuelle Implementierung der Spezifikation notwendig, so dass die spezifizierten Dienste weniger Fehler zeigen werden.

Die im zweiten Teil beschriebene Unterstützung für die Planung des Betriebs der Dienste ermöglicht durch die Nutzung von Komponentendiagrammen die Komposition mehrerer Dienste zu einer größeren verteilten Applikation. Die einzelnen Dienste oder auch ganze Kompositionskomponenten werden dann in einem Verteilungsdiagramm auf verschiedene Rechenknoten im System verteilt. Diese beiden Diagrammarten wurden um einige Ausdrucksmöglichkeiten erweitert, um die Integration der einzelnen Komponenten und Dienste und die Verteilung der Komponenten und Dienste genauer zu spezifizieren. Diese Spezifikation erlaubt den Betrieb des Systems durch die im nächsten Kapitel beschriebene Laufzeitumgebung.

Die Informationen der oben beschriebenen Diagramme sind Eingabedaten für die im nächsten Kapitel beschriebene Laufzeitumgebung. Dazu werden diese Informationen in eine für die Laufzeitumgebung verständliche XML-Datenstruktur übersetzt. Die Informationen aus der XML-Datei können auch wieder in das Fujaba-Werkzeug eingelesen werden, um die Informationen der XML-Datei als Diagramme zu visualisieren und geeignet zu ändern.

5 Betrieb von Komponenten

Die Laufzeitumgebung hat die zwei Aufgaben, den Entwickler der Komponenten zu entlasten sowie die Lauffähigkeit des Systems im lokalen Netz sicherzustellen. Die Konfiguration der verteilten Applikation, wie in Kapitel 4 angesprochen, wird von einer Laufzeitumgebung umgesetzt. Abbildung 5.1 zeigt die Einordnung dieses Kapitels in die Arbeit. Nach Entwurf und Implementierung der einzelnen Komponenten und deren Planung für den Betrieb wird mit Hilfe der Laufzeitumgebung der Betrieb entsprechend den Planungsdiagrammen (Komponenten- und Verteilungsdiagramme) des letzten Kapitels realisiert.

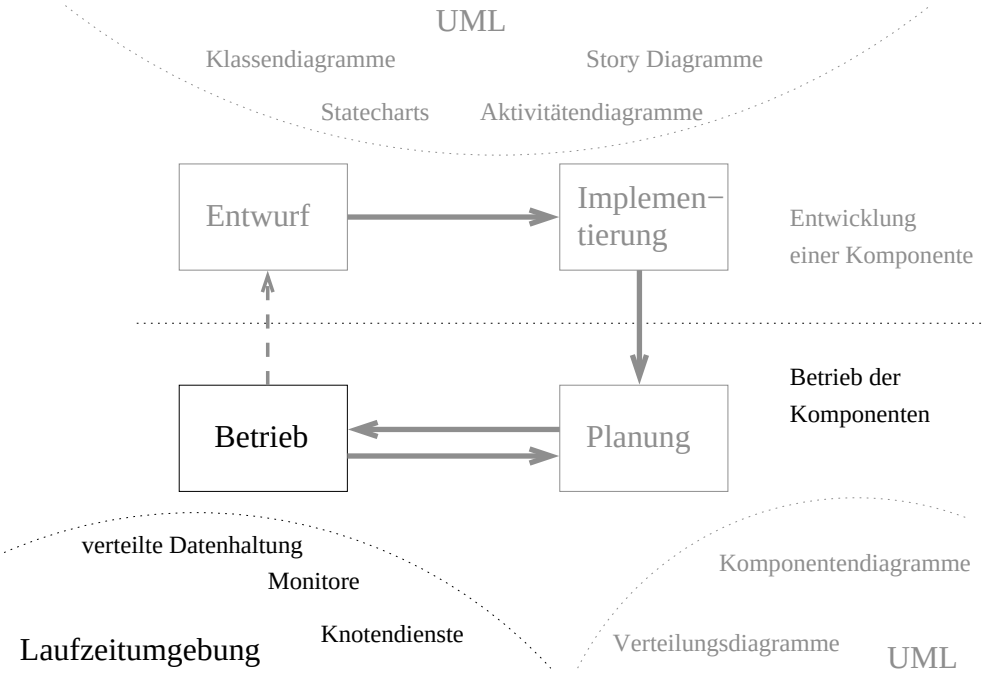

Abbildung 5.1: Einordnung der Laufzeitumgebung in das Gesamtsystem

Bezüglich des ersten Punktes ist das Ziel, dem Entwickler die Möglichkeit zu geben, sich (fast) vollständig auf die Entwicklung der Applikationslogik zu konzentrieren. Die Einbettung der Komponenten in das aktuelle System, wie zum Beispiel die Verknüpfung mit anderen Komponenten in Hinsicht auf bestimmte Bedingungen und die Anmeldung an den Lookupdiensten, wird vom Systemadmi-

nistrator konfiguriert und von der Laufzeitumgebung durchgeführt.

Der zweite wichtige Punkt betrifft die automatische Überprüfung des Zustandes des Systems. Diese Überprüfung soll bei Ausfällen von Netzknoten, Ausfällen von Netzwerkverbindungen (gesamte Teilnetze sind vorübergehend nicht erreichbar) sowie nach einem vorübergehenden kompletten Netzausfall die Funktionsfähigkeit des Systems sicherstellen.

Die Laufzeitumgebung soll explizit Heterogenität in Bezug auf die einzelnen Rechenknoten (Betriebssystem, Hardware, etc.) sowie auf die eingesetzten Dienste (Möglichkeit der Einbindung von nicht für die Laufzeitumgebung entwickelten Diensten) unterstützen. Einzige Voraussetzungen sind eine Standard Edition der Java-Virtual-Machine sowie TCP- und UDP-Protokolle. Wie in Abschnitt 2.2.1 bereits beschrieben, sind Transparenz, Skalierbarkeit, Fehlerbehandlung und Nebenläufigkeit Probleme verteilter Systeme. Zu deren Lösung bietet die Laufzeitumgebung Ansätze.

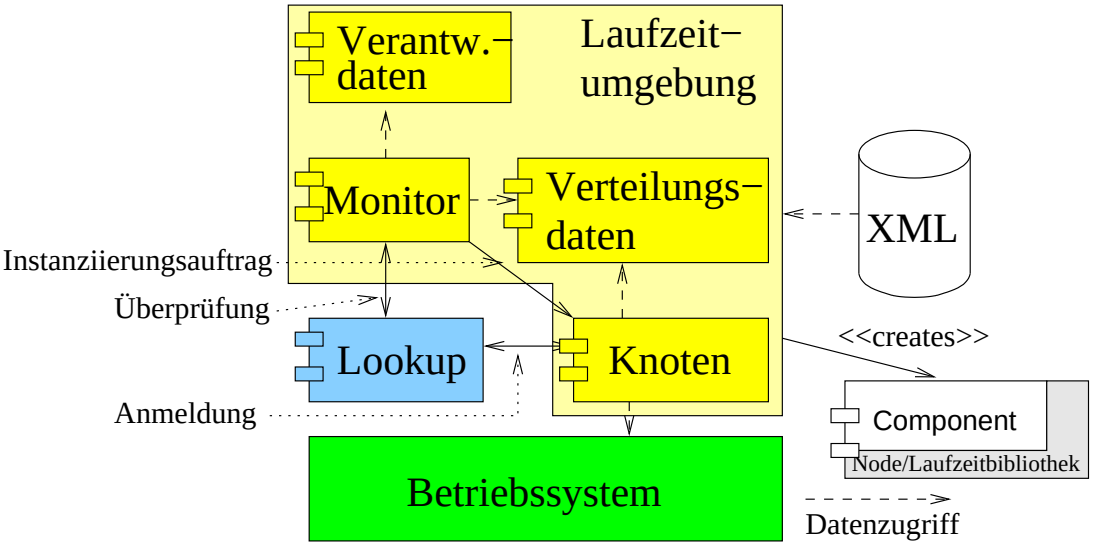

Abbildung 5.2: Übersicht der Laufzeitumgebung

Um die obigen Ziele zu erreichen, werden verschiedene Infrastrukturdienste benötigt. Abbildung 5.2 zeigt in einem Verteilungsdiagramm die beteiligten Komponenten und deren Interaktionen. Auf die Darstellung der Schnittstellen wurde für eine bessere Übersichtlichkeit verzichtet. Eine gemeinsame Datenbasis enthält die Informationen aus den in Kapitel 4 beschriebenen Komponenten- und Verteilungsdiagrammen[1]. Diese Daten werden verteilt in mehreren Diensten gespeichert. Für diese Datenhaltung wird ein sequentielles Konsistenzmodell (siehe Abschnitt 2.3.1) benötigt, damit keine Daten der Planungsdiagramme verloren gehen oder überschrieben werden. Zusätzlich muss auf jedem Knoten ein spezieller Dienst laufen (Knotendienst), der alle den lokalen Rechner betreffenden Aufgaben

[1] In Abbildung 5.2 als *Verteilungsdaten* dargestellt.

erledigt. Mittels dieses Dienstes werden andere Dienste auf diesem Rechenknoten gestartet. Die korrekte Dienstanmeldung und Dienstintegration wird durch eine Laufzeitbibliothek im Prozess der neuen Komponente durchgeführt.

Im System müssen mehrere Monitore existieren, welche die Lauffähigkeit des Systems überprüfen und im Fehlerfall geeignete Gegenmaßnahmen treffen. Diese Monitore übernehmen die Verantwortung für Komponenten aus den Verteilungsdiagrammen, dass diese Dienste im Netz verfügbar sind. Um die Verfügbarkeit dieses Systems zu erhöhen, werden diese Verantwortungen der Monitore gegenüber den Komponenten in einer zweiten Datenbasis gespeichert, die mit dem schwächeren Konsistenzmodell der eventuellen Konsistenz arbeitet. Analog zum Konzept der Mietverträge der Jini-Technologie werden auch diese Verantwortungsdaten mit Mietdauern versehen, so dass ausgefallene Monitore ersetzt werden können. Die im Weiteren vorgestellte Lösung betrachtet nur ein lokales Netz, da durch die Fokussierung auf lokale Netze die Skalierbarkeit der Lösung erhöht werden kann, indem Multicast-Nachrichten [Wyb90] für eine effiziente 1-n-Kommunikation genutzt werden.

Des Weiteren existiert ein Dienst für die Überprüfung der korrekten Komponentenkomposition nach [Gie01b] sowie eine Administrationskonsole, welche die Visualisierung des aktuellen Systems und dessen initiale Konfiguration durch Laden der Verteilungsbeschreibungen in die Datenbasis des Systems ermöglicht.

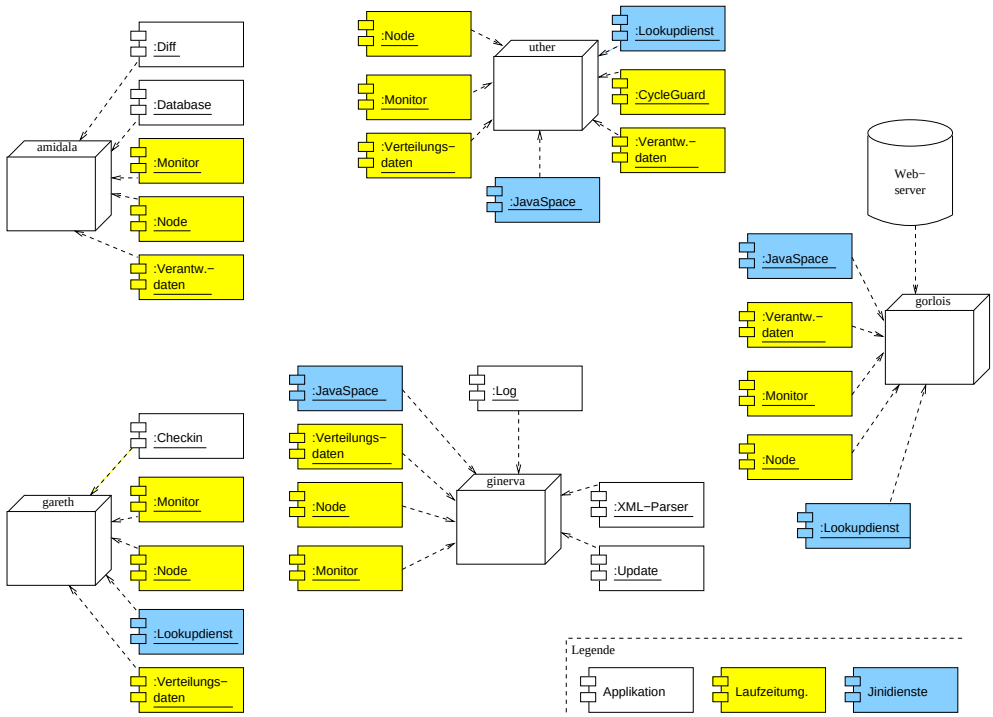

Abbildung 5.3: Beispiel mit allen beteiligten Diensten

In Abbildung 5.3 ist ein Beispiel für ein konkretes System dargestellt. Das Sy-

stem besteht aus fünf Rechenknoten. Auf diesen Knoten laufen diverse Dienste. Die Dienste mit hellblauem Hintergrund sind Dienste, die von Jini benötigt bzw. zur Verfügung gestellt werden. Dies beinhaltet zum Beispiel den Lookupdienst, bei dem sich die gestarteten Dienste anmelden und über den nach bestimmten Diensten gesucht werden kann, und einen Dienst, der JavaSpaces unterstützt. Dienste mit gelben Hintergrund sind die Dienste der im Weiteren vorgestellten Laufzeitumgebung. Diese Dienste stellen die Grundlagen für das reibungslose Funktionieren des verteilten Systems zur Verfügung. Dienste mit weißem Hintergrund sind die Komponenten des in Abschnitt 1.5 vorgestellten Beispielszenarios. Diese Komponenten realisieren das verteilte Versionierungs- und Konfigurationsmanagementsystem DSD. Die Laufzeitumgebung stellt sicher, dass jederzeit alle diese Applikationsdienste zur Verfügung stehen.

Sicherheit wird in der durch die einzelnen Komponenten und die Laufzeitumgebung realisierten komplexen Applikation zu einem großen Teil über die korrekte Anwendung von Java-Sicherheits-Strategien (Policies) erreicht. In diesen Sicherheitsstrategien werden die Kommunikationsmöglichkeiten der einzelnen Komponenten eingeschränkt und die Ausführung von Klassen auf digital signierte Klassen beschränkt.

5.1 Verteilte Speicherung der Verteilungsdaten

Grundlage der Laufzeitumgebung ist eine verteilte Datenhaltung der in den Komponenten- und Verteilungsdiagrammen spezifizierten Planung des Betriebes. Die Daten werden dabei verteilt auf mehreren Knoten im lokalen System gespeichert, um Fehlertoleranz gegenüber Ausfällen von Knoten und des Netzwerks zu erreichen. Diese Funktionalität wird auf diesen Knoten von einem `Deployment-Storage`-Dienst realisiert.

Um einen hinreichend konsistenten Zustand der verschiedenen Kopien der Daten im Netz zu haben, wird ein sequentielles Konsistenzmodell benötigt (siehe 2.3.1). Schreibzugriffe auf diese Datenhaltung werden vom Administrator des Systems durchgeführt, wenn er die Planungsdiagramme ändert, Lesezugriffe werden von allen Diensten der Laufzeitumgebung durchgeführt und treten deshalb um Größenordnungen öfter auf. Aus diesem Grunde muss die Fehlertoleranz bei Lese- und Schreibzugriffen unterschiedlich sein, so dass die Wahrscheinlichkeit, dass ein Lesezugriff nicht durchführbar ist, niedriger ist als die Wahrscheinlichkeit für die Nichtdurchführbarkeit eines Schreibzugriffes. Zudem sollten Lesezugriffe signifikant schneller durchführbar sein als Schreibzugriffe.

5.1.1 Konsistenzprotokolle für sequentielle Konsistenz

Um obige Anforderungen zu realisieren, sind in der Literatur verschiedene Algorithmen vorgestellt worden. Diese Algorithmen unterscheiden sich in der Hin-

sicht, ob es einen primären Server gibt, auf dem die Schreibzugriffe ausgeführt werden, oder ob der Schreibzugriff auf beliebigen Datenverwaltungsknoten ausgeführt werden kann.

Ein einfacher Algorithmus, um die Repliken auf mehreren Knoten konsistent zu halten, basiert auf dem *Write-All-Read-One* Konzept [BHG87, S.267]. Lesezugriffe können hier auf jeder Replik durchgeführt werden. Schreibzugriffe müssen allerdings auf allen Kopien durchgeführt werden. Jeder Schreibzugriff muss dabei entweder auf allen Repliken oder keiner Replik durchgeführt werden. Dies kann mit Hilfe des Two-Phase-Commit-Protokolls [Gra78] erreicht werden. Mit diesem Konzept sind die Repliken zwar immer konsistent, aber das System ist nicht fehlertolerant, da jeder Schreibzugriff blockiert, auch wenn nur ein System ausgefallen ist.

Alsberg und Day beschreiben in [AD76] ein Protokoll, bei dem die Daten auf einem primären Server liegen. Der primäre Server wird von Ersatzsystemen unterstützt, die im Normalfall die Anfragen einfach an das primäre System weiterleiten. Alle Schreiboperationen müssen bei blockierendem Verhalten vor der Rückmeldung zum anfragenden System erst bei allen Ersatzsystemen durchgeführt werden. Bei einem nicht-blockierenden System werden die Schreiboperationen auf den Ersatzsystemen erst später durchgeführt. Bei einem Fehler des primären Systems wird seine Aufgabe von einem Ersatzsystem übernommen. Das blockierende Protokoll erfüllt die Forderungen von sequentieller Konsistenz. Bei einem nicht-blockierenden Protokoll kann nicht davon ausgegangen werden, dass im Fehlerfall bereits alle Ersatzsysteme die Schreiboperation ausgeführt haben. In diesem Fall kann die Konsistenz gefährdet sein.

Aus diesem Grund ist das blockierende Protokoll vorzuziehen. Ein Nachteil in diesem Fall ist allerdings, dass die Geschwindigkeit des gesamten Systems vom langsamsten Teilsystem bestimmt wird. Die in Abschnitt 2.2 beschriebenen Ziele der höheren Verarbeitungsgeschwindigkeit und Skalierbarkeit eines verteilten Systems sind daher nicht realisierbar. Problematisch ist auch, dass bei Netzwerkpartition keine Schreibzugriffe durchführbar sind, da die Schreiboperationen nicht auf allen Ersatzsystemen durchführbar sind.

Abstimmungsprotokolle (eingeführt von Thomas in [Tho79] und generalisiert von Gifford in [Gif79]) bieten in dieser Hinsicht Vorteile. Bei diesen Abstimmungsprotokollen müssen nicht auf allen Teilsystemen die Schreib- und Leseoperationen durchgeführt werden, sondern nur auf einem bestimmten Teil. In [Tho79] müssen diese Operationen nur auf mehr als der Hälfte der Teilsysteme durchgeführt werden. Dadurch ist gewährleistet, dass keine zwei Operationen sich gegenseitig überschneiden. Bei einer Netzwerkpartition besteht die Möglichkeit, dass in der einen Hälfte des Netzes Schreibzugriffe möglich sind, wenn dort mehr als die Hälfte der Rechenknoten angesiedelt sind.

Das von Gifford beschriebene Protokoll generalisiert obiges Konzept in Hinsicht auf Gewichtung der verschiedenen Teilsysteme. Es wird im nächsten Abschnitt beschrieben und für die verteilte Speicherung der Planungsdaten genutzt.

5.1.2 Gewichtete Abstimmung

Bei diesem Verfahren werden die einzelnen Teilsysteme gewichtet. Durch diese Gewichtung können bestimmte Systeme, die besonders schnell oder besonders zuverlässig sind, bevorzugt werden. Um einen Zugriff durchzuführen, ist es notwendig, eine bestimmte Menge von Stimmen zu erhalten, also diese Operation auf einer bestimmten Anzahl von Systemen durchzuführen. Diese Menge von Stimmen kann durchaus bei Lese- und Schreibzugriffen unterschiedlich sein. Für Lesezugriffe müssen bei insgesamt N Stimmen im Gesamtsystem mindestens N_l und für Schreibzugriffe mindestens N_s Stimmen eingeholt werden. Bei der Wahl dieser Werte müssen die folgenden beiden Bedingungen eingehalten werden:

$$N_l + N_s > N \tag{5.1}$$

$$N_s > N/2 \tag{5.2}$$

Die Bedingung 5.1 stellt sicher, dass keine Schreib-Lese-Konflikte auftreten; die Bedingung 5.2 stellt sicher, dass keine Schreib-Schreib-Konflikte auftreten. Im Rahmen dieser Bedingungen kann das Verfahren für das aktuelle Problem konfiguriert werden. Zum Beispiel kann in einem System, in dem sehr viele Lesezugriffe vorkommen, N_l sehr niedrig gewählt werden, damit die Lesezugriffe schnell ausgeführt werden können. Die seltener auftretenden Schreibzugriffe sind dann allerdings langsamer, da aus Bedingung 5.1 folgt, dass $N_s > N - N_l$ gelten muss, und somit N_s bei kleinem N_l groß wird, und daher die Stimmen vieler Systeme eingeholt werden müssen.

Jedes Datenelement in dem System hat eine Versionsnummer. Diese Versionsnummer wird um eins erhöht, wenn ein Schreibzugriff auf dieses Element durchgeführt wird. Diese neue Versionsnummer und die zu schreibenden Daten werden dann mittels einer verteilten Transaktion auf jedem über die Abstimmung gefundenen System geschrieben. Veraltete Daten auf diesen Teilsystemen werden dadurch automatisch überschrieben. Bei einem Lesezugriff werden die Datenelemente von den über die Abstimmung gefundenen Systemen gebündelt über eine Transaktion gelesen und das Element aus dieser Menge mit der höchsten Versionsnummer enthält die aktuellen Daten.

Bei Ausfall eines einzelnen Systems wird dieses System nicht für die Abstimmung genutzt und das Gesamtsystem kann weiterarbeiten, solange die nötigen Stimmen für Lese- oder Schreibzugriffe gesammelt werden können. Bei einer Netzwerkpartition wird die Konsistenz gewährleistet, da durch die Bedingung 5.2 nur in einem Teil des Netzwerks Schreibzugriffe möglich sind. Lesezugriffe sind auch in den anderen Teilen des Netzes möglich, solange die nötigen Stimmen gesammelt werden können. Vorteilhaft ist, dass nach Behebung der Probleme keine Recovery durchgeführt werden muss. Wenn ein Teilsystem nach einem Ausfall wieder funktionsfähig ist, wird es bei der nächsten Schreiboperation, an der es beteiligt ist, automatisch auf den aktuellen Stand gebracht.

Beispiel

Das folgende Beispiel basiert auf der folgenden Konfiguration: 4 Knoten werden für das System genutzt. Die Stimmen sind folgendermaßen verteilt: $a = 4$, $b = 2$, $c = 3$ und $d = 1$. Dies ergibt eine Gesamtstimmenanzahl $N = 10$. Die nötige Stimmenanzahl für Lesezugriffe beträgt $N_l = 5$, für Schreibzugriffe $N_s = 6$. Mit diesen Werten sind die Bedingungen 5.1 und 5.2 erfüllt. Alle Knoten enthalten im initialen Zustand keine Daten (Versionsnummer: 0).

In Abbildung 5.4 wird eine Reihe von Schreib- und Lesezugriffen bei teilweisen Ausfällen des Systems durchgeführt. Durchgestrichene Knoten sind ausgefallen und stehen für die Operation nicht zur Verfügung. Knoten mit gestricheltem Rahmen werden in der Abstimmung nicht genutzt. Knoten mit durchgezogenem Rahmen werden in der Abstimmung genutzt.

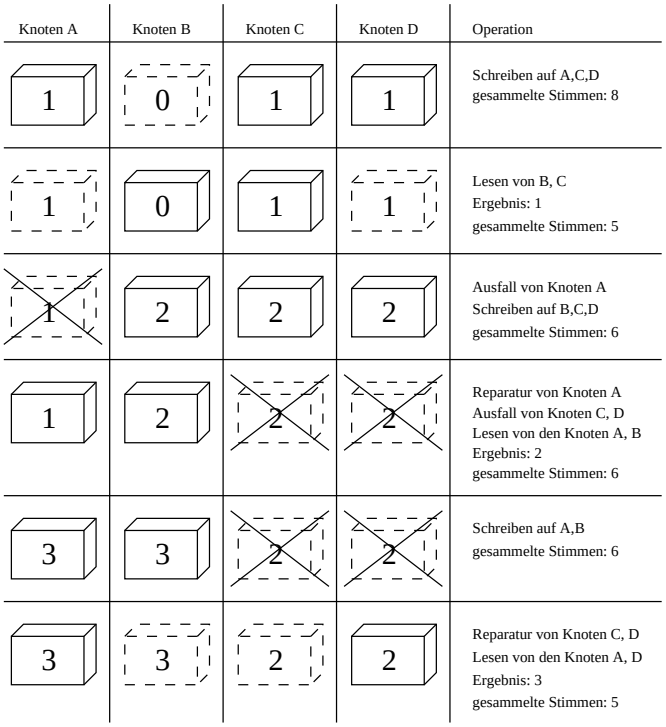

Abbildung 5.4: Abstimmungs-Beispiel

Analyse und Diskussion

Jalote analysiert in [Jal96, S.298] die Fehlertoleranz des Konzepts[2]. In einem konkreten Beispiel geht er davon aus, dass jeder Knoten durchschnittlich 30 Tage bis zu einem Ausfall verfügbar ist. Ausgehend von der Dauer einer Reparatur eines

[2]Für die Herleitung der benutzten Formeln siehe [Jal96, S.295].

Systems errechnet er die Werte von Tabelle 5.1. Bei einer in einem wichtigen System durchaus anzunehmenden Reparaturdauer von einem Tag und darunter liegt die optimale Anzahl der Systeme (Spalte 2) bei $N_O = 15$, die durchschnittliche Zeit, bis ein Ausfall auftritt, liegt dann bei $MTTF = 4794$ Tagen und die Verfügbarkeit bei $\alpha = 99.9791\%$.

Reparaturdauer	Optimales N_O	Optimale $MTTF$	Verfügbarkeit α
1	15	4794	0,999791
2	9	213	0,990735
3	5	85	0,965909
4	5	57	0,93429
5	3	45	0,9
10	3	30	0,75
15	3	25	0,625

Tabelle 5.1: Analyse der gewichteten Abstimmung

Gifford beweist im Anhang von [Gif79], dass dieses Verfahren serielle Konsistenz nach [EGLT76] bietet. Eswaran et al. schreiben allerdings in ihrem Artikel nur über Serialisierbarkeit bei Transaktionen und da Gifford in seinem Papier auch serielle Konsistenz mit Transaktionen erklärt, kann man davon ausgehen, dass Gifford mit serieller Konsistenz Serialisierbarkeit bei Transaktionen meint. Nach [TvS02, S.302] und [AW94] ist Serialisierbarkeit bei Transaktionen vergleichbar zu sequentieller Konsistenz. Der Unterschied liegt in der Granularität. Sequentielle Konsistenz wird auf einzelnen Lese- und Schreiboperationen, Serialisierbarkeit auf Mengen von Operationen definiert.

Bernstein et al. fassen die Nachteile dieses Konzepts in [BHG87, S.299] zusammen: Der erste Nachteil ist, dass Lesezugriffe teuer sind, da immer mehrere Teilsysteme an dem Lesezugriff beteiligt sind. In dem Fall $N_l = 1$ tritt dieses Problem nicht auf, allerdings muss dann nach Bedingung 5.1 $N_s = N$ gelten, wodurch dann bei Schreibzugriffen keine Fehlertoleranz gewährleistet ist. Der zweite Nachteil ist, dass viele Teilsysteme nötig sind, um Ausfälle zu maskieren. In dem einfachen Mehrheitsfall nach Thomas [Tho79] muss das System $2k + 1$ Teilsysteme umfassen, um k Ausfälle von Teilsystemen zu tolerieren. Der dritte Nachteil ist, dass die Konfiguration des Systems, dies beinhaltet die Namen der einzelnen teilnehmenden Systeme und deren Stimmen, festgelegt sein muss und nicht im Betrieb geändert werden kann.

Trotz dieser Nachteile überwiegen die Vorteile dieses Konzepts vor allem bei Netzwerkpartition und hinsichtlich der Flexibilität bei der initialen Konfiguration des Systems. Es wird deshalb für diese Arbeit genutzt und implementiert.

Implementierung

Die für diese Arbeit erstellte Implementierung dieses Konzepts basiert auf Jini, JavaSpaces und dem Two-Phase-Commit-Protokoll. Die Möglichkeit, bei einem Jini-Dienst Funktionalität in den Proxy zu integrieren, erlaubt eine sehr elegante Implementierung. Das gesamte Konzept wird im Proxy implementiert. Die einzelnen Datenhaltungssysteme sind persistente JavaSpaces. Die Lese- bzw. Schreibzugriffe auf den einzelnen JavaSpaces werden über Transaktionen vorgenommen, die durch das Two-Phase-Commit-Protokoll von einem Transaktionsdienst verwaltet werden.

Abbildung 5.5 zeigt den für die gewichtete Abstimmung verantwortlichen Teil des Klassendiagramms des realisierten Systems. Das Design ist an das Originaldesign von Gifford angelehnt. Dies betrifft auch die Benennung der Klassen. Die Implementierung erlaubt die parallele Nutzung von mehreren verteilten Dateien mit unterschiedlichen Dateinamen. Diese Dateien können unterschiedliche Konfigurationen der nötigen Stimmen für Lese- und Schreibzugriffe besitzen. Im aktuellen System wird nur eine Datei genutzt. Die Klasse `WeightedVoter` ist der Ausgangspunkt für die Nutzung des Systems. Über die `suites`-Assoziation kann für eine bestimmte Datei eine `FileSuite` erreicht werden. In dieser Klasse sind die Algorithmen für Lese- und Schreibzugriffe (`read()` / `write()`) gekapselt. Die Methoden `collectReadQuorum()` und `collectWriteQuorum()` sammeln die nötigen Stimmen für den jeweiligen Zugriff. Von diesen Methoden wird eine `VotingException` geworfen, falls für den Zugriff nicht genügend Stimmen gesammelt werden konnten. Eine Instanz der Klasse `Representative` kapselt den Zugriff auf einen JavaSpace mit einem bestimmten Namen. Die Methoden der Klasse `DistributedEntry` lesen und schreiben dann die Daten für eine bestimmte Datei auf einem bestimmten persistenten JavaSpace.

Die Konfigurationsdaten werden aus der Datei `VotingConfig.xml` gelesen. Nach diesen Konfigurationsdaten wird ausgehend von einer Instanz der Klasse `WeightedVoter` die korrekte Objektstruktur erzeugt. Diese Instanz wird dann als Proxy eines Jini-Dienstes verwendet. Wenn der Proxy dann auf dem Client genutzt wird, werden über die Methode `initialize()` die beteiligten JavaSpaces gesucht. Danach können die Daten aus einer `FileSuite` gelesen und geschrieben werden.

Die Abbildungen 5.6 und 5.7 zeigen die Geschwindigkeit der Implementierung bei 200 aufeinanderfolgenden Lese- und Schreibzugriffen. Die minimalen, maximalen und durchschnittlichen Zugriffszeiten in Millisekunden zeigt Tabelle 5.2. Der erste Zugriff von einem Client dauert durch den Aufbau von Caches zwischen 7 und 8 Sekunden.

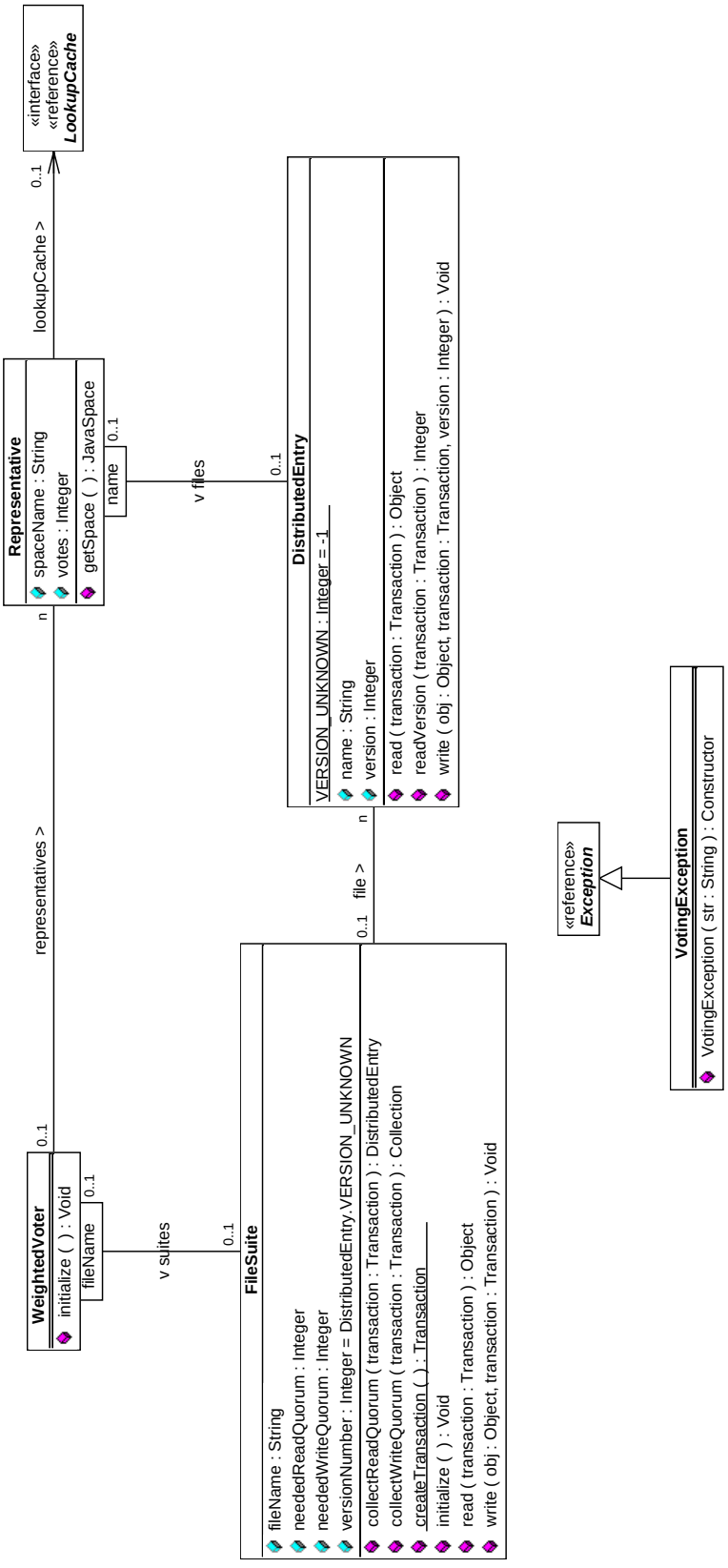

Abbildung 5.5: Klassendiagramm für Weighted Voting

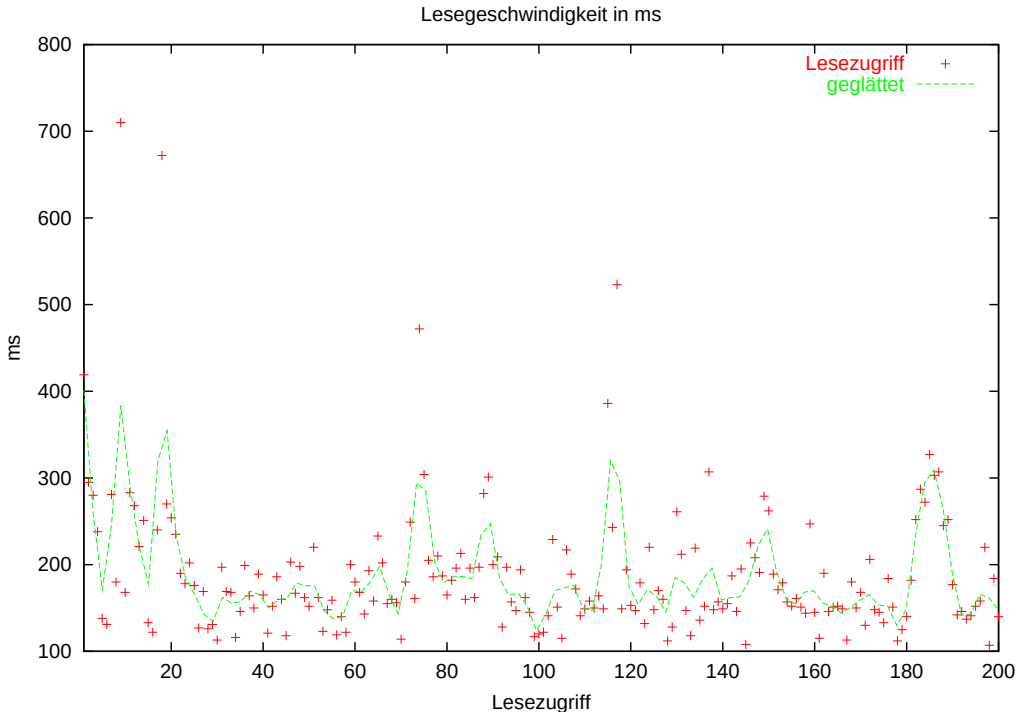

Abbildung 5.6: Lesegeschwindigkeit

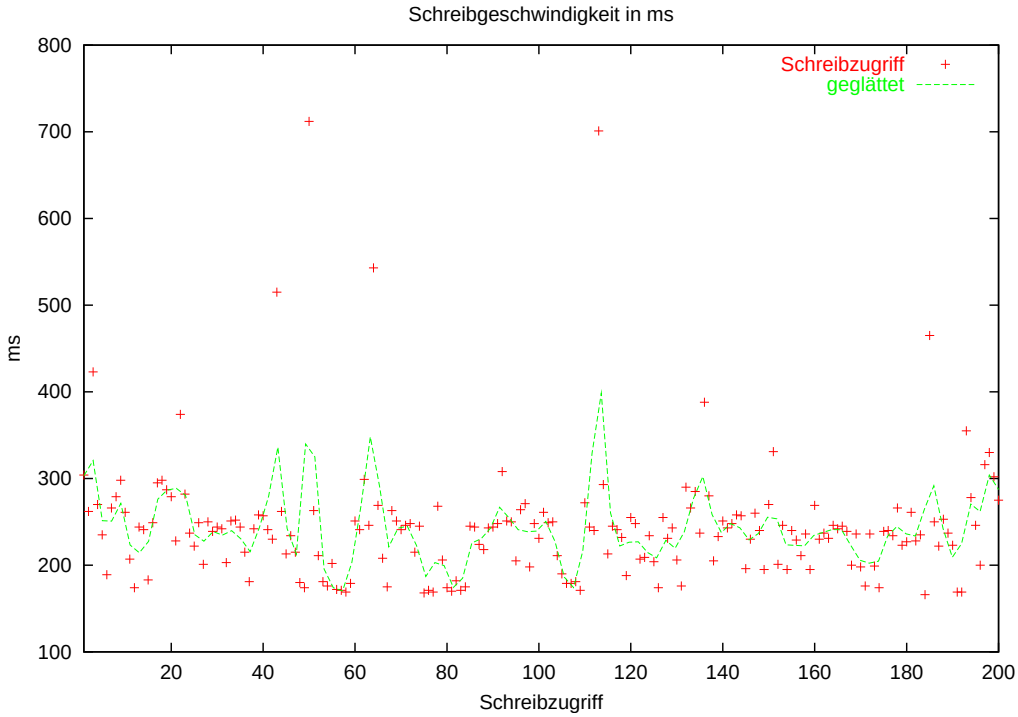

Abbildung 5.7: Schreibgeschwindigkeit

Zugriffszeit in Millisek.	Lesezugriff	Schreibzugriff
min.	107	166
max.	710	712
durchschnittl.	188	243

Tabelle 5.2: Zugriffszeiten

5.2 Knotendienst

Auf jedem Rechenknoten im lokalen Netz, der für das verteilte System genutzt werden soll, muss, wie in der Einleitung dieses Kapitels beschrieben, ein Knotendienst ausgeführt werden. Dieser Dienst ermöglicht es, andere Dienste auf diesem Rechner zu starten und zu beenden. Um die Auslastung des lokalen Netzes berechnen zu können oder um Dienste aufgrund der Auslastung migrieren zu können, gibt der Knotendienst auf Anfrage Informationen über die lokalen Auslastungsdaten bekannt. Der Knotendienst sollte nach dem Booten des Rechners vom Betriebssystem automatisch gestartet werden, um den Rechenknoten auch ohne Benutzereingriffe wieder in das System einfügen zu können.

Jeder Knoten hat bestimmte Eigenschaften wie Betriebssystem, IP-Adresse, evtl. Hostname sowie die Eigenschaften seiner virtuellen Maschine (JDK-Version, etc.). Tabelle 5.3 zeigt die möglichen Eigenschaften der Knoten. Auf jedem Rechner sollte nur ein Knotendienst gestartet sein.

Name	Beschreibung
hostname	Name des Knotens
java.version	Version der Java Virtual Machine
os.name	Name des Betriebssystem des Knotens
os.version	Version des Betriebssystems
ip	IP-Adresse des Knotens

Tabelle 5.3: Mögliche Knotenattribute

5.2.1 Starten und Beenden von Diensten

Mittels der `ManageService`-Schnittstelle werden auf den einzelnen Knoten Dienste gestartet bzw. beendet.

Jeder Dienst hat einen Namen, der sich durch seine Zugehörigkeit zu einer Kompositionskomponente zusammensetzt (siehe Abschnitt 4.3) und eindeutig definiert ist. Dabei werden die Namen der Vaterkomponenten ähnlich wie bei Java-Paketen mit einem Punkt verbunden. Der Datenbank Dienst in der DSD Komponente heißt also „DSD.Datenbank“ .

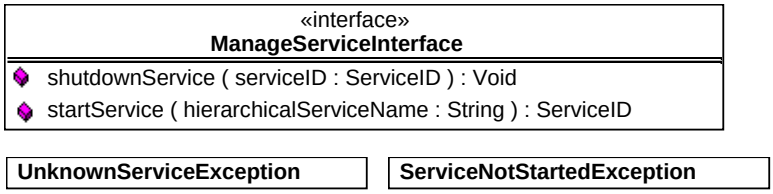

Abbildung 5.8: Schnittstelle, um Dienste zu starten und zu beenden.

Beim Start eines Dienstes durch den Knotendienst wird zuerst mit dem Namen nach den zugehörigen Informationen in der verteilten Datenhaltung gesucht. Wenn die Informationen gefunden wurden, wird der Dienst in einer neuen virtuellen Maschine gestartet. Der Dienst wird in einer neuen virtuellen Maschine gestartet, damit der Knotendienst durch einen Fehler in diesem Dienst nicht betroffen ist. Bei einem Absturz der virtuellen Maschine durch diesen Dienst ist der Knotendienst weiter funktionsfähig. Die zurückgegebene `ServiceID` identifiziert den Dienst eindeutig und mit ihr kann der Dienst später beendet werden. Falls die Beschreibung des Dienstes nicht gefunden werden kann, wird eine `UnknownServiceException` geworfen. Des Weiteren wird eine `ServiceNotStartedException` geworfen, wenn ein Dienst beendet werden soll, der gar nicht auf diesem Knoten gestartet wurde.

Der Start eines Dienstes beinhaltet die Anmeldung an den Lookupdiensten und die Suche nach den benötigten Schnittstellen. Wenn ein Dienst im Lookupdienst gefunden wird, der eine benötigte Schnittstelle implementiert, die in den Verteilungsbeschreibungen eventuell spezifizierten Bedingungen erfüllt und auch bei der Überprüfung der Abhängigkeiten kein Zyklus erkannt wird, wird dem gestarteten Dienst der gefundene Dienst mitgeteilt.

5.2.2 Lokale Daten

Jeder Knoten bietet die Möglichkeit, Informationen über den aktuellen Zustand des Rechenknotens zu erhalten. Dies können andere Dienste nutzen, um zum Beispiel einen Lastausgleich der genutzten Rechenknoten zu ermöglichen. Des Weiteren können über diese Schnittstelle die Attribute des Knotens aus Tabelle 5.3 abgerufen werden.

Die Schnittstelle bietet die Möglichkeit, sich für den Erhalt der regelmäßig verschickten Daten anzumelden. Der datenerhaltende Dienst muss ein Objekt übergeben, das die `RemoteDataListener` Schnittstelle implementiert. Auf diesem Objekt wird dann die Methode `consume` mit den passenden Parametern aufgerufen. Da dieses Objekt über das Netzwerk zum Rechenknoten gesendet wird und die Methode auf diesem Rechenknoten aufgerufen wird, muss dieses Objekt per RMI die Daten über das Netzwerk zurücksenden.

Alle Daten enthalten eine Referenz auf den Knoten von dem sie versandt wurden und einen Zeitstempel, der die Millisekunden seit dem 1. Januar

1970 00:00:00 Uhr GMT angibt. Es gibt verschiedene knotenspezifische Daten:

- `LoadData`: Dieses Objekt enthält die durchschnittliche Auslastung des Knotens in Bezug auf verschiedene Zeitfenster (letzte Minute, letzten 5 Minuten, letzten 10 Minuten)
- `MemoryData`: Hier wird die Größe des physikalischen Speichers und die Größe des Auslagerungsspeichers sowie jeweils deren freie Größe angegeben.

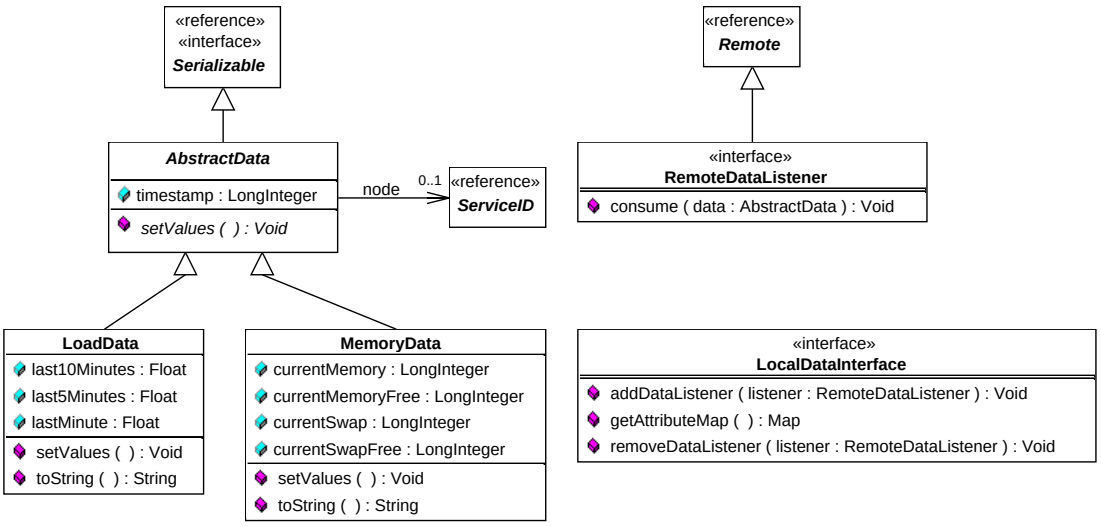

Abbildung 5.9: Lokale Daten und Observer Pattern

Da diese Informationen durch die virtuelle Maschine nicht zur Verfügung gestellt werden, muss eine plattformspezifische Bibliothek benutzt werden. Diese Bibliothek wird mit Hilfe des Java Native Interface (JNI) angesprochen. Für Linux wurde eine Bibliothek für die Laufzeitumgebung realisiert, die diese Daten ausliest.

5.3 Einbettung der Komponenten

Wie im vorherigen Abschnitt auf der rechten Seite in Abbildung 5.2 dargestellt, wird der einzelne Dienst durch den Knotendienst gestartet. Der Dienst wird in einer neuen virtuellen Maschine gestartet. In dieser neuen virtuellen Maschine werden die einzelnen Aufgaben für die Einbettung der Komponenten in die Jini-Umgebung durch eine Laufzeitbibliothek durchgeführt. Der Entwickler muss sich nicht mit dem Starten des Dienstes, der Anmeldung an den Lookupdienst oder dem Verbinden der Schnittstellen befassen. Diese Funktionalität wird durch die Laufzeitbibliothek geleistet. Um diese Funktionalität speziell für den Dienst zu

realisieren, benötigt der zu startende Dienst eine Klasse, welche die FrameWork-Schnittstelle (Abbildung 5.10) erfüllt. Mittels dieser Klasse erhält die Laufzeitbibliothek die Information, ob der Dienst ein Backend hat. Wenn ja, wird dieses Backend mit Hilfe dieser Klasse gestartet und beendet. Des Weiteren enthält diese Klasse eine Methode, um den Proxy und Attribute für die Komponente zu erzeugen, die dann im Lookupdienst eingetragen werden. Nach der erfolgreichen Anmeldung der Komponente am Lookupdienst wird die dabei erzeugte `ServiceID` der FrameWork-Klasse mitgeteilt. Als letzte Operation wird ein gefundener benötigter Dienst der Komponente mitgeteilt.

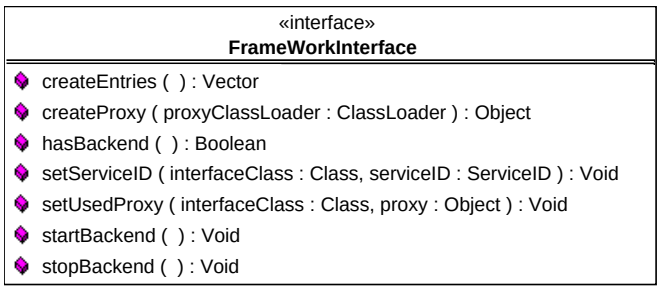

Abbildung 5.10: FrameWorkInterface

Es wird von der Laufzeitbibliothek garantiert, dass die backendspezifischen Methoden (also `hasBackend()`, `startBackend()` und `stopBackend()`) vor dem Aufruf der Methode `createProxy()` aufgerufen werden. Die `createProxy()` Methode bekommt einen ClassLoader übergeben, der Klassen aus dem Proxy-Jar-Archiv laden kann. Dieser ClassLoader muss genutzt werden, um den Proxy zu instanziieren. Alle weiteren für Jini-Dienste notwendigen Einstellungen werden von der Laufzeitbibliothek und den zugehörigen Werkzeugen durchgeführt.

Mit Hilfe der Laufzeitumgebung sollen natürlich auch Dienste verwaltet werden können, die nicht in Hinblick auf die Laufzeitumgebung entwickelt worden sind. Unter diese Dienste fallen zum Beispiel die Jini Infrastruktur Dienste, wie Lookupdienst, JavaSpaces-Dienst, Transaktionsdienst sowie alle Dienste, die von Dritten entwickelt und benutzt werden. Um diese Dienste trotzdem in die Laufzeitumgebung einbinden zu können, kann auch für diese Dienste die vorgestellte FrameWork-Schnittstelle durch ein zusätzliches Objekt implementiert werden.

5.4 Monitor

Monitore sind zuständig für die Überwachung der Verfügbarkeit der Dienste im System. Die Informationen über die Verteilungsbeschreibungen sind in der, in Abschnitt 5.1 beschriebenen, verteilten Datenhaltung gespeichert. Die Monitore sind dafür verantwortlich, dass diese Verteilungsbeschreibungen immer realisiert

sind, also zum Beispiel die in den Verteilungsbeschreibungen enthaltenen Dienste gestartet bzw. beendet werden.

Für jede Komponente muss sich ein Monitor als Verantwortlicher in das in Abschnitt 5.5 beschriebene Verantwortungsdatensystem eintragen. Diese Eintragung verläuft analog zum Prinzip der Mietdauern in Jini. Das bedeutet für die jeweiligen verantwortlichen Monitore, dass sie periodisch ihre Eintragung erneuern müssen. Ist eine Eintragung abgelaufen, das heißt der Monitor hat sich zu lange nicht mehr gemeldet, kann und muss ein anderer Monitor die Verantwortung für diese Komponente übernehmen. Durch diese Technik ist gewährleistet, dass, falls ein Monitor ausfällt, nach einer kurzen, vorher definierten Zeitdauer ein anderer Monitor die Aufgabe übernimmt. Dieser neue Monitor muss dann in den Verantwortungsdaten der Dienste, die noch existieren und in denen der alte Monitor referenziert wird, sich selber eintragen. Er übernimmt die Verantwortung für diese Dienste.

Ein für eine Kompositionskomponente, also eine aus mehreren Diensten gebildete Komponente, verantwortlicher Monitor muss jederzeit gewährleisten, dass die einzelnen Bestandteile der Komponente gestartet sind. Für jeden nicht gestarteten Dienst wählt der Monitor zufällig einen passenden Knotendienst aus und gibt dem Knotendienst einen Instanziierungsauftrag für den neu zu startenden Dienst. Falls ein Knoten ausfällt, startet der Monitor innerhalb kürzester Zeit nach Bekanntwerden (wenn der Jini Lookupdienst den Proxy entfernt und alle Interessenten die Benachrichtigung darüber erhalten) die betroffenen Dienste auf anderen Knoten neu. Da diese Ereignisse verloren gehen können, muss der Monitor periodisch überprüfen, welche Dienste in den Lookupdiensten noch enthalten sind.

Prinzipiell ist der Monitor ein aktives System, welches alle Aufgaben selbsttätig ausführt. Der Monitor könnte auch als normale Applikation (also nicht als Dienst) realisiert werden, aber um bestimmte Information vom Monitor zu erhalten, zum Beispiel auf welchem Knoten der Monitor einen bestimmten Dienst gestartet hat, bietet sich die Implementierung als Dienst an.

Backend

Das Backend ist für die eigentliche Arbeit zuständig. Hier werden die Datenstrukturen verwaltet und hier befinden sich die Listener für den Empfang diverser Nachrichten der Jini-Lookupdienste.

Es gibt zwei Bereiche im Backend. Der erste Teil ist das RMI-Backend, welches die oben angesprochenen Informationen liefert. Der zweite Bereich beinhaltet die eigentliche Implementierung durch eine zeitpunkt- und taskbasierte Verhaltensstruktur. Zu bestimmten Zeitpunkten werden bestimmte Tasks ausgeführt, die selber andere Tasks für einen bestimmten Zeitpunkt eintragen können. Diese Tasks werden zu den bestimmten Zeitpunkten nacheinander ausgeführt und innerhalb eines Threads abgearbeitet.

Es gibt sechs verschiedene Tasks, deren Zusammenwirken die Funktion des Monitors erfüllen. Die erste Funktion ist die Suche nach Komponenten in der Verteilungsbeschreibung, für die sich entweder kein Monitor verantwortlich zeigt oder deren verantwortlicher Monitor seine Mietdauer überschritten hat. Wird eine Komponente gefunden, für welche die vorherigen Bedingungen gelten, trägt sich der Monitor in der verteilten Datenhaltung als Verantwortlicher ein und es werden die folgenden zusätzlichen Tasks gestartet. Ein neu gestarteter Task verlängert periodisch die Miete für die Komponente in dem Verantwortungsdatensystem. Ein zweiter neu gestarteter Task überprüft periodisch für alle Dienste, aus der die Komponente besteht, ob diese an dem passenden Lookupdienst angemeldet sind. Für alle nicht angemeldeten Dienste wird ein weiterer Task gestartet, der für diese, basierend auf der Verteilungsbeschreibung, einen passenden Knoten sucht und den Dienst über den Knotendienst auf diesem Knoten startet. Ein weiterer Task überprüft periodisch, ob die Komponente noch in den Verteilungsbeschreibungen enthalten ist und der Monitor noch für sie verantwortlich ist. Wenn nein, werden alle diese Komponente betreffenden Tasks aus der Taskstruktur entfernt. Des Weiteren gibt es einen Task, der einen Dienst, der auf einem Knoten gestartet wurde, wieder beendet.

Koordination

Es wird vorausgesetzt, dass sich ein Monitor nur dann als Verantwortlicher für eine Komponente einträgt, wenn bisher für die Komponente kein Monitor als Verantwortlicher eingetragen ist oder die Mietdauer eines verantwortlichen Monitors bereits abgelaufen ist.

Problematisch kann die Situation sein, wenn ein Monitor kurzzeitig vom Netz getrennt wurde und zwischenzeitlich ein neuer Monitor dessen Aufgaben übernommen hat. Wenn beide Monitore nun die Aufgaben wahrnehmen, kann es passieren, dass bei einem ausgefallenen Dienst, von jedem Monitor ein Ersatzdienst gestartet wird. Aus diesem Grund geht ein Monitor, der vom Netz getrennt ist und deshalb seine Miete nicht verlängern kann, davon aus, dass ein anderer Monitor bereits seine Aufgaben übernommen hat und er deshalb seine Aufgaben nicht mehr wahrnehmen darf.

Zusätzlich überprüft ein Monitor von Zeit zu Zeit selber, ob er noch verantwortlich für eine Komponente ist, und wenn nicht, entfernt er diese Komponente aus seinen lokalen Datenstrukturen.

Ein Problem besteht darin, wenn Dienste nach einer Netzwerkpartition bereits auf anderen Knoten neu gestartet wurden und nach Behebung der Partition sich wieder am Lookupdienst eintragen wollen. In diesem Fall werden die alten Dienste, wenn möglich, beendet. Der, laut Verantwortungsdatensystem, für diese Komponente verantwortliche Monitor ist dann für das Beenden der Dienste zuständig. Allerdings ist es in diesem Fall nicht möglich, definitiv zu garantieren, dass alle alten Dienste beendet wurden. Es wird nur garantiert, dass jeder Dienst zumindestens einmal im System vorhanden ist.

5.5 Verteilte Speicherung der Verantwortungsdaten

Die im vorherigen Abschnitt erläuterten Verantwortungsdaten der Monitore sowie die zugehörigen Mietdauern müssen gespeichert werden. Da diese Daten in jeder Situation, vor allem auch im Fehlerfall, gelesen und geschrieben werden müssen, kann kein so starkes Konsistenzmodell wie in der Speicherung der Planungsdaten aus Abschnitt 5.1 genutzt werden. Das Konzept der zeitlich beschränkten Verantwortungsübernahme für eine Komponente durch einen Monitor und die daraus folgende Selbstheilung des Systems erlaubt es, ein schwächeres Konsistenzmodell zu nutzen.

In Abschnitt 2.3.1 wurde mit *eventueller Konsistenz* ein Modell beschrieben, welches sehr wenig Restriktionen enthält. Dieses Modell gewährleistet, dass über einen gewissen Zeitraum eine Konsistenz entsteht. Im Folgenden wird ein Verfahren vorgestellt, welches die Erreichung einer eventuellen Konsistenz gewährleistet.

Das System baut auf mehreren Backends auf mehreren Knoten des Systems auf, die sich per Multicast-Nachrichten (siehe [Wyb90]) verständigen. Mittels dieser Multicast-Nachrichten werden einerseits die verteilten Daten geändert sowie auch die Daten gesammelt, wenn andere Dienste Informationen benötigen. Da Multicast-Nachrichten von allen beteiligten Parteien empfangen werden, ist eine aufwendige 1-zu-n-Kommunikation, wie zum Beispiel durch viele 1-zu-1-Verbindungen mittels RMI, nicht nötig. Außerdem können alle Beteiligten die Kommunikation mithören und dadurch Entscheidungen für ihre lokalen Daten treffen. Jeder Knoten hat verschiedene Daten gespeichert, die verschiedene Komponenten betreffen. In diesen Daten wird gespeichert, welcher Monitor für die jeweilige Komponente zuständig ist und wie lange seine Mietdauer ist.

Um Probleme mit nicht synchron laufenden Uhren der Knoten zu vermeiden, werden die Mietdauern in relativer Zeit in den einzelnen Backends gespeichert. Wenn nach dieser Mietdauer gefragt wird, wird als Antwort die relative Zeit zwischen der aktuellen Zeit und der Zeit, als die letzte Verlängerung eingetragen wurde, zurückgegeben. Durch die Nutzung der relativen Zeit sind die Zeiten auf den unterschiedlichen Knoten nicht von den lokalen Zeiten der Knoten abhängig, sondern unterscheiden sich nur noch durch die unterschiedliche Transportzeit der Nachrichten sowie die unterschiedliche Dauer der Verarbeitung der Daten auf den Knoten. Dieser Unterschied ist in lokalen Netzen typischerweise kleiner als der Unterschied der verschiedenen lokalen Zeiten der Knoten.

Grundlegend für die Koordination ist ein Abstimmungsverfahren. Für dieses Verfahren werden die Restriktionen insoweit aufgeweicht, dass in bestimmten Fällen Daten verloren gehen bzw. nach Netzwerkseparierungen überschrieben werden können. Durch die speziellen Verhaltensweisen der Monitore und das Konzept der Mietdauern kann diese geringere Konsistenz gegenüber der in Abschnitt 5.1 vorgestellten Lösung genutzt werden und damit eine höhere Verfügbarkeit

erreicht werden.

Das Abstimmungsverfahren basiert auf relativer Mehrheit. Aus den möglicherweise vielen verschiedenen Daten im System wird ein gemeinsames Datum bestimmt. Bei einem Lesezugriff werden die Daten aller erreichbaren Teilsysteme eingeholt. Das Datum, welches auf der Mehrzahl der Systeme gespeichert ist, wird den anderen Daten vorgezogen. Bei Gleichheit der abgegebenen Stimmen gewinnt das Datum in der Abstimmung, welches den kleineren Hashwert des Inhalts hat; dieses Verhalten dient dazu eine Entscheidung herbeizuführen. Die Entscheidung durch Abstimmung wird nicht nur vom Anfragenden, sondern auch von jedem der Teilsysteme durchgeführt. Dies ist möglich, da durch die Multicast-Nachrichten alle Systeme die Nachrichten erhalten. Nachdem jedes System diese Entscheidung durchgeführt hat, ändert es, wenn nötig, seine lokalen Daten. Durch dieses passive Mithören der Daten und die lokale Abstimmung gleichen sich die Daten der Teilsysteme an. Da Multicast-Nachrichten nicht garantiert jedes Teilsystem erreichen, kann es passieren, dass ein Teilsystem bei einer lokalen Abstimmung einmal zu einem anderen Ergebnis kommt, beim nächsten Lesezugriff werden diese Daten jedoch durch die nächste Abstimmung wieder überschrieben. Ein Schreibzugriff wird von allen Teilsystemen, welche die Nachricht empfangen, direkt ausgeführt. Durch das Überschreiben der lokalen Daten wird nach Behebung eines Fehlerfalls eine Stabilisierung des Systems erreicht, so dass alle Teilsysteme wieder dieselben Daten beinhalten.

Nachrichten

Eine Nachricht besteht aus einem Header sowie einem optionalen Datenteil. Der Header enthält den Typ der Nachricht sowie eine SessionID. Die SessionID wird zu Beginn einer Sitzung, die mit einer GETDATA-Nachricht beginnt, zufällig erzeugt. Um eine mögliche Überschneidung der IDs zu vermeiden, wird die ID aus den Ausgangsdaten IP-Adresse (128bit für IPv6 siehe [Hin96]), Zeit (32bit) und zufällige Bits (32bit) zuzüglich des Namens der Daten gebildet. Die ID hat eine Länge von 192 Bits plus die Länge des Namens. Die Erzeugung dieser SessionID lehnt sich an die Erzeugung der ServiceID in Jini ([AOS+99] und [Sun01a, S.118]) an. Diese SessionID dient dazu, Nachrichten und Antworten auf diese Nachricht zu verbinden sowie über den Namen verschiedene Daten anzusprechen. Eine Antwort auf eine Nachricht muss dieselbe SessionID haben wie die originale Nachricht. Eine Nachricht darf eine maximale Größe von 64kb nicht überschreiten. Diese Restriktion entsteht durch das verwendete UDP-Protokoll, welches nur Pakete mit einer Maximalgröße von 64kb unterstützt. Um diese Restriktion zu umgehen, müssten mehrere Pakete geschickt werden. Das JavaGroup-Toolkit [Ban98] ermöglicht eine zuverlässige Multicast-Kommunikation, die größere Pakete automatisch in passende Pakete unterteilt und beim Empfänger automatisch zusammensetzt. In dieser Arbeit erreichen die Nachrichten allerdings keine Größe von über 64kb, so dass dieses JavaGroup-Toolkit nicht genutzt wird.

Es gibt folgende Nachrichtentypen:

GETDATA: Mit Hilfe dieser Nachricht werden die Teilsysteme aufgefordert, ihre lokalen Daten zurück zu senden. Als Antwort auf diese Nachricht schicken die Backends die Nachricht LOCALDATA. Bei dieser Nachricht ist der Datenteil leer.

LOCALDATA: Diese Nachricht wird von den Backends als Antwort auf eine GETDATA-Nachricht versandt. Im Datenteil werden die lokalen Daten des Backends versandt. Die Daten werden in serialisierter Form verschickt, das bedeutet die Objektstrukturen werden durch den in [Sun98b] spezifizierten Algorithmus in eine speicherbare und über das Netz sendbare Form transformiert. Durch Deserialisierung können dann die Objektstrukturen wiederhergestellt werden.

NEWDATA: Diese Nachricht wird für den Schreibzugriff versandt. Im Datenteil wird eine neue Version der Daten versandt. Der Datenteil wird von den Teilsystemen unter dem in der SessionID enthaltenen Namen gespeichert.

Wie in Abschnitt 3.3 beschrieben ist ein großer Vorteil von Jini, dass ein Proxy nicht fest an ein Backend gebunden ist, sondern flexibel arbeiten kann. Diese Möglichkeit wird im Folgenden genutzt, indem der Proxy, wie oben beschrieben, mit allen Backends kommuniziert. Wenn nun Änderungen an den Daten vorgenommen werden sollen, wird die verschickte Multicast-Nachricht von allen Backends erhalten, welche die vom Proxy gesendeten Daten übernehmen. Wenn der Proxy einen Lesezugriff auf die Daten durchführt, schickt er per Multicast-Nachricht allen Backends eine GETDATA-Nachricht. Als Antwort schicken diese nun eine Multicast-Nachricht mit ihren Kopien der Daten und der Proxy entscheidet anhand des oben genannten Abstimmungsmodus, welche Daten an den Anfragenden zurückgeliefert werden.

Wie bereits oben beschrieben erhalten alle Backends alle Nachrichten der anderen Backends und des Smartproxies. Bei Änderungsnachrichten (NEWDATA) der Daten durch den Proxy übernimmt jedes Backend die in der Nachricht enthaltenen neuen Daten. Bei Abfrage der Daten durch den Smartproxy (GETDATA) schicken alle Backends eine Antwortnachricht (LOCALDATA) mit ihrer lokalen Version der Daten. Da auch diese Nachricht von jedem Backend empfangen wird, kann nun jedes Backend aufgrund der eingegangenen Antworten eine Abstimmung durchführen. Die von der Mehrheit der Backends abgegebenen Stimmen übernimmt das Backend nach dem oben beschriebenen Abstimmungsverfahren dann als neue Version seiner aktuellen Daten.

Abbildung 5.11 zeigt zwei Statecharts, die das grobe Verhalten des Backends in Reaktion auf diverse Nachrichten abbilden. Die NEWDATA-Nachricht wird direkt vom Backend abgearbeitet. Nach Eingang einer GETDATA-Nachricht wird für die folgenden Eingänge der LOCALDATA-Nachrichten und der nachfolgenden

Abstimmung eine neue Sitzung in einem neuen Thread gestartet, der nebenläufig zum restlichen System das rechte Statechart abarbeitet. Dieses Statechart zeigt, dass in dieser Sitzung nach Ablauf von 10 Sekunden ohne einen neuen Eingang einer LOCALDATA-Nachricht die Abstimmung durchgeführt wird. Diese starre Grenze kann verbessert werden, indem im Verlauf von mehreren Abstimmungen die durchschnittliche Zeit approximiert wird, bis alle Nachrichten beim Empfänger angekommen sind. Man kann dann die Wartezeit an diese durchschnittliche Zeit anpassen. Wenn eine Stimme erst nach der Abstimmung eintrifft, kann die Wartezeit nach oben verlängert werden, um auf eine eventuell höhere Auslastung des Netzes oder der Backends zu reagieren.

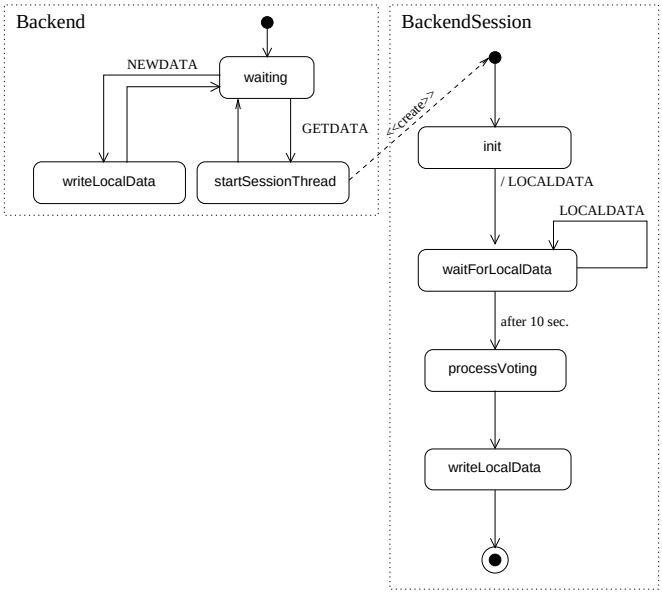

Abbildung 5.11: Statecharts für das Verhalten der Backends

Beispiel und Analyse

Für ein verteiltes System ist das Verhalten im Fehlerfall sehr wichtig. Im Folgenden wird das Verhalten beispielhaft in verschiedenen Situationen dargestellt sowie die Änderungen der lokalen Daten der Backends nachvollzogen.

In den folgenden Beispielen werden die Backends mit dem Smartproxy abgebildet. Ausgangssituation ist ein lokales Netz mit sechs Knoten, die in zwei Subnetzen à drei Knoten organisiert sind.

Um die verschiedenen Daten in den Backends der einzelnen Knoten zu unterscheiden, wird im Folgenden von Versionsnummern der enthaltenen Daten gesprochen. Durch diese Nummern werden keine Ordnungsrelationen zwischen diesen Daten beschrieben, sondern sie dienen alleine der Unterscheidung der Daten.

In Abbildung 5.12 werden die beiden Subnetze dargestellt. Die Komponenten innerhalb der Subnetze visualisieren die Backends der einzelnen Dienste. Oben im Rahmen wird die Nummer des Knotens angegeben; unten steht die aktuelle Version der Daten des Knotens. Die Kommunikation zwischen den beiden Subnetzen läuft nur über die dick gezeichnete Netzwerkverbindung.

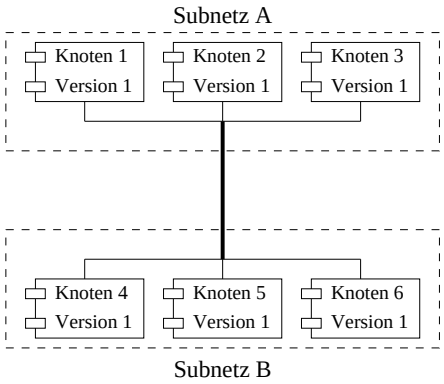

Abbildung 5.12: Ausgangssituation

Der Ausfall eines Knotens hat keine großen Veränderungen der Situation zur Folge. Nach dem Ausfall des Knotens 4 ergeben sich aus dem normalen Ablauf des Systems zwei Änderungen, so dass die restlichen Knoten die Daten in Version 3 gespeichert haben. Nach Reboot und Neustart des Backends auf Knoten 4 stellt sich die Situation wie in Abbildung 5.13 dar. Die Ellipse enthält die Versionsnummern der über die Multicast-Nachrichten verschickten Daten. Sie visualisiert die Tatsache, dass Multicast-Nachrichten von allen Beteiligten erhalten werden.

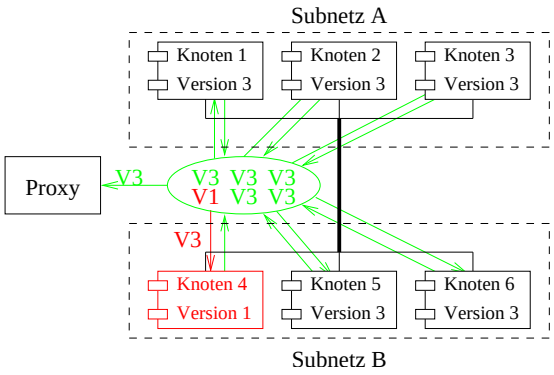

Abbildung 5.13: Situation nach dem Reboot von Knoten 4

Das Backend auf Knoten 4 hat die bei sich vor dem Ausfall gespeicherten Daten in Version 1 geladen. Bei der nächsten Abfrage von Daten durch den Smartproxy schickt das Backend seine Version der Daten zurück. Von allen anderen Backends bekommt das Backend nun allerdings Nachrichten mit der Version 3

der Daten. Jedes Backend entscheidet aufgrund der Antworten der anderen Knoten, ob seine lokalen Daten noch korrekt sind. Das Backend auf Knoten 4 erkennt, dass die Version 3 von fünf Backends favorisiert wird, während die Version 1 nur von ihm selber bevorzugt wird. Es überschreibt seine lokalen Daten daraufhin mit der Version 3. Als Ergebnis haben nun alle Backends wieder dieselben Daten und der Proxy liefert auf Grund der Abstimmung die Daten in der Version 3 an den anfragenden Nutzer zurück.

Bei einem Ausfall der Netzwerkverbindung zwischen den beiden Subnetzen entwickeln sich die Daten der Backends in den beiden Subnetzen getrennt voneinander weiter. Im Subnetz A wurden die Daten durch Änderung der Monitoreinträge auf die Version 6 geändert, im Subnetz B auf die Version 8. Es gilt zusätzlich: $CRC(V6) < CRC(V8)$. Nach der Behebung der Netzwerkpartition stellt sich die folgende Situation in Abbildung 5.14 dar.

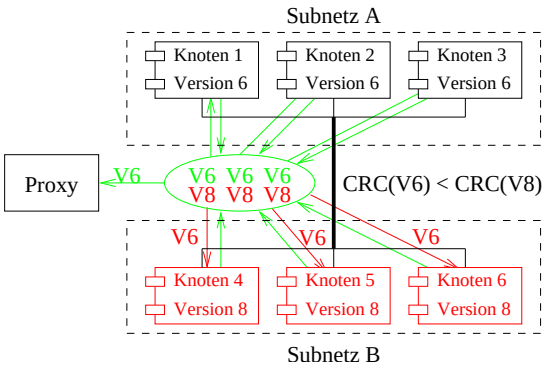

Abbildung 5.14: Situation nach Behebung der Netzwerkpartition

Wenn der Proxy nun die Backends nach den Daten fragt, ergibt sich ein Gleichstand zwischen den Stimmen für Version 6 und Version 8. In diesem speziellen Fall entscheidet der niedrigere CRC-Wert der Daten von Version 6, dass bei der Abstimmung die Version 6 vorgezogen wird. Die Knoten 4, 5 und 6 übernehmen die Version 6 der Daten. Als Ergebnis haben nun alle Backends wieder dieselben Daten.

Implementierung

Abbildung 5.15 zeigt das Klassendiagramm der Implementierung. Die Klassen `Backend` und `BackendSession` sind bereits aus Abbildung 5.11 bekannt. Verschiedene Sitzungen werden über die oben beschriebene Sitzungsnummer (SessionID) unterschieden. Die durch LOCALDATA-Nachrichten eingehenden Daten werden in `LocalData` Objekten gespeichert. Die Klasse `StreamMessageMultiplexer` ist für die Unterscheidung der Nachricht nach ihrem Typ und die Verteilung der Nachrichten über Aufrufe der passenden Methoden der Klasse `Backend` zuständig. Die Klasse `MulticastMessageConnector` ist für den Versand und den Empfang

der Nachrichten als Multicast-Pakete zuständig. Diese Klasse läuft als Thread nebenläufig und verteilt die Nachricht an alle Interessenten, von denen die Klasse `StreamMessageMultiplexer` einer ist. Durch diese Trennung von konkreter Netzwerkkommunikation (`MulticastMessageConnector`) und Nachrichtenverteilung auf höherer Abstraktionsebene (`StreamMessageMultiplexer`) ist es einfach möglich, die konkrete Netzwerkkommunikation zu ändern, ohne dass der restliche Teil des Systems davon betroffen ist. Über die Schnittstelle `DataFilter` können nach Eingang der Daten (über NEWDATA) und vor Ausgang der Daten (über LOCALDATA) die Daten bearbeitet und evtl. geändert werden. Diese Möglichkeit wird genutzt, um die Mietdauern in relative Zeiten zu konvertieren. Schlussendlich werden die Daten durch die Klasse `DummyBackendStorage` nach jedem Schreibzugriff in einer Datei im lokalen Dateisystem gespeichert. Diese Klasse wird über die Schnittstelle `BackendStorageInterface` angesprochen, so dass ein Austausch der Speicherung möglich ist. Für die Speicherung der Daten des Backends ist keine Datenbank bzw. keine Transaktionseigenschaft notwendig, da bei einer fehlerhaften Speicherung die falschen Daten bei der nächsten Abstimmung überstimmt werden.

5.6 Fehlerbehandlung

Während in Abbildung 5.4 das Verhalten der verteilten Datenbasis der Planungsdiagramme im Fehlerfall und in Abschnitt 5.5 das Verhalten der verteilten Datenhaltung der Verantwortungsdaten im Fehlerfall und deren Inhalt Gegenstand der Betrachtung waren, werden ausgehend von obigen Erkenntnissen die Auswirkungen der verschiedenen Fehlerfälle auf das Gesamtsystem sowie die Arbeit der Monitore betrachtet. Wie in Abschnitt 5.5 vorgestellt, wird auch in diesem Abschnitt ein lokales Netz mit zwei Subnetzen à 3 Knoten benutzt. Die Knoten 1 bis 3 sind im Subnetz A, die Knoten 4 bis 6 im Subnetz B. Zwischen diesen beiden Subnetzen gibt es eine Netzwerkverbindung.

Wie in Abbildung 5.16 dargestellt, wird auf jedem Knoten ein Monitor und ein Applikationsdienst des in Abschnitt 1.5 dargestellten Beispiels ausgeführt. Die auf jedem Knoten laufenden Knotendienste und die Dienste zur Speicherung der Verantwortungs- und Verteilungsdaten wurden zum Zweck der Übersichtlichkeit nicht in die Abbildungen aufgenommen. Der Monitor auf Knoten 2 ist verantwortlich für die Kompositionskomponente, die alle sechs Applikationsdienste enthält. In der verteilten Datenhaltung ist die aktuelle Konfiguration, wie in Abbildung 5.16 abgebildet, gespeichert.

5.6.1 Ausfall eines Knotens

Bei Ausfall eines Knotens gilt es zwei Situationen zu unterscheiden: Ist ein Monitor, der die Verantwortung für die Komponente übernommen hat, auf dem

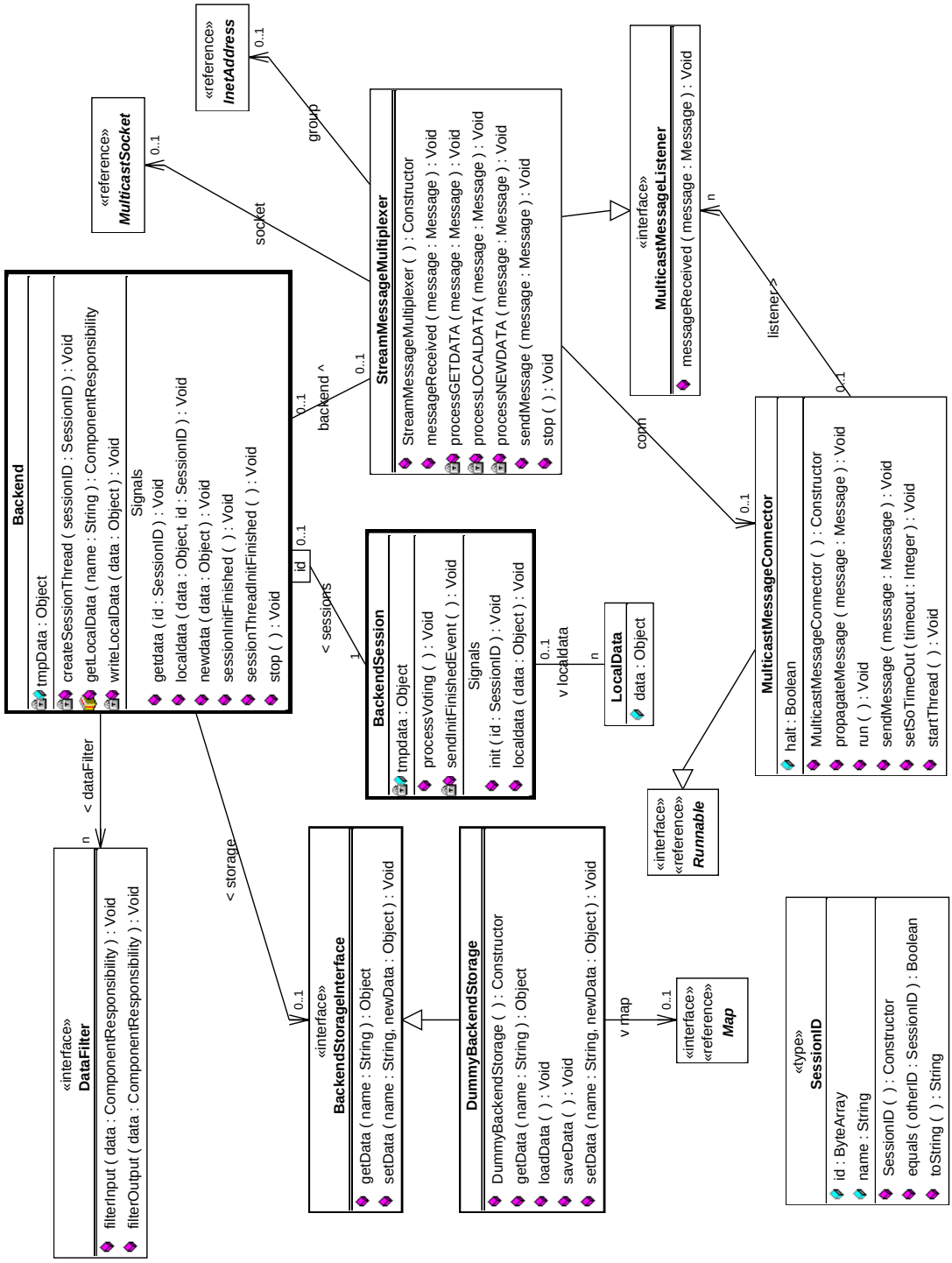

Abbildung 5.15: Klassendiagramm des Speichersystems der Verantwortlichkeiten

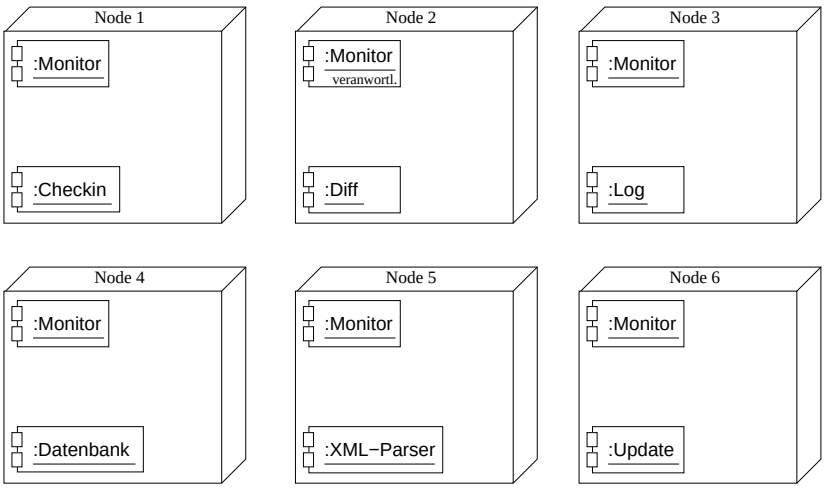

Abbildung 5.16: Situation des Netzwerks vor Auftritt der Fehler

ausgefallenen Knoten ausgeführt worden, oder nicht.

Der zweite Fall ist der Einfachere. Als Beispiel für diesen Fall kann man den Ausfall von Knoten 4 nennen. Der für die Komponente verantwortliche Monitor auf Knoten 2 erkennt, dass der Datenbankdienst nicht mehr im System vorhanden ist. Der Monitor startet diesen Dienst nun auf einem anderen Knoten (zum Beispiel Knoten 5) neu. Nach einem Neustart des Knotens 4 und dem damit zusammenhängenden Start der diversen Laufzeitumgebungsdienste ergibt sich die Situation in Abbildung 5.17.

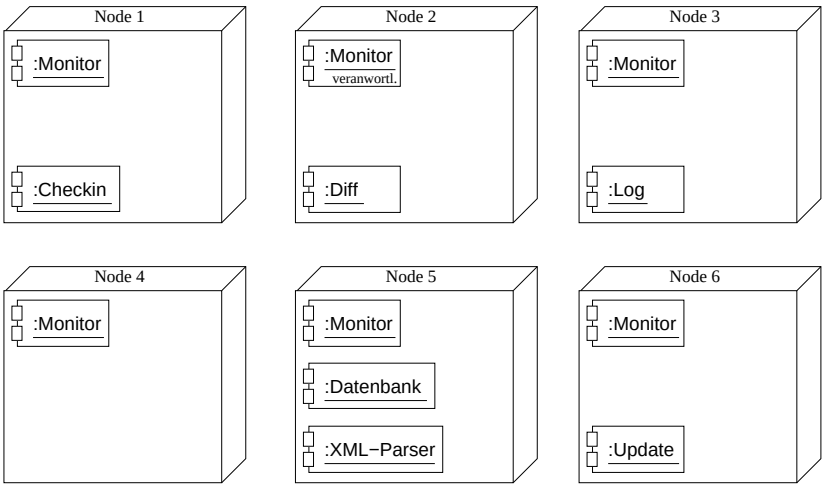

Abbildung 5.17: Situation nach Ausfall von Knoten 4

Der erste Fall ist etwas schwieriger als der Zweite. Hier fällt der Knoten aus, auf dem der verantwortliche Monitor ausgeführt wird. In der dargestellten Situation betrifft dieser Fehler den Knoten 2. Nach dem Ausfall von Knoten 2 existiert

der für die Komponente verantwortliche Monitor sowie ein Dienst („Diff“) nicht mehr. Wie in Abschnitt 5.4 dargestellt, übernimmt nach Ablauf der Mietdauer ein anderer Monitor (zum Beispiel derjenige auf Knoten 5) die Verantwortung für das Subsystem. Dieser Monitor erkennt nun, dass der „Diff“-Dienst nicht mehr vorhanden ist und startet ihn zum Beispiel auf Knoten 1 neu. Nach einem Reboot des Knotens 2 ergibt sich die Situation in Abbildung 5.18.

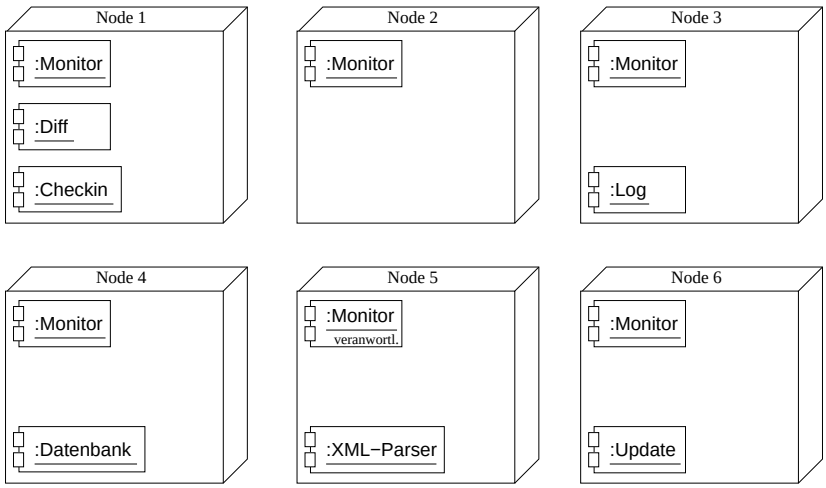

Abbildung 5.18: Situation nach Ausfall von Knoten 2

5.6.2 Netzwerkpartition

Bei Teilausfällen von Netzwerkverbindungen finden größere Veränderungen statt. Als Beispiel wird das Verhalten bei einem Problem der Netzwerkverbindung zwischen den Subnetzen A und B gezeigt. Nach Ausfall dieser Netzwerkverbindung entwickeln sich die beiden Subnetze unterschiedlich weiter.

In Subnetz A erkennt der weiterhin verantwortliche Monitor, dass die Dienste „XML-Parser“, „Datenbank“, „Update“ nun nicht mehr im verkleinerten Netz existieren. Als Reaktion darauf startet er diese Dienste auf den Knoten des Subnetzes A neu. Im Subnetz B stellt sich die Situation unterschiedlich dar. In diesem Subnetz ist der verantwortliche Monitor durch die Netzwerkpartition nicht mehr erreichbar. Da der verantwortliche Monitor aus dem Subnetz A seine Mietdauer im Verantwortungsdatensystem aus Subnetz B nicht mehr verlängern kann, übernimmt ein Monitor aus diesem Subnetz nach Ablauf der Mietdauer die Verantwortung. Dieser Monitor findet nun die Dienste „Checkin“, „Diff“, „Log“ nicht und startet sie deshalb auf den Knoten des Subnetzes B neu. Die Verantwortungsdaten der einzelnen Subnetze enthalten nun unterschiedliche Daten, welcher Monitor für die Komponente verantwortlich ist. Abbildung 5.19 zeigt die Situation nach Ablauf der beschriebenen Aktionen.

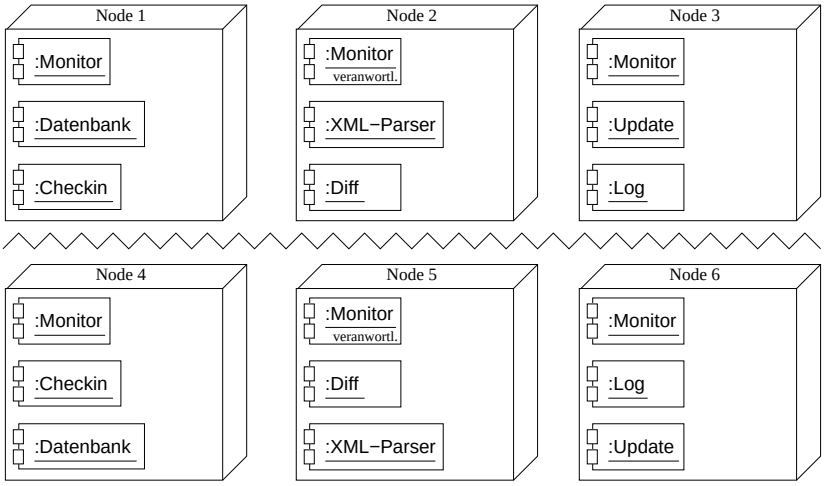

Abbildung 5.19: Netzwerkpartition

Nach erfolgreicher Behebung der Netzwerkpartition synchronisieren sich die Verantwortungsdatendienste wie in Abschnitt 5.5 dargestellt. Da die beiden Subnetze gleich groß sind, entscheidet der geringere CRC der Daten (für dieses Beispiel werden die Daten des Subnetzes A gewählt). Der Monitor auf Knoten 5 hat nun die Verantwortung für die Komponente verloren, als Reaktion darauf beendet er die Dienste, für die er während der Netzwerkpartition verantwortlich war und die öfter als nötig im Netz existieren. Nach diesen Abläufen ist die in Abbildung 5.20 gezeigte Situation erreicht, die Netzwerkpartition erfolgreich behoben und das System wieder konsistent.

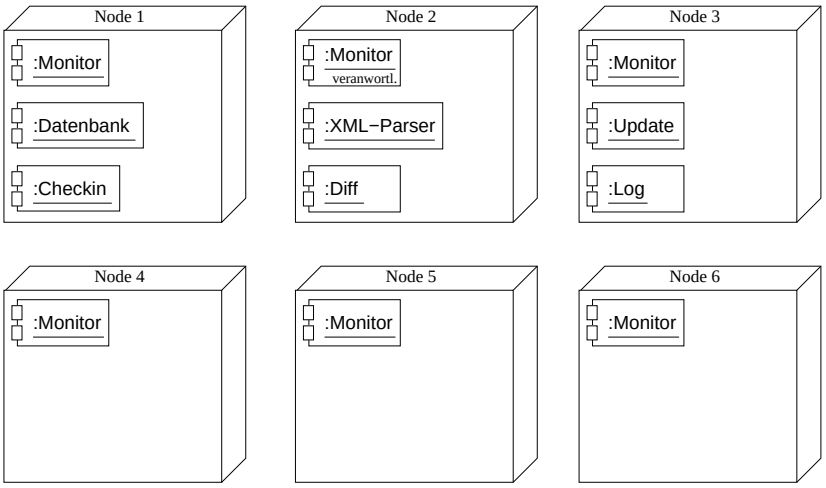

Abbildung 5.20: Situation nach Behebung der Netzwerkpartition

5.7 TYCS Cycleguard

Der TYCS-Ansatz [Gie01b] bietet die Möglichkeit, die Komposition von Softwarekomponenten zu überprüfen und potentiell gefährliche zyklische Abhängigkeiten zwischen Komponenten zu verhindern. In Abschnitt 2.1 wurde die Problematik der Abhängigkeiten zwischen Komponenten bereits genauer beschrieben. Giese definiert in [Gie01a, S.16] ein offenes System als ein System, welches Verbindungen zu unkontrollierbaren oder sogar zur Designzeit nicht erwarteten Elementen hat. In diesem Zusammenhang bilden die in Abschnitt 2.1 skizzierten eingekauften Komponenten ein offenes System.

Durch die Abhängigkeiten zwischen den Komponenten kann es passieren, dass bei Zusammenstellung der Komponenten zu einer Applikation alle Komponenten über ihre Abhängigkeiten zirkulär verbunden sind. Dies bedeutet, dass Operationen an einer Komponente über diese Abhängigkeiten am Ende wieder die Operation auf dieser Komponente aufrufen und so eine verteilte Endlosschleife bilden bzw. bei Synchronisation der Operationen ein Deadlock resultiert. In hierarchischen Applikationen können, wie von Lam und Shankar in [LS94] vorgeschlagen, diese gefährlichen Verbindungen vermieden werden, indem die Applikation in verschiedene Schichten unterteilt wird, in denen nur Aufrufe von Methoden auf Komponenten in unteren Schichten erlaubt sind. Da bei eingekauften Komponenten üblicherweise allerdings kein Wissen darüber existiert, unter welchen Bedingungen sie andere Komponenten nutzen, können solche Schleifen nicht im voraus erkannt werden. Giese adressiert in [Gie01b] dieses Problem der zyklischen Abhängigkeiten. Seinem Vorschlag nach müssen die Komponenten explizit spezifizieren, welche ihrer Schnittstellen von welchen Schnittstellen anderer Komponenten abhängen. Aufgrund dieser Informationen besteht die Möglichkeit, potentiell gefährliche Verbindungen der Komponenten zu erkennen und zu verhindern.

Wenn die in Abschnitt 5.3 beschriebene Laufzeitbibliothek die einzelnen Komponenten zur Laufzeit verbindet, überprüft sie diese Verbindungen mit Hilfe des Cycleguard-Dienstes. Diese Überprüfung wird durch einen gerichteten, azyklischen Graphen geleistet, der aus den bereits bestehenden Verbindungen besteht. Diese Verbindungen sind einerseits die Beziehungen zwischen den Schnittstellen zweier Komponenten und andererseits die Abhängigkeiten zwischen angebotenen und benutzten Schnittstellen innerhalb einer Komponente. Wenn die zu überprüfende Verbindung beim Einfügen in den Graphen zu einem Zyklus führen würde, ist diese Verbindung potentiell gefährlich und es wird dem Knotendienst vorgeschlagen, diese Verbindung nicht durchzuführen.

Der Graph wird mit Hilfe der OpenJGraph-Bibliothek [Sal01] verwaltet. Dabei wird jeder Schnittstelle eine eindeutige Nummer zugewiesen. Diese Nummer wird im Graphen mit einem Knoten repräsentiert. Verbindungen zwischen den Schnittstellen werden durch die Kanten des Graphen repräsentiert. Nach Starten eines Dienstes wird eine Instanz der Klasse `CycleGuardIDEntry` erzeugt und nach

Setzen der Schnittstellen-Nummer in die Jini-Attribute des Dienstes eingefügt. Die OpenJGraph-Bibliothek bietet nicht nur Graphalgorithmen, sondern auch die Visualisierung der Graphen. Abbildung 5.21 zeigt den Abhängigkeitsgraphen während der Laufzeit.

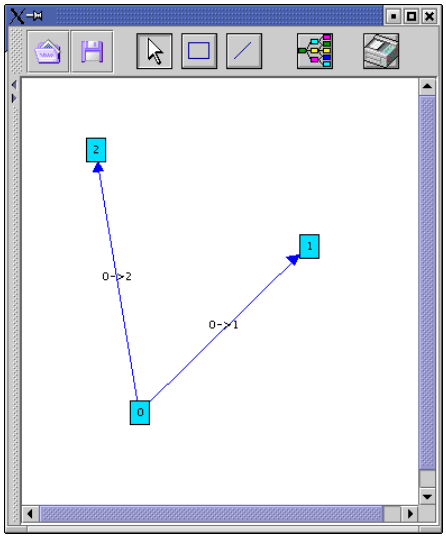

Abbildung 5.21: Graph der Komponentenverbindungen

Der Aufwand für eine Zyklensuche mittels einer Tiefensuche [CLR90, S.482] nach [CLR90, S.479] ist in einem gerichteten Graphen mit V Knoten und E Kanten $\Theta(E+V)$. Die Anzahl der Knoten ist durch die Anzahl der Schnittstellen beschränkt und die Anzahl der Kanten kann im schlechtesten Fall quadratisch mit der Anzahl der Schnittstellen wachsen, wenn jede Schnittstelle mit jeder anderen Schnittstelle verbunden ist, also $E = V^2$ gilt. Die bisher implementierte Lösung skaliert daher schlecht, da der Aufwand polynomial mit der Menge der Komponenten ansteigt.

Dieser Dienst ist für die Funktion der Laufzeitumgebung nicht unbedingt notwendig, deshalb wird er nur genutzt, wenn er gefunden wird. Da der Graph nur innerhalb einer Instanz dieses Dienstes existiert, darf nur eine einzige Instanz dieses Dienstes im System existieren, da es nur einen einzigen Graphen im System geben darf. Fehlertoleranz ist dadurch naturgemäß nicht gewährleistet, allerdings ist das System auch nach Ausfall dieses Dienstes weiter funktionsfähig. Alleine bei Überprüfung der Komponentenverbindung kann es passieren, dass durch den Verlust von Informationen potentiell gefährliche Verbindungen nicht erkannt werden. Allerdings werden keine korrekten Verbindungen fälschlicherweise als nicht korrekt erkannt. Der Dienst in der derzeitigen Realisierung ist daher nicht hoch verfügbar.

5.8 Administrationskonsole

Die Administrationskonsole ist kein Dienst, sondern eine normale Applikation. Dieses Werkzeug hat mehrere Aufgaben während des Betriebes der Komponenten im verteilten System. Die Applikation ist verantwortlich für das Bearbeiten der verteilten Datenhaltung der Verteilungsdaten (Laden der Informationen neuer Verteilungsbeschreibungen, Löschen alter Daten, etc.) und die Visualisierung der aktuellen Verteilung der Komponenten mittels der in Abschnitt 4.4 vorgestellten und in 4.4.2 angepassten Verteilungsdiagramme.

Abbildung 5.22 zeigt die Administrationskonsole, welche die aktuelle Situation der Verteilung der Komponenten aus dem Beispielszenario (Abschnitt 1.5) zu einem bestimmten Zeitpunkt im Netz darstellt. Im Netz sind vier verschiedene Rechenknoten für das verteilte System verfügbar. Alle diese Knoten benutzen Linux in der Version 2.4.17 respektive 2.4.18 als Betriebssystem und das JDK 1.3.1 der Firma Sun Microsystems. Auf dem Knoten ,,ginerva“ werden die Komponenten ,,XML-Parser“, ,,Database“ und ,,Diff“ ausgeführt. Die Komponenten ,,Checkin“ und ,,Log“ wurden vom Monitor auf dem Knoten ,,uther“ und die ,,Update“-Komponente auf dem Knoten ,,mordred“ gestartet. Der Knoten ,,tintagel“ wird aktuell nicht genutzt.

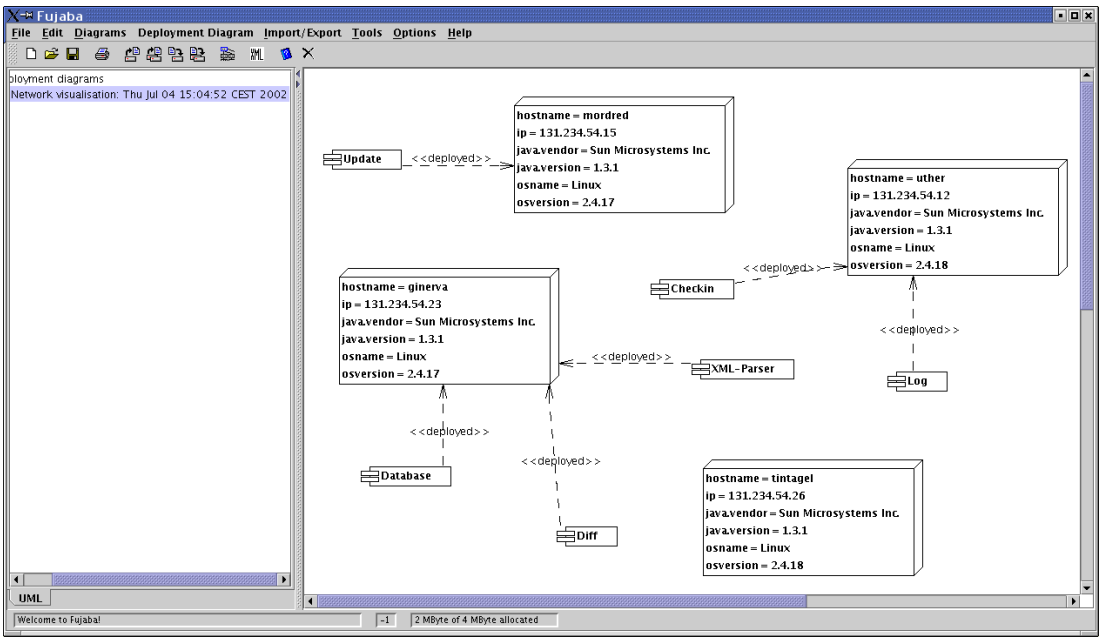

Abbildung 5.22: Visualisierung der aktuellen Situation im Netzwerk

5.9 Zusammenfassung

Die in diesem Kapitel vorgestellte Laufzeitumgebung ermöglicht den Betrieb von Komponenten in einem verteilten System. Die Laufzeitumgebung besteht aus mehreren Teilen, die durch ihre Zusammenarbeit den fehlertoleranten Betrieb der Komponenten gewährleisten. Auf jedem Rechenknoten des Systems ermöglicht ein Knotendienst Komponenten des Systems auf diesem Knoten zu starten. Die einzelnen Komponenten werden mittels einer Laufzeitbibliothek gestartet, die alle jini-relevanten Aufgaben übernimmt und so den Entwickler der Komponente von diesen Entwicklungsaufgaben entlastet. Mehrere Dienste speichern verteilt die Informationen aus den Komponenten- und Verteilungsdiagrammen. Die sequentielle Konsistenz dieser verteilt gespeicherten Daten wird mittels des Verfahrens der gewichteten Abstimmung erreicht. Für die Realisation der in den Verteilungsdiagrammen enthaltenen Spezifikation sind mehrere Monitore verantwortlich. Diese Monitore stellen sicher, dass auch bei Ausfall von Rechenknoten und bei einer Netzwerkpartition, alle Komponenten aus dem Verteilungsdiagramm im System verfügbar sind. Um Ausfälle von Monitoren tolerieren zu können, müssen diese Monitore sich periodisch in einem Verantwortungsdatensystem eintragen, so dass aus dem Ausbleiben dieser Eintragungen auf den Ausfall der Monitore geschlossen werden kann. Diese Verantwortungsdaten werden auch verteilt, allerdings mit einer schwächeren Konsistenz, gespeichert. Für die Überprüfung der Komponentenabhängigkeiten wird ein Überprüfungsdienst der erlaubten Kommunikationsverbindungen zwischen den einzelnen Komponenten vorgestellt. Eine Administrationskonsole erlaubt die Visualisierung der Situation des Systems und eine Änderung der verteilt gespeicherten Daten.

Durch diese Laufzeitumgebung werden die Ziele dieser Arbeit in Hinsicht auf Fehlertoleranz und Skalierbarkeit erreicht. Aufbauend auf die im vorherigen Kapitel dargestellte Unterstützung des Entwurfs, der Implementierung und der Planung des Betriebs ergibt sich die angestrebte durchgängige Unterstützung.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde eine durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen vorgestellt. Der gesamte Lebenszyklus der Komponenten wird mit der Unified Modelling Language unterstützt. Entwurf und Implementierung der einzelnen Komponenten wird mit Hilfe von Struktur- und Verhaltensdiagrammen spezifiziert. Aus diesen Diagrammen wird Java-Quelltext generiert, der dann kompiliert und für den Betrieb vorbereitet wird. Durch die Spezifikation der Applikation auf einer hohen Abstraktionsebene und der Generierung der Implementierung aus der Spezifikation wird eine verbesserte Wartbarkeit gegenüber manuell erstellten Implementierungen erreicht. Die einzelnen Komponenten werden dann mit Hilfe von Komponentendiagrammen zu Applikationen zusammengesetzt. In Verteilungsdiagrammen werden durch Bedingungen die Knoten spezifiziert, auf denen die einzelnen Komponenten im Betrieb ausgeführt werden können.

Mittels einer auf der Jini-Technologie basierenden Laufzeitumgebung wird die aus einzelnen Komponenten bestehende Applikation gemäß der Informationen aus den Verteilungsdiagrammen im System betrieben. Die genutzte Jini-Technologie erlaubt es die Eigenschaften verteilter Systeme explizit in das Design der Komponenten einzubeziehen. Die Laufzeitumgebung ist für die Fehlertoleranz des Systems verantwortlich. Die in dieser Arbeit vorgestellte Lösung nutzt, im Gegensatz zu dem in Abschnitt 3.3.2 beschriebenen RIO Projekt, keine zentralen Systeme, deren Verfügbarkeit für die Verfügbarkeit des Gesamtsystems essenziell wichtig ist. Des Weiteren wird die Integration der einzelnen Komponente in das Gesamtsystem durch eine Laufzeitbibliothek geleistet, um den Entwickler von den damit verbundenen Standardaufgaben zu entlasten.

6.2 Ausblick

Die Umsetzung der Komponenten- und Verteilungsdiagramme aus der UML-Spezifikation wurde auf die für diese Arbeit relevanten Bereiche beschränkt. Eine umfassende, über diese Arbeit hinausgehende Umsetzung der Spezifikation, zum Beispiel die Unterstützung der die Komponente implementierenden Artefakte in

Komponentendiagrammen, ist für die Nutzung als Dokumentationswerkzeug sehr wichtig. Eine weitere mögliche Erweiterung ist die Nutzung von speziellen Stereotypen mit geeigneten Icons für Komponenten und Rechenknoten in Verteilungsdiagrammen, um eine anschaulichere Ansicht anbieten zu können.

Eine Erweiterung der Laufzeitumgebung ist die Möglichkeit der gegenseitigen Überwachung der Monitore. Bei Ausfall eines Monitors erkennt ein anderer Monitor diese Situation und startet auf dem jeweiligen Knoten einen neuen Monitor. Diese Fähigkeit könnte auch zum initialen Starten (Bootstrapping) des Systems genutzt werden. Außerdem ist es denkbar, dass auf einem Knoten mehrere Knotendienste parallel laufen, so dass auch bei Ausfall eines Knotendienstes der Knoten im System weiter genutzt werden kann. Die Laufzeitumgebung kann auch durch eine fehlertolerante Erweiterung des Cycleguard-Dienstes verbessert werden. Hierbei muss der innerhalb des Dienstes verwaltete Graph in mehreren Diensten im System verteilt gespeichert werden.

Des Weiteren ist es denkbar, eine Komponente zur Lastverteilung im System zu entwickeln, so dass die einzelnen Komponenten im System verschoben werden können, um die Auslastungsdifferenz zwischen den einzelnen Knoten des Systems zu minimieren und zu verhindern, dass bestimmte Systeme die Geschwindigkeit des Gesamtsystems massiv beeinflussen. Außerdem sollte die Möglichkeit bestehen, mehrere Laufzeitumgebungen parallel im Netz zu benutzen, die sich gegenseitig nicht stören oder beeinflussen. Dies ist in Teilen im Design des Systems bereits berücksichtigt. Es muss allerdings eine Konfigurationsmöglichkeit dafür geschaffen werden.

Für das als Beispiel in dieser Arbeit genutzte DSD-System ist im Anschluss an diese Arbeit geplant, das System in Zukunft durch die Laufzeitumgebung zu verwalten. In diesem Zusammenhang könnte eine weitere Richtung zukünftiger Arbeiten sein, die Skalierbarkeit bei großen verteilten System zu erhöhen. Dies könnte durch Hierarchisierung des verteilten Systems und der einzelnen Komponenten erreicht werden.

Die in dieser Arbeit vorgestellte Lösung wird durch einige Restriktionen der genutzten Jini-Technologie eingeschränkt. Durch die inhärente Nutzung von RMI ist es problematisch Jini mit Java-Virtual-Machines zu nutzen, die kein RMI unterstützen, wie zum Beispiel die Micro Edition (ME) für eingebettete Systeme. In [Pre00] wird eine Lösung angeboten, die RMI durch ein eigenes Verfahren ersetzt, welches die Ressourcen weniger stark in Anspruch nimmt und für die Micro Edition verfügbar ist. Um diesen RMI-Ersatz zu nutzen, ist es allerdings erforderlich die Lookupdienste anzupassen. Al-Theneyan et al. präsentieren in [ATMZ00] eine Möglichkeit, Jini in Umgebungen zu nutzen, die keine Multicast-Nachrichten unterstützen. Um auch Nicht-Java-Programmen die Teilnahme an Jini-Umgebungen zu ermöglichen, kann die Lösung in [MWL00] genutzt werden. Eine Nutzung dieser Ansätze würde die Anwendbarkeit der in dieser Arbeit vorgestellten Lösung verbessern.

Die vorgestellte Administrationskonsole kann stark erweitert werden, indem

die Situation des Netzes nicht als Momentaufnahme dargestellt wird, sondern kontinuierlich die im Netz vorkommenden Änderungen mitprotokolliert und dargestellt werden. Zusätzlich ist eine Möglichkeit zur Änderung des Systems, zum Beispiel durch manuelles Verschieben einer Komponente auf einen anderen Rechenknoten, wünschenswert.

Als Letztes sollte das gesamte System mehr Sicherheit unterstützen. Dies kann allerdings nur erreicht werden, wenn die benutzte Jini-Technologie explizit Sicherheit unterstützt, was zur Zeit noch nicht der Fall ist. Im Rahmen des Davis-Projekts [Pro02a] soll Jini um Sicherheitsaspekte erweitert werden. Allerdings ist zur Zeit dieser Arbeit dort noch kein Release einer um Sicherheitsaspekte erweiterten Jini-Umgebung erfolgt, sondern als Denkanstoß nur ein erster Entwurf einer Basis für eine sichere Jini-Umgebung. In [EN01] und [HKV00] werden andere Vorschläge für Sicherheit in Jini-Umgebungen vorgestellt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AD76] Peter A. Alsberg and John D. Day. *A Principle for Resilient Sharing of Distributed Databases.* In *Proceedings of the 2nd International Conferenc on Software Engineering*, pages 562–570. ACM press, 1976. 59

[AOS+99] Ken Arnold, Bryan Osullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, and Bryan O'Sullivan. *The Jini(TM) Specification.* The Jini(TM) Technology Series. Addison-Wesley, June 1999. 27, 73

[ARI00] ARIBA, INC., INTERNATIONAL BUSINESS MACHINES CORPORATION AND MICROSOFT CORPORATION. *Universal Description, Discovery and Integration (UDDI) Technical White Paper*, September 2000. 24

[AT01] Navid Aghdaie and Yuval Tamir. *Client-Transparent Fault-Tolerant Web Service.* In *20th IEEE International Performance, Computing, and Communications Conference*, pages 209–216, 2001. 31

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. *Efficient and language-independent mobile programs.* In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, volume 31, pages 127–136, May 1996. 13

[ATMZ00] Ahmed Al-Theneyan, Piyush Mehrotra, and Mohammad Zubair. *Enhancing Jini for use across non-multicastable networks.* Technical report, ICASE, 2000. 2000-34. 88

[AW94] Hagit Attiya and Jennifer L. Welch. *Sequential consistency versus linearizability. ACM Transactions on Computer Systems*, 12(2), 1994. 62

[AW99] D. H. Akehurst and A. G. Waters. *UML specification of distributed system environments.* Technical Report 18-99, University of Kent at Canterbury, May 1999. 39

[Ban98] Bela Ban. *Design and Implementation of a Reliable Group Communication Toolkit for Java*, 1998. 73

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Wine. *Simple Object Access Protocol (SOAP) 1.1.* Techreport, World Wide Web Consortium (W3C), May 2000. 23

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. 59, 62

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Company, 1996. 15

[BJW99] Antoine Beugnard, Jean-Marc Jezequel, and Damien Watkins. *Making Components Contract Aware. IEEE Computer*, 32(7):38–45, July 1999. 10

[Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, Menlo Park CA, 1993. (Second Edition). 38

[Bro00] *Brockhaus Multimedial Premium 2001.* Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, 2000. 17

[CDK00] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design.* Addison-Wesley, 2000. Third Edition. 11, 13, 15

[Ced93] Per Cederqvist. *Version Management with CVS*, 1993. 6

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990. 30

[Che76] Peter Pin-Shan Chen. *The Entity-Relationship Model - Towards a Unified View of Data. ACM Transactions on Database Systems*, 1(1):9–36, March 1976. 38

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990. 84

[Dij74] Edsger W. Dijkstra. *Self-stabilizing systems in spite of distributed control. Communications of the ACM*, 17(11):643–644, November 1974. 20

[DKMT96] Daniel M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari. *A Scalable and Highly Available Web Server.* In *Proceedings of the IEEE Computer Conference (COMPCON), Santa Clara*, pages 85–92, March 1996. 31

[Dol00] Shlomi Dolev. *Self-Stabilization.* MIT Press, March 2000. 21

[DYK01] Linda G. DeMichiel, L. Ümit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans^{TM} Specification.* Sun Microsystems, August 2001. Version 2.0. 25

[EGLT76] K. P. Eswaran, J.N. Gray, R. A. Lorie, and I. L. Traiger. *The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM*, 19(11):624–633, November 1976. 62

[EN01] P. Eronen and P. Nikander. *Decentralized Jini security.* In *Proceedings of the Network and Distributed System Security Symposium*, February 2001. 89

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, June 1999. 30

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. *Impossibility of distributed consensus with one faulty process. Journal of the ACM*, 32(2):374–382, 1985. 17

[FNT98] T. Fischer, J. Niere, and L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling.* Diplomarbeit, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, July 1998. 4, 39, 50

[GGT01] M. Gehrke, H. Giese, and M. Tichy. *A Jini-supported Distributed Version and Configuration Management System.* In *Proc. of the International Symposium on Convergence of IT and communications (ITCom2001), Denver, USA*, August 2001. 6

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reuseable Object-Oriented Software.* Addison-Wesley, 1994. 46

[Gie01a] Holger Giese. *Object-Oriented Design and Architecture of Distributed Systems.* Berichte aus der Informatik. Shaker Verlag, March 2001. 83

[Gie01b] Holger Giese. *Typed Component Systems, Version 1.0*. TechReport Bericht tr-ri-01-224 Reihe Informatik, Fachbereich Mathematik-Informatik, Universität Paderborn, May 2001. 57, 83

[Gif79] David K. Gifford. *Weighted Voting for Replicated Data*. In *Proceedings of the seventh symposium on Operating systems principles*, volume 7 of *ACM Symposium on Operating Systems Principles*, pages 150–162. ACM press, 1979. 59, 62

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996. 24

[GNZ01] M. Gehrke, J. Niere, and A. Zündorf. *Distributed Software Development using Jini*. In *Proc. of the* 4^{th} *International Workshop on Software Engineering over the Internet (SEoI), Toronto, Canada*, May 2001. 6

[Gra78] J. N. Gray. *Notes on Database Operating Systems*. In *Operating Systems an Advanced Course*, Lecture Notes in Computer Science. Springer Verlag, 1978. 30, 59

[HA90] P. Hutto and M. Ahamad. *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories*. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pages 302–311. IEEE Press, 1990. 19

[Har87] D. Harel. *STATECHARTS: A Visual Formalism for complex systems. Science of Computer Programming*, 3(8):231–274, 1987. 40

[Hin96] Robert M. Hinden. *IP Next Generation Overview. Communications of the ACM*, 39(6):61–71, 1996. 73

[HKV00] Peer Hasselmeyer, Roger Kehr, and Marco Vos. *Trade-offs in a Secure Jini Service Architecture*. In *USM*, pages 190–201, 2000. 89

[HT94] Vassos Hadzilacos and Sam Toueg. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report TR94-1425, Dept. of Computer Science, University of Toronto, 1994. 17

[HW90] M. P. Herlihy and J. M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. 19

[ISO98] ISO/IEC. *Information technology – Open Distributed Processing – Trading function: Specification – Part 1,2,3:*, 1998. ISO/IEC 13235-1, 13235-2, 13235-3:1998. 11

[Jal96] Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1996. 61

[JCJO96] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1996. (Revised Printing). 38

[KNNZ99] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Using UML as a visual programming language.* Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, August 1999. 39

[Köh99] H.J. Köhler. *Code-Generierung für UML Kollaborations-, Sequenz-, und Statechart-Diagramme.* Diplomarbeit, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, October 1999. 39

[Lam79] L. Lamport. *How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. IEEE Transactions on Computers*, C-28(9):690–691, 1979. 19

[Led02] Pascal Ledru. *Smart Proxies for Jini Services. ACM SIGPLAN Notices*, 37(4):57–61, April 2002. 28

[LRvV98] Nico H. Lassing, Daan B. B. Rijsenbrij, and Johannes C. van Vliet. *A View on Components.* In *DEXA Workshop*, pages 768–777, 1998. 10

[LS88] R. Lipton and J. Sandberg. *PRAM Consistency: A Scalable Shared Memory.* Techreport, Princeton University, September 1988. CS-TR-180-88. 20

[LS94] S. S. Lam and A. U. Shankar. *A theory of interfaces and modules i-Composition theorem. IEEE Transactions on Software Engineering*, 20(1):336–355, January 1994. 83

[LY99] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1999. Second Edition. 24

[Mey87] Bertrand Meyer. *Design by Contract.* Techreport, 1987. TR-EI-12/CO. 10

[Moc87] Paul Mochapetris. *DNS: The Domain Name System.* RFC 1034, Network Working Group, ISI, November 1987. 11

[MWL00] S. Müller-Wilken and W. Lamersdorf. *Enhancing JINI to support Non-JAVA Parties.* In *Proc. 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000)*, volume VII, pages 78–83. International Institute of Informatics And Systematics, 7 2000. 88

[NET00] *Microsoft .NET: Realizing the Next Generation Internet.* Techreport, Microsoft, June 2000. White Paper. 24

[Net02] NetBeans. *NetBeans,* http://www.netbeans.org/, June 2002. 31

[NS73] I. Nassi and B. Shneiderman. *Flowchart Techniques for Structured Programming. ACM SIGPLAN Notices*, 8(8):12–26, August 1973. 38

[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 2.3.1 Specification*, October 1999. Revision 2.3.1: OMG Technical Document formal/99-10-07. 24

[OMG01] Object Management Group. *OMG Unified Modelling Language 1.4*, September 2001. 9, 39, 48

[ONE01] *Sun[tm] Open Net Environment (Sun ONE) Software Architecture.* Sun Microsystems, Inc., 2001. 24

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* Phdthesis, Institut für Instrumentelle Mathmatik Bonn, 1962. 38

[Pre00] Stephan Preuß. *NetObjects - Dynamische Proxy-Architektur für Jini.* In *Proc. NET.OBJECTDAYS 2000.* Messekongresszentrum Erfurt, Germany., Oct 9-12 2000. 88

[Pro02a] Davis Project. *Davis,* http://developer.jini.org/exchange/projects/davis/, June 2002. 89

[Pro02b] Out Of The Box Project. *OutOfTheBox,* http://developer.jini.org/exchange/projects/outofbox/, June 2002. 33, 43

[Pro02c] The Jakarta Project. *ANT,* http://jakarta.apache.org/ant, 2002. 33

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991. 38

[Sal01] Jesus M. Salvo. *OpenJGraph 0.9.1 - Java Graph and Graph Drawing Project, http://openjgraph.sourceforge.net/*, December 2001. 83

[Sun98a] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*, October 1998. Revision 1.8, JDK 1.4. 11, 28, 42

[Sun98b] Sun Microsystems Inc. *Java^{TM} Object Serialization Specification*, November 1998. Revision 1.43, JDK 1.2. 28, 74

[Sun99] Sun Microsystems. *JavaSpaces Specification*, January 1999. Revision 1.0. 30

[Sun01a] Sun Microsystems. *Jini Technology Core Platform Specification*, December 2001. Revision 1.2. 27, 73

[Sun01b] Sun Microsystems. *RIO - Architecture Overview*, 2001. 2001/03/15. 32

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998. 9

[Tho79] Robert H. Thomas. *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM Transactions on Database Systems*, 4(2):180–209, 1979. 59, 62

[Tho97] Tommy Thorn. *Programming languages for mobile code. ACM Computing Surveys*, 29(3):213–239, 1997. 13

[Tol02] Robert Tolksdorf. *Programming Languages for the Java Virtual Machine, http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html*, 2002. 24

[TvS02] Andrew Tanenbaum and Maarten van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002. 12, 15, 17, 18, 19, 20, 62

[Uni82] United States Department of Defense. *Reference Manual for the Ada Language, revised mil-std 1815 edition*, July 1982. 10

[VN02] Steven J. Vaughan-Nichols. *Web Services: Beyond the hype. IEEE Computer*, 35:18–21, February 2002. 23

[Wal99] Jim Waldo. *The Jini architecture for network-centric computing. Communications of the ACM*, 42(7):76–82, 1999. 27

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1983. 10

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. *A Distributed Object Model for the Java System. Computing Systems*, 9(4):265–290, 1996. 28

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendal. *A Note on Distributed Computing.* Techreport, Sun Microsystems Laboratories, November 1994. TR-94-29. 13, 14, 26

[Wyb90] Dieter Wybranietz. *Multicast-Kommunikation in verteilten Systemen.* Springer Verlag, 1990. 27, 57, 72