Universität Bielefeld
Technische Fakultät
Arbeitsgruppe Rechnernetze und verteilte Systeme

Diplomarbeit

Konzeption eines verteilten Datenarchivierungssystems

von

Jan E. Hennig

Betreuer: Prof. Peter B. Ladkin PhD FBCS
I Made Wiryana

Bielefeld, September 2003

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig bearbeitet und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bielefeld, den 5. September 2003 Jan E. Hennig

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Immer mehr Daten müssen über längere Zeiträume verfügbar gehalten werden. Dies liegt zum einen am stetig wachsenden Datenaufkommen an sich und zum anderen an der Erkenntnis, dass auch ältere Datenbestände für aktuelle Analysen und Auswertungen benötigt werden. In einzelnen Gebieten existieren auch gesetzliche Auflagen zur Bewahrung von Daten. Zudem können gewonnene Daten nicht immer sofort verarbeitet werden. Immer größere Datenmengen physikalisch abzuspeichern ist mit der aktuellen Entwicklung bei den Speichermedien nicht allzu schwierig.
Das Hauptproblem bei dieser anfallenden Datenflut ist, den Überblick über die Daten zu bewahren. Bei zunehmender Datenmenge wird es zum Wiederfinden unumgänglich, die Daten zu ordnen und zu sortieren. Aber auch hier stoßen größtenteils manuelle Systeme an ihre Grenzen. Abhilfe schaffen Datenarchivierungssysteme, die analog zu den altbekannten Archiven von auf Papier gespeicherten Daten diese Aufgabe für digitale Daten übernehmen.
Herkömmliche Datenarchivierungssysteme jedoch haben einige Schwachstellen. So sind einige davon auf einen bestimmten Regelsatz für die Ordnung der Daten festgelegt, so dass die Benutzung auf eng begrenzte Gebiete beschränkt bleibt. Andere wiederum kennen keine Rechteverwaltung, so dass das Einhalten etwaiger Zugriffsrechte mit

anderen auf das System aufgesetzten Verfahren sichergestellt werden muss. Zudem bekommen viele Systeme Schwierigkeiten, wenn sie deutlich mehr als eine zur Erstellungszeit vorgesehene Datenmenge verwalten müssen. Diese Schwierigkeiten können sowohl die Erweiterung der Speicherkapazität als auch die Zugriffsgeschwindigkeit betreffen. Letzteres bezieht sich hauptsächlich auf Systeme, welche auf eine streng zentralisierte Struktur setzen.
Ich entwickle hier nun ein verteiltes Datenarchivierungssystem, das die nun bekannten Probleme berücksichtigt und versucht einen gangbaren Weg zwischen den jeweiligen Problemextremen zu finden und aufzuzeigen.

1.2 Aufgabenfelder eines Archivierungssystems

Ein Archivierungssystem ist ein komplexer Verbund. Zum Lösen der Gesamtaufgabe werden mehrere Aufgabenfelder miteinander verbunden. Für die Erfüllung der Aufgabe des eigentlichen Archivierens umfasst dies folgende zeitlich aufeinander aufbauende Aufgaben:

Ⓐ Datenspeicherung (Storing)

Die Datenspeicherung umfasst den Vorgang, die von einem Produzenten (aus Sicht des Archivierungssystems) erzeugten Daten und deren Beschreibung für die spätere Suche und Auslieferung abzulegen und für den Zugriff verfügbar zu machen. In einem verteilten System muss dabei unter Umständen die neue Information nicht nur am Punkte des Aufnehmens der Daten und Beschreibungen, sondern auch über die restlichen Orte verteilt und dort bekanntgegeben werden. In diesem Aufgabenschritt wird vom System selbst die Beschreibung der Ablageposition der Daten erzeugt und in die vom Produzenten gelieferte Beschreibung der Daten mit aufgenommen. Für die Datenspeicherung ist somit schreibender Zugriff für die Datenablage und die Ablage der Beschreibungen nötig.

Ⓑ Datensuche (Query)

Die im vorherigen Schritt in das System aufgenommenen Daten müssen wiedergefunden werden. Der Benutzer – bzw. aus Sicht des Archivierungssystems der Konsument – wird daher eine Suche ausführen, um die für ihn relevanten Daten zu finden. Für diese Suche ist allein die bei der Speicherung übergebene Beschreibung der Daten von Belang. Dies kann vom System zu Effizienzsteigerungen ausgenutzt werden, da die Beschreibung üblicherweise erheblich geringer im Umfang ausfällt als die eigentlichen Daten. Für die Datensuche ist lediglich lesender Zugriff auf die Beschreibungen, nicht auf die archivierten Daten nötig.

Ⓒ Datenauslieferung (Retrieval)

Als letzten Schritt zur Wiedererlangung der Daten wird der Konsument eine Auslieferung der Daten verlangen. Dies kann er anhand der im Schritt Ⓑ an ihn gelieferten Auswahl tun. Hierzu wird vom System die im Schritt Ⓐ erzeugte Beschreibung der Ablageposition der Daten zu Rate gezogen und auf diese lesend zugegriffen. Die Daten werden dann letztendlich an den Konsumenten ausgeliefert.

In der Praxis werden Produzent und Konsument möglicherweise auch ein und dieselbe Person sein. Für die Betrachtung der Aufgaben ist dies jedoch irrelevant.

Die in Ⓐ bis Ⓒ zu verarbeitenden Daten sind allesamt statischer Natur. Sie werden im ersten Schritt in das System aufgenommen und in den anderen Schritten nicht verändert sondern nur abgefragt. Hingegen könnte die Ermöglichung eines *Austauschens* von bereits vorhandenen Daten durch neue Daten im Schritt Ⓐ dieses ändern. Sofern ein System diesen Aspekt zulässt, muss dieser bei der Betrachtung der Speicherung genauer beachtet werden. Für die grundsätzliche Aufgabe des Archivierens allerdings ist eine derartige Funktionalität nicht nur ungewollt sondern teilweise sogar schädlich, da der Sinn eines Archivs in der Bewahrung der Daten liegt und niemals ein Austausch oder eine Veränderung daran vorgenommen werden soll.

Zu den ureigenen Aufgaben der reinen Archivierung selbst gesellen sich noch Aufgaben zur Pflege und Inbetriebhaltung des Systems:

Ⓓ Verwaltung (Management)

Dieses Aufgabenfeld umfaßt die Handhabung des Systems. Aus den Daten *über* die vom System ausgeführten Aktionen aus den Aufgaben Ⓐ bis Ⓒ können Rückschlüsse gebildet werden über den Bedarf und die nötigen Maßnahmen zur Erhaltung und Verbesserung des Betriebs. Diese über den Betrieb gesammelten Informationen sind dynamische Informationen, welche speziell für das Betriebsumfeld gelten. Für die Auswertung ist also der lesende Zugriff auf Vorgangsprotokolle nötig, nicht aber ein Zugriff auf die Daten und Beschreibungen selbst. Für eine Reorganisation im Laufe von Verbesserungen durch die Verwaltung hingegen kann durchaus der Zugriff, sowohl lesend wie auch schreibend, auf alle Systemteile vonnöten sein.

1.3 Daten und Metadaten

Die im System anfallenden Daten werden logisch getrennt in „Daten” und zugehörige, die Daten beschreibende Daten, die sogenannten „Metadaten”. Metadaten sind also Daten *über* andere Daten. Als Analogie zum Bereich herkömmlicher Archive[1] entspricht ein in den Archivbestand aufzunehmendes Buch den aufzunehmenden Daten. Die zu dem Buch als Beschreibung in die Repertorie[2] aufgenommenen Metadaten wie Verfasser und Erscheinungsdatum entsprechen dabei den beschreibenden Metadaten zu den digitalen Daten wie beispielsweise eingebender Benutzer und Modifikationsdatum. Jene Metadaten werden vom System für das Wiederfinden der eigentlichen Daten im Gesamtdatenbestand verwendet.

Darüberhinaus existieren im System eine Vielzahl von „Verwaltungsdaten”, die für den eigentlichen Systembetrieb verwendet werden, jedoch selbst nicht Teil des im System vorhandenen Archivs sind. Eine Analogie für Verwaltungsdaten ist eine Liste mit Zugangsberechtigten zu einem Archiv oder Archivteil oder eine Übersicht über die ver-

[1] Eine alte, ausführliche Beschreibung der Aufgaben von Archiven findet sich in Meyers Konversationslexikon [15].

[2] Repertorie: Fachbegriff aus dem Bereich der Archivierung; bezeichnet das Verzeichnis, in das alle wichtigen Daten zu den Archivalien selbst wie beispielsweise Titel und Entstehungsdatum sowie zum Stand- bzw. Lagerort im Archiv eingetragen werden.

schiedenen Archivabteilungen.

1.4 Die Problemstellung und der Aufbau der Arbeit

In dieser Arbeit werden verschiedene bestehende und besonders auch neue Einzelkonzepte zu einem Ganzen zusammengeführt, das sich zu einem verteilten Datenarchivierungssystem vereinigt. Zurückgreifen möchte ich dabei auf bestehende und neue Konzepte von örtlich gebundenen sowie auch von verteilten Datei- wie auch Archivierungssystemen.
Es soll ein System entworfen werden, welches beliebig kategorisierte Daten archivieren und wieder ausliefern kann. Als Hauptanforderung an das System ist dabei zu berücksichtigen, dass Daten an verschiedenen Standorten in das System eingegeben werden und hauptsächlich an denselben Standorten wieder abgerufen werden. Trotzdem soll bei Bedarf systemweit darauf zugegriffen werden können. Das System soll dabei ermöglichen, dass auch bei zeitweiligem Verlust der Datenverbindung zwischen diesen Standorten eingeschränkt weiter gearbeitet werden kann, d.h. in diesem Fall ohne eine systemweite Zugriffsmöglichkeit. Weiterhin soll das System Rechnung tragen über die administrativ zugewiesenen Rechte, die Daten einzusehen oder zu verändern.

Zuerst wird sich Kapitel 2 dem aktuellen Stand der Entwicklung auf dem Gebiet der Archivierungssysteme widmen. Die genauen Ausprägungen der sich ergebenden Anforderungen klärt Kapitel 3. Es schließen sich ein Kapitel zum eigentlichen Design des Archivierungssystems sowie eines zur prototypischen Implementation an. Abschließend wird eine Zusammenfassung und ein Ausblick auf eine mögliche Weiterentwicklung gegeben.

Kapitel 2

Überblick über den aktuellen Entwicklungsstand

In diesem Kapitel werden verschiedene Konzepte und Möglichkeiten vorgestellt, welche sich mit größeren und kleineren Teilproblemen auf den Gebieten der Gestaltung von Verteilten Systemen, der Konsistenzbewahrung in einem Verteilten System, der Datenhaltung und -langzeitarchivierung und der Zugriffsverwaltung beschäftigen. Anschließend werden ausgewählte Archivierungssysteme und -programme vorgestellt, welche den derzeitigen Entwicklungsstand repräsentieren und mit Systemen und Programmen verglichen, welche einzelne Teilaspekte aus dem Spektrum der Gesamtaufgabe der verteilten Archivierung abdecken. Zuerst aber werden einige wichtige Begriffe geklärt, die im weiteren Verlauf dieser Arbeit in der folgend gegebenen Definition verwendet werden.

2.1 Begriffsklärung

Es gibt derzeit eine Vielzahl von Systemen, die unter dem Begriff „Archivierungssystem" beworben oder entwickelt werden. Die jeweilige Ausprägung der einzelnen Systeme hingegen variiert sehr stark. Die im Folgenden aufgeführten Begrifflichkeiten werden daher dazu verwendet, diese Ausprägungen einfacher eingrenzen und kennzeichnen zu können.

2.1.1 Architektur

Verteiltes System

Die Funktionalität des Systems baut auf der Verteilung auf mehr als einen Rechner auf. Als verteilte Systeme werden hier nur Systeme angesehen, die immanent auf die Verteilung aufgebaut sind. Es reicht dazu nicht aus, dass beispielsweise mehrere Webbrowser parallel auf das System zugreifen können.

Spezialisiert

Das System ist spezialisiert auf die Verwaltung nur bestimmter Daten, evtl. zusammengehend mit speziellen Datenformaten für diese Daten.

Allgemein

Das System kann mit Daten allgemeiner Art umgehen und ist nicht beschränkt auf einen bestimmten bzw. wenige Datentypen oder ein bzw. wenige Datenformate.

Feste Kategorien

In das System ist die Möglichkeit (oder der Zwang) eingebunden, die Daten in eine vorgegebene Kategorie einzubinden. Diese wird zum Organisieren und zum späteren Wiederauffinden der Daten verwendet.

Freie Kategorien

Ähnlich zu festen Kategorien werden bei einem System mit freien Kategorien auch Daten gruppiert und eingeordnet. Die verwendbaren Kategorien sind jedoch vom Benutzer frei definierbar. Dies schließt die Vorgabe von Kategorien durch den Hersteller nicht aus.

2.1.2 Besondere Funktionen

Versionskontrolle

Ein System mit Versionskontrolle verwendet ein Datenmodell mit der Möglichkeit einer zeitlichen Entwicklung der Daten. Zu verschiedenen Zeitpunkten wird der dann aktuelle Zustand der Daten dem System bekannt gegeben. Dieses legt jenen aktuellen Zustand als eine weitere Version in der diesen Daten zugeordneten Historie ab.

Rechte- bzw. Zugriffsverwaltung

Auf die im System gespeicherten Daten darf nicht beliebig zugegriffen werden. Es werden – meist einem Benutzerkonto oder ganzen Benutzergruppen zugeordnet – Rechte für den Zugriff verwaltet und evtl. Buch über die erfolgten Zugriffe geführt.

Formatübersetzung

Daten liegen in einem bestimmten Format gespeichert vor. Benötigt werden aber diese Daten in einem anderen Format, sei es durch die Verwendung mehrerer verschiedener Programme gleichzeitig oder dem Veralten von eingesetzten Programmen. Das System kann auf Benutzerwunsch die Daten direkt aus einem Format in ein anderes konvertieren.

Merkmalsextraktion

Das System kann automatisch und direkt aus den Daten Merkmale zur Verwendung in der Datenorganisation herausarbeiten, nach denen später auch gesucht werden kann.

Hardwareeinbindung

Zum System gehört neben der Software auch eine möglicherweise speziell darauf zugeschnittene Hardware, damit das Gesamtsystem funktioniert. Diese ist dabei fest mit dem softwareseitigen Teil des Systems verbunden bzw. die Aufgabe kann ohne das Vorhandensein der Hardware vom System nicht erfüllt werden.

2.2 Forschung und bestehende Konzepte

Aus der Aufgabenstellung aus Kapitel 1.2 ergeben sich verschiedene Problemfelder, die es zu beachten gilt. Verschiedene Konzepte sind entwickelt worden oder noch in der Entwicklung, um diesen Problemen zu begegnen:

2.2.1 Trennen der Ablage von Daten und Metadaten

Die eigentlichen Daten werden getrennt von ihren Beschreibungen, den Metadaten, abgelegt. Der Datenbestand der Beschreibungsdaten ist üblicherweise erheblich kleiner im Umfang als der der Daten. Diese Methode ermöglicht es, den Datenbestand der Beschreibungsdaten effizienter zu durchsuchen. So wird zunächst nur der Zugriff auf die Ablage der Beschreibungsdaten und nicht permanent zu allen Daten benötigt. Dieses Prinzip wird ausführlich in „Reference Model for an Open Archival Information System (OAIS)" des CCSDS Secretariats der National Aeronautics and Space Administration (NASA) [5] beschrieben. Abbildung 2.1 zeigt den Hauptdatenfluss der Aufgabenfelder Ⓐ bis Ⓒ in einem mit dieser Datentrennung ausgerüsteten Archivierungssystem.

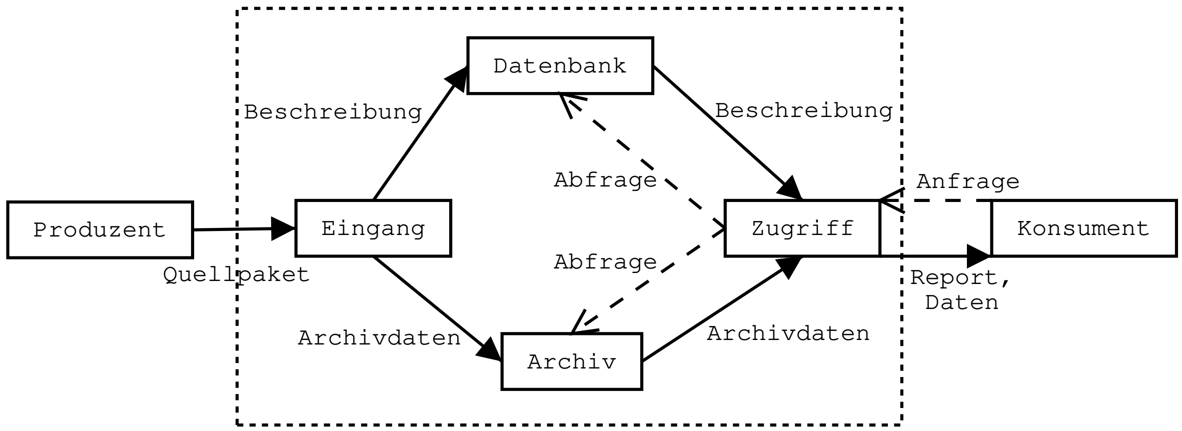

Abbildung 2.1: Hauptdatenfluss in einem Archivierungssystem mit Trennung von Daten- und Metadatenablage

2.2.2 Datenablage im Dateisystem

Die Ablage der Archivdaten im herkömmlichen Dateisystem ist eine Erweiterung zum Konzept aus 2.2.1 und wird von Käster in seiner Diplomarbeit „Konzeption und Implementierung eines SQL-Datenbank-Backends zur Speicherung von Multimediadaten” [3] für Bilddaten behandelt. Datenverarbeitende Zusatzprogramme oder Plugins wie Merkmalsextraktionen können sehr einfach mit den vorhandenen Dateien ihre Aufgabe erledigen und müssen nicht an spezielle Datenformate und -zugriffsoptionen des Archivierungsprogramms angepasst werden. Dieses Vorgehen ermöglicht es außerdem, die bereits länger bekannten Vorteile von Dateisystemen zu nutzen, inklusive der gängigen Methoden zur Sicherung der Daten von Dateisystemen. Im Falle eines Systemausfalls können die Daten so leichter mit herkömmlichen Datenrettungsmethoden wiederhergestellt werden. Allerdings können abgelegte Datenträger ebenso leicht von Fremdpersonen gelesen werden, an dieser Stelle ist also besondere Vorsicht angebracht, siehe Abschnitt 2.2.6.

2.2.3 Metadaten in einer Datenbank

Die Ablage der Metadaten in einer Datenbank ist eine Erweiterung zum Konzept der Datentrennung aus 2.2.1 und wird in vielen Texten beschrieben, besonders ausführlich in „Reference Model for an Open Archival Information System (OAIS)” der NASA [5]. Dieses Vorgehen ermöglicht die Nutzung langjähriger Erfahrung auf dem Gebiet der Suche in Datenbeständen von Datenbanken, welche für diese Aufgabe optimiert sind. Die Metadaten müssen für die Suche (siehe Abbildung 2.2 auf der nächsten Seite) ohnehin ständig verfügbar vorgehalten werden.

2.2.4 Mittlerschicht zwischen Datenablage und Benutzer

Das Erschaffen einer Mittlerschicht zwischen Datenablage und Benutzer behandelt mehrere Probleme:

- Die Trennung ermöglicht es, dass zuerst nur die Metadaten für die Datensuche benötigt werden, bis der Benutzer den Eintrag seiner Wahl anfordert und dieser daraufhin ausgeliefert wird (siehe Abbildung 2.2).

- Es führt ebenfalls zu einer besseren Skalierbarkeit in der Programmierung, da die Aufgabenbereiche voneinander abgeschirmt sind und so im Programm auch getrennt und über wohldefinierte Kontaktstellen erstellt werden können (siehe Abbildung 4.1 auf Seite 49).

- Die Datenablage ist nicht mehr an einen oder mehrere Benutzer bzw. Standorte gebunden. Hierdurch kann das Gesamtsystem leicht erweitert werden (siehe Abbildung 4.2 auf Seite 50), denn der Zugriff auf die Archivdaten wird klein gehalten.

Das Erschaffen einer Mittlerschicht mit diesen Vorteilen wird für herkömmliche Dateisysteme in „Cluster File System in Compaq TruCluster Server" [46] der Fa. Compaq, „Frangipani: A Scalable Distributed File System" [47] von Thekkath, Mann und Lee, in verschiedenen Arbeiten von Akinlar et al. [48, 49, 50], in „Lustre: A Scalable, High-Performance File System" [51] der Lustre.org, sowie in Miners „New Advances in the Filesystem Space" [52] behandelt.

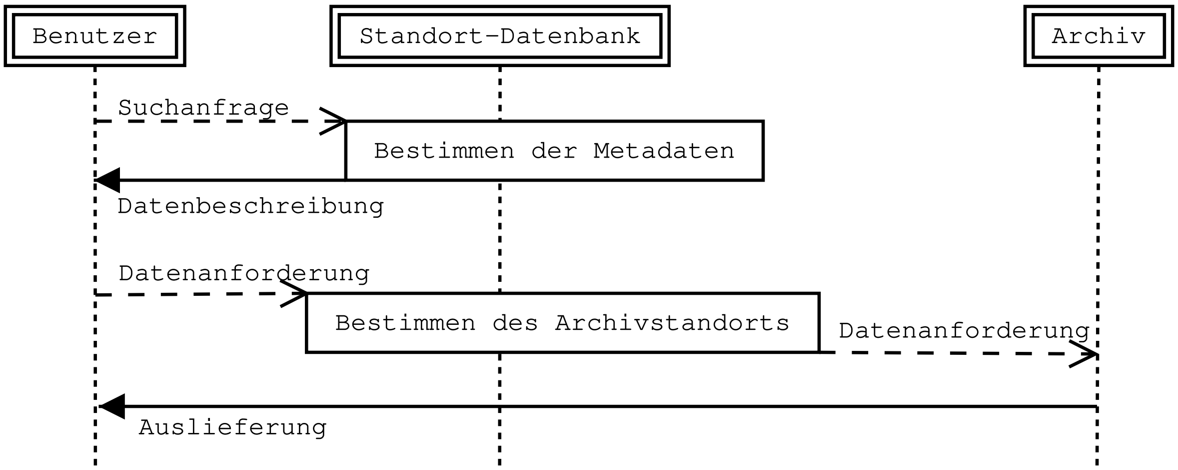

Abbildung 2.2: Zeitlicher Ablauf von der Suche bis zur Auslieferung

2.2.5 Datenreplikation

Die Datenreplikation zur Skalierbarkeit des Gesamtsystems umfasst in erster Linie die Replikation der Metadaten an allen Standorten. Diese werden für jeden Suchvorgang benötigt, die archivierten Daten weniger oft (erst nach einer erfolgreichen Suche, siehe Abbildung 2.2 auf der vorherigen Seite). Je näher (im Sinn von weniger aufwendig zu besorgen) die Daten zum Benutzer liegen, desto leichter fällt es, in ihnen zu suchen. Es erwächst aber das Problem der Konsistenzhaltung, besonders bei zeitweiligem Ausfall eines Systemteils. Aspekte von Datenreplikation und dabei möglichen Einsparungen von Datentransfers werden in „The Design of a Multicast-based Distributed File System"[45] von Grönvall et al. behandelt. Häufig benötigte Daten werden sofort repliziert, seltener benötigte Daten erst bei Bedarf. Eine Replikation der Daten neben den Metadaten ist hierbei nicht ausgeschlossen, zumal die Daten in einem Archivierungssystem nach deren Aufnahme nicht mehr verändert werden und sich dadurch die Konsistenzerhaltung erleichtert.

2.2.6 Regelmäßiger Datenträgeraustausch

Datenträger altern mit der Zeit oder werden bei stetem Gebrauch abgenutzt. Daher wird in allen größeren Archivierungssystemen zur Bestandssicherung ein regelmäßiger Austausch der Datenträger, die die Persistenz des Archivs bilden, durchgeführt. Die Grundprinzipien dazu werden im OAIS Reference Model der NASA [5] ausgeführt. Problematisch ist dabei auch das Vernichten oder die abgesicherte Aufbewahrung ausgedienter Datenträger, damit unautorisierten Personen die zum Zeitpunkt des Austauschs immer noch lesbaren Datenträger nicht in die Hände fallen. Dieses kann nicht mehr durch ein Archivierungssystem alleine sondern muss durch ein umgebendes Gesamtkonzept sichergestellt werden.

2.2.7 Verteilte Datenspeicherung

Eine Erhöhung der Datensicherheit kann durch eine verteilte Datenspeicherung erreicht werden. Durch eine passende Verteilung der Daten auf verschiedene Standorte (im Sinne von physikalisch von einander entfernten Orten) kann, besonders im Zusammenhang mit der Datenreplikation, das Gesamtsystem auch bei katastrophalen Einwirkungen auf einen einzelnen Standort weiterbestehen. Dieses benötigt nicht notwendigerweise ein verteiltes System, eine identische Kopie eines an einem Standort vorhandenen Systems an einem zweiten Standort erfüllt diese Bedingung bereits. Die Vor- und Nachteile verteilter Datenspeicherung spielen in Compaqs Cluster File System [46], Frangipani von Thekkath et al [47], in verschiedenen Arbeiten von Akinlar et al. [48, 49, 50], und in „Lustre: A Scalable, High-Performance File System" [51] der Lustre.org, eine Rolle. Sicherheit in Speicherungssystemen allgemein ist Thema von Saltzer et al. in „End-to-End Arguments in System Design" und von „Security Must Be Baked In, Not Painted On" von Patilla.

2.2.8 Gemeinsamer Namensraum

Das Erhalten eines gemeinsamen Namensraums für alle Systemteile umfasst zum einen die Zugriffsmöglichkeit auf eine Datei über ein und denselben Namen von verschiedenen Standorten aus und ermöglicht darauf basierend eine ebenso einheitliche Verwaltung der Zugriffsbeschränkungen und -rechte. Die Vor- und Nachteile eines gemeinsamen Namens- und Suchraums sowie das Einbetten von Zugriffsrechten vermittelt „A Scheme to Construct Global File System" [11] von Hua, Chaoyang, Yafei, Bin und Xiaoming. Ein gemeinsamer Namensraum kann bereits durch die Vergabe einer eindeutigen Nummer für jeden Eintrag geschaffen werden. An diese Nummer werden dann beispielsweise die Leserechte für das damit assoziierte Objekt gebunden. Die Schwierigkeit bei einem verteilten System liegt in der möglichst wenig aufwendigen Synchronhaltung bei der Nummernerstellung.

2.2.9 Zentralisierung

Zentralisierung stellt eine Erweiterung zu 2.2.8 dar und umfasst die Bildung einer – mehr oder weniger ausgeprägten – Hierarchie zur Erhaltung eines gemeinsamen Namensraumes und zur Überwachung der Zugriffsbeschränkungen und -rechte. Durch die zentralisierte Bearbeitung kann die Einstellung dieser Rechte an einem Ort erfolgen. Dies erleichtert die Administration. Allerdings kann diese Zentralisierung schnell auch zu einem Flaschenhals für die Erweiterungsmöglichkeiten und die Skalierung des Systems werden, da alle oder zumindest viele Anfragen über eine zentrale Stelle laufen müssen. Einen grundlegenden Aufbau eines zentralisierten Archivierungssystems beschreibt das OAIS Reference Model der NASA [5].

2.2.10 Dezentralisierung

Das Gegenteil von 2.2.9 führt zum Abbau von Hierarchien. In einem Extremum, der Hierarchielosigkeit, sind alle Teile eines Systems gleichwertig und müssen von Teil zu Teil (Peer to Peer, P2P) kooperieren. Ein möglichst flaches System kann leicht erweitert werden, aber die Zusicherung eines gemeinsamen Namensraums und der Zugriffsbeschränkungen und -rechte fällt schwer. Die Frage der Routenfindung und der Datensicherheit in flachen Hierarchien wird in „Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems" [53] von Rowstron und Druschel, „Kademlia: A Peer-to-peer Information System Based on the XOR Metric" [54] von Maymounkov und Mazières, „Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications" [55] von Stoica, Morris et al., von Burkard in „Herodotus: A Peer-to-Peer Web Archival System" [56] und von Dabek in „A Cooperative File System" [57] behandelt.

2.2.11 Sichere Reihenfolge von Aktionen

Ähnlich wie beim Zweiphasen-Commit-Protokoll (dieses wird im Zuge einer Erweiterung zu einem robusten Commit-Protokoll in Abschnitt 4.2.3 beschrieben, siehe auch

„Operating Systems Concurrent And Distributed Software Design" [1] von Bacon und Harris, sowie „Asynchronous Lock Distribution in the New File Repository" [4] von Risnes) bei Datenbanken muss auch in einem verteilten System auf Konsistenzerhaltung der Daten und Metadaten geachtet werden. Es gibt verschiedene Ansätze, um dieses bei minimalem Kommunikationsaufwand zu bewerkstelligen, um eine größtmögliche Skalierbarkeit zu erreichen. Eine Möglichkeit stellen sichere, systemweit einheitlich einzuhaltende Reihenfolgen von Aktionen dar. Einige Ausprägungen davon werden in „DiFFS: a Scalable Distributed File System" [43] von Karamanolis et al. beschrieben, die auch den zeitweiligen Ausfall von Knoten oder Netzwerkteilen abzusichern vermögen. Die dort beschriebene Möglichkeit weist jedoch zusätzlichen Aufwand auf für eine regelmäßig stattzufinden habende „Garbage Collection", die bei Ausfällen übriggebliebenen und unverknüpften Daten aus dem System entfernen muss.

2.2.12 Persistente Nachrichten

In einer verteilt arbeitenden Anwendung müssen zur Koordination der Einzelteile Nachrichten zwischen den Systemteilen ausgetauscht werden. Dabei ist es in vielen Fällen wichtig, dass eine Nachricht ihr Ziel genau einmal erreicht, beispielsweise dann, wenn die Kontrolle über eine gemeinsame Ressource (z.B. über ein Semaphor) von einem Systemteil an einen anderen übergeben wird. Im Falle des Ausfalls der Verbindung zwischen den Systemteilen oder eines der involvierten Systemteile kann es bei herkömmlichen Netzen und Algorithmen zu Inkonsistenzen kommen, welche durch zum Teil sehr aufwendige Verfahren nach einem solchen Ausfall wieder beseitigt werden müssen, bevor mit der Arbeit fortgefahren werden kann. Lowell und Chen stellen in „Persistent Messages in Local Transactions" [10] eine protokollunabhängige Möglichkeit zur vorsorglichen Behandlung dieser Problematiken vor, die die bislang üblichen und aufwendigen nachsorgenden Maßnahmen überflüssig machen kann. Diese Möglichkeit ist das Verwenden von persistentem Speicher[1] zur Absicherung von in Transaktionen gesichertem Nachrichtenversand und zugehörigen Variablenänderungen. Dieses Kon-

[1] Nichtflüchtiger Speicher wie Magnetic RAM (MRAM) oder Festplatten

zept führt jedoch zu einem erhöhten Aufwand beim Versenden einer Nachricht und ist sehr von der Geschwindigkeit des verwendeten Speichers abhängig, so dass es in der Praxisanwendbarkeit Probleme bereitet.

2.2.13 Kategorien im Suchraum

Es lassen sich benutzerspezifizierte Kategorien in den Suchraum einbinden. Wie Abbildung 2.3 auf der nächsten Seite zeigt, gibt es verschiedene Strategien, um eine Suche in einem Datenbestand zu vereinfachen. Ein universelles Archivierungssystem sollte sich hierbei möglichst flexibel verhalten und es dem Benutzer ermöglichen, eigene Vorstellungen von Kategorien, auch mehrstufiger, einzubringen. Dieses Konzept wird in „The New BFS" [14] von Younger und von Käster in seiner Diplomarbeit [3] vertreten.

2.2.14 Einbetten automatischer Übersetzung und Emulation

Dieses Konzept steht ein wenig abseits von den übrigen. Es befaßt sich mit dem Problem des Aktuellhaltens von Datenformaten, das durch die stetige Weiterentwicklung der verwendeten Software und auch teilweise der Hardware auftritt. Dies geschieht, wenn nach und nach die Fähigkeit aktueller Programme, die Datenformate von älteren Programmen oder Programmversionen zu lesen, verloren geht, da die älteren Programme nicht mehr vom Hersteller unterstützt werden. Entweder müssen die damit in einem veralteten Format abgelegten Daten verlustfrei in ein neues Format konvertiert werden oder es muss eine alte Anwendung stets benutzbar über die Möglichkeit der Emulation der alten Programmumgebung parat gehalten werden. Andernfalls werden die alten Daten plötzlich nutzlos. Eine direkte Einbettung von derartigen Übersetzungsmöglichkeiten in ein auf lange Jahre angelegtes Archivierungskonzept könnte dabei das Problem der Neuablage und Neuerfassung alter Daten nach einer manuellen Konvertierung umgehen. Diese Idee wird in „Digitales Alzheimer - Maßnahmen gegen den Gedächtnisschwund bei digitalen Bilbiotheken" [6] von Sietmann und in „Wider das digitale Vergessen" [7] von Rötzer kurz umrissen. In Ecksteins Artikel „Gemeinsames Dokumentenformat: Keine Einheitssprache in Sicht" [9] werden Überlegungen zur Ver-

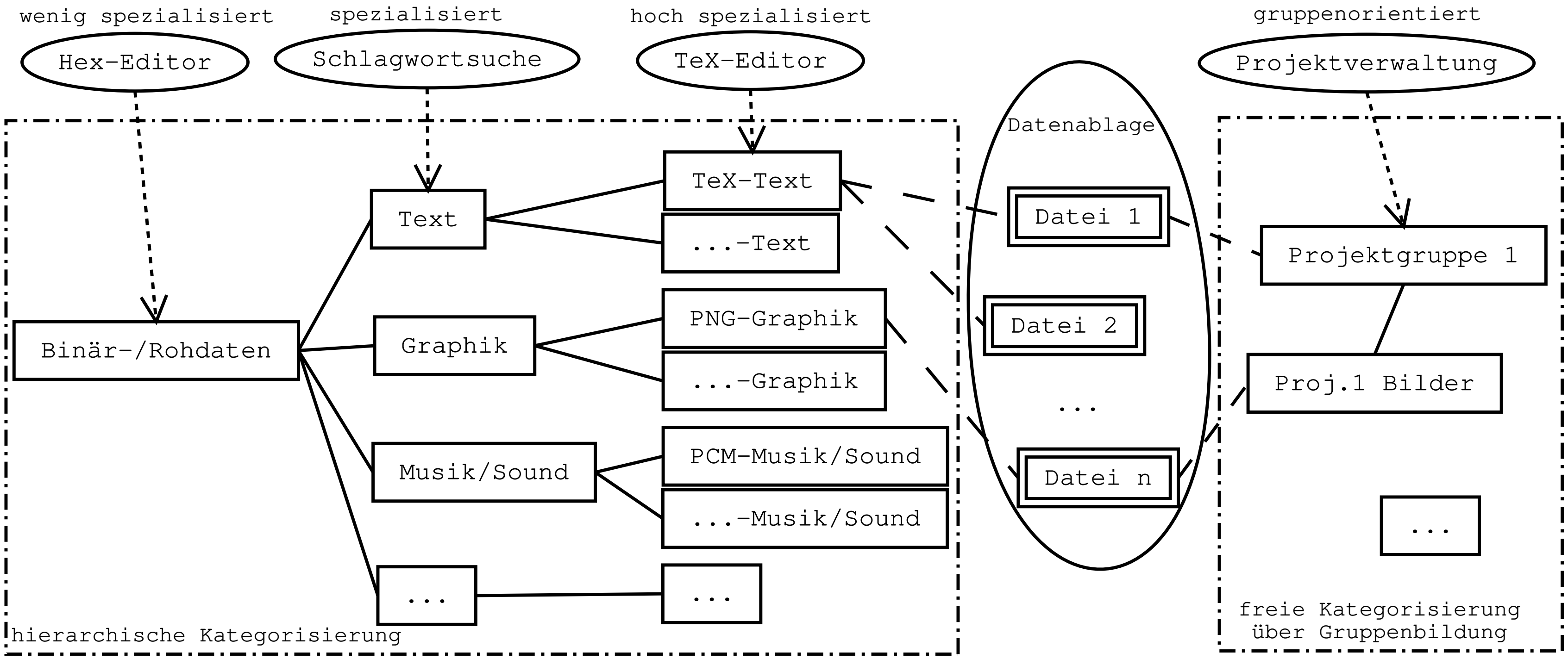

Abbildung 2.3: Kategorienbildung und Beispielanwendungen anhand von hierarchischer und frei gruppierbarer Ordnung: Dargestellt sind in der Mitte die eigentlichen Daten in einer Datenablage und rechts und links daneben zwei Möglichkeiten für Sichten auf diese Daten. Der rechte Teil zeigt eine Möglichkeit der freien Gruppierbarkeit dieser Daten, wie sie beispielsweise mit Hilfe von Projektverzeichnissen und -gruppen realisiert werden kann. Der linke Teil zeigt, wie diese Daten in einem Kategorienbaum einsortiert werden könnten. Oberhalb des linken Teils sind Beispielanwendungen genannt, welche von links nach rechts gehend immer weiter spezialisierte Datenkategorien für ihren Betrieb voraussetzen: So kann ein Hex-Editor beliebige Daten verarbeiten, nicht jedoch ein auf TEX spezialisierter Texteditor. Die beiden vorgestellten ordnunggebenden Möglichkeiten schließen sich dabei nicht notwendigerweise gegenseitig aus sondern können miteinander koexistieren.

einheitlichung von Dokumentformaten dargelegt. Solange es unterschiedliche Formate gibt, wird auch eine Übersetzung zwischen Formaten benötigt werden.

2.3 Bestehende Systeme

Nachdem nun ein Überblick über bestehende und neue Konzepte gegeben wurde und für den Vergleich von Systemen notwendige Begriffe in deren Bedeutung eingegrenzt und festgelegt wurden, kann nun eine Vorstellung und ein Vergleich der Funktionalität von bestehenden Systemen erfolgen. Zuerst werden nun exemplarisch für die große Menge an bestehenden Archivierungssystemen derer zehn aktuelle Archivierungssysteme vorgestellt, welche Pate für Gruppen ähnlich strukturierter Systeme stehen. Anschließend werden noch einige Vergleichssysteme vorgestellt, welche zwar selbst kein Archivierungssystem darstellen, jedoch Teilaspekte der dort benötigten Funktionalität abdecken. Sie dienen der Veranschaulichung der Einzelaspekte für die in Abschnitt 2.4 folgend vorgenommene Auswertung des derzeitig aktuellen Entwicklungsstandes.

2.3.1 Aktuelle Archivierungssysteme

Archie

Aus der Produktbeschreibung (siehe [20]):
„Archie verwaltet die Dokumente zentral [, ...] verbindet die Funktionalität eines Archivierungssystems, Dokumentenmanagementsystems und Content Management Systems [, ...] ist eine Client/Server-Anwendung und basiert auf einer SQL Datenbank. [...] Sie können alle Versionen einer Datei verfolgen mit dem Hinweis auf die Person, die die Änderungen durchgeführt hat."
Archie steht mit dieser Funktionalität hier stellvertretend für zentral organisierte Archivierungssysteme mit Versions- und Zugriffsverwaltung, welche mit allgemeinen Daten ohne weitere Spezialisierung umgehen können.

smartDax

Aus der Produktbeschreibung (siehe [21]):
„Das digitale On-Line-Archivierungssystem auf CD [...] Archivierte Daten [sind] über NFS erreichbar. [...smartDax ist] Hard- und Software-Lösung in einem."
smartDax stellt beinahe einen klassischen Fileserver dar, der allerdings abweichend davon die Daten nicht auf Festplatten sondern auf CDs unterbringt und somit ein Write-Once-Read-Many (WORM) Fileserver ist.
smartDax ist hier der Stellvertreter für die nicht so zahlreich vorhandenen Systeme, die mit integrierter Hardwareeinbindung daherkommen. Durch eine Hardwareeinbindung kann von einem Archivierungssystem auch die Aufgabe eines regelmäßigen Datenträgermedienaustausches übernommen werden auf den rein softwarebasierte Systeme keinen Einfluss besitzen. Da heute bekannte Datenträger immer einem gewissen Verfall und Verschleiß unterliegen, erhöht solch ein regelmäßiger Austausch die allgemeine Sicherheit des Systems vor Datenverlust und -verfälschung.

MIAMI Informations- und Archivierungssystem für multimediale Inhalte an der Universitäts- und Landesbibliothek (ULB) Münster

Aus der Programmbeschreibung (siehe [22]):
„MIAMI [...] ermöglicht es, digitale multimediale Objekte aus der Hochschule zu publizieren, bereitzustellen und [zu] archivieren [und ...] enthält nicht nur beschreibende Daten zu den Inhalten, sondern es archiviert auch die zugehörigen digitalen Texte, Dateien und Bilder."
MIAMI steht für recht allgemeingültige Systeme, die die Daten auf einer Weboberfläche anbieten. Diese Gruppe macht den Großteil der aktuellen Systeme aus. MIAMI vermag auf andere Daten als Texte und Bilder nur zu verweisen und macht sie nicht der angezeigten Webseite zu eigen.

EASY Archivierungssystem

EASY kann Texte in „elektronischen Mappen" (siehe [23]) ablegen und verschiedene Versionen der Mappen vorhalten. Diese Mappen entsprechen einer einfachen nichtrekursiven Gruppierungsmöglichkeit für die Daten.
Es ist ein System, welches auf Texte spezialisiert ist. Während der Erfassung kann ein Text in das PDF-Format[2] umgewandelt werden. EASY kann zudem die Daten in fest vorgegebene Kategorien einteilen.

AdAkta® Archivierungssystem für Praxis und Klinik

Aus der Produktbeschreibung (siehe [24]):
„AdAkta archiviert alle bildhaften Informationen, vom gescannten Dokument bis zum Digitalen Röntgen. [. . .] Der integrierte MediaManager verwaltet alle Platten und CDs vollautomatisch im Netz."
Bilder werden von AdAkta® stark komprimiert abgelegt. Nach der Auslieferung liegt aber wieder das Ursprungsformat vor, so dass in der Außenwirkung keine Formatübersetzung stattfindet, zudem kann diese nicht vom Benutzer gesteuert werden. Es dient hier als Beispiel für die Systeme, die Aufgaben in einem stark spezialisierten Bereich ausführen, hier ist es das Umfeld von Arztpraxis und Kliniken. Die erfassten Bilder können in feste Kategorien eingebunden werden. Das Programm kann ab einer gewissen Ausbaustufe einen CD-Wechsler ansteuern.

Artec Webbasiertes Archivierungssystem für das Evangelische Zentralarchiv (EZA) in Berlin

Aus der Produktbeschreibung (siehe [25]):
„Die zentrale Funktion des Archivierungssystems ist eine optimierte Volltextsuche."
Das System ist hauptsächlich ein Content-Management-System, bei dem mehrere Anwender auf die Gestaltung von Webseiten einwirken können. Zudem kann über den

[2] Portable Document Format der Adobe Systems Incorporated, siehe dazu [58].

Text auf den Webseiten eine Suche durchgeführt werden. Die eigentliche „Archivierung” geschieht durch Abspeichern der Seiten in einer PostgreSQL-Datenbank[3].

ArcFlow - Archivierungs- und Workflow-Management-System

ArcFlow erfasst digitale Dokumente direkt und andere Dokumente über ein Scanner-Interface und bietet diese ähnlich einem Microfiche-System[4] wieder an. Während der Erfassung wird eine Texterkennung (OCR[5]) durchgeführt und anhand von definierbaren Regeln eine Klassifizierung durchgeführt (z.B läßt sich „definieren, daß in den ersten 200 Zeichen des Dokumentes der Begriff Rechnung vorkommen muß”, siehe [27]). Über diese Klassifizierung oder eine Volltextsuche kann das Dokument wieder aufgefunden werden.
ArcFlow erlaubt dabei dem Benutzer, Daten frei in Kategorien einzuteilen und zu gruppieren. Es steht zudem Pate für Systeme mit integrierter Merkmalsextraktion. Das System ist jedoch auf Texte und Bilder beschränkt. Die Einbindung des Scanner-Interfaces alleine wird noch nicht als Hardwareeinbindung für den Zweck der Archivierung gewertet. Diese Hauptaufgabe wird auch ohne jene Hardware erfüllt.

DiVAN: Verteiltes audivisuelles Archivierungssystem

DiVAN konzentriert sich auf das Gebiet der Video-Archivierung. Dabei kann das Material in verschiedene Formate umgewandelt werden. Weitere Informationen können als Annotation zum Video abgelegt werden. Das Gesamtsystem arbeitet verteilt über mehrere Rechner. Zu DiVAN siehe [29].
DiVAN vertritt damit die sich momentan noch stark entwickelnde Gattung von hoch spezialisierten Systemen, welche für die Verarbeitung anderer als reiner Text- und statischen Bilddaten entworfen sind. Zudem verfügt es als eines von sehr wenigen Systemen über die Möglichkeit der automatischen Formatübersetzung. Es ist außerdem ein Ver-

[3] zu PostgreSQL siehe [26].
[4] Ein analoges Speichersystem, bei dem die Daten in bildhafter Form sehr stark verkleinert auf Mikrofilm vorliegen, siehe „Microfilm and Microfiche” [28] von Dalton.
[5] „Optical Character Recognition” bezeichnet Zeichen- und Texterkennung aus bildhaften Daten.

treter der eng gekoppelten verteilten Systeme, bei denen sich die Verteilung auf einen Rechnerverbund in netzwerktechnisch örtlicher Nähe beschränkt.

DSpace Organisation der MIT Libraries

DSpace ist hauptsächlich ein Webpräsentationssystem. Es benutzt Metainformationen gemäß des Dublin Core Formats[6] für die Verwaltung beliebiger Dokumente. Sowohl die Metadaten als auch die Daten werden vom System selbständig verwaltet und gespeichert. Auch eine Dokumenthistorie wird geführt. Die Speicherung wie auch die Verwaltung erfolgt dabei an mehreren, verteilten Orten. Zu DSpace siehe [30].

NDLTD - Networked Digital Library of Theses and Dissertations

NDLTD ist ein Zusammenschluss mehrerer Einrichtungen. Sie benutzt Metainformationen gemäß RFC1807[7] für die zentrale Verwaltung von Thesen und Dissertationen. Die eigentlichen Dokumente bleiben dabei bei den einsendenden Parteien abgelegt. NDLTD stellt also ein Verweissystem dar, wobei die Metainformationen zentral, die Dokumente aber stark verteilt[8] vorliegen. Zu NDLTD siehe [32].

2.3.2 Vergleichssysteme

CVS-Versionskontrolle

Das „Concurrent Versions System" (siehe [34]) ist eine Möglichkeit für den Einsatz von Versionskontrolle für beliebige Daten. Es ist ein zentral organisiertes System, das keine Verteilung benötigt.

CVS kann mit verschiedenen Versionen von Textdaten effizienter umgehen als mit anderen Daten, da es bei Textdaten nur die Änderungen von einer zur nächsten Version

[6] Zum Dublin Core Format siehe [31].

[7] Request for Comments Nummer 1807 – zum Text der RFC1807 siehe [33].

[8] „stark verteilt" meint hier, dass die Daten *nicht* unter einer administrativ einheitlichen Verwaltung organisiert sind, sondern diese auf den administrativ unabhängigen Webseiten der zusammengeschlossenen Einrichtungen abgelegt werden.

speichert, bei anderen Daten werden diese komplett in die Historie aufgenommen. Bei Textdaten kann es zudem teilweise automatisch eine neue Version aus mehreren gleichzeitig eingehenden Änderungen zu einer Datei zusammenfügen.

BitKeeper Quellcodemanagement

BitKeeper ist hauptsächlich ein Versionskontrollsystem wie CVS, ist dabei aber spezialisiert auf Quellcode-Texte. Ein besonderer Aspekt liegt auf der dem System innewohnenden Verteilung des Gesamtsystems. Zu BitKeeper siehe [35].

Network File System (NFS)

NFS ist ein Vertreter von verteilten Dateisystemen als stark begrenzte Untermenge eines Archivierungssystems. Es stellt freigegebene Verzeichnisse eines Wirtsrechners (NFS-Server) mehreren Kommensalrechnern[9] (NFS-Clients) zur Verfügung. Dabei werden die Benutzungsrechte beim Wirtsrechner geprüft. Zu NFS siehe [36].

ps2pdf-Konverter

Dieses Skript ist ein Teil vom Programmpaket „Ghostscript" (siehe [37]). Es soll hier stellvertretend für einzeln anwendbare Konvertierungsprogramme stehen. Es nimmt Daten im PostScript-Format[10] als Eingabe und gibt diese im PDF-Format[11] umgewandelt aus.

eDonkey2000

eDonkey2000 ist eines von vielen dezentralen Peer-to-Peer (P2P) Dateiaustauschprogrammen (FileSharing). Bei P2P-Netzen existieren keine oder wie im Fall von eDonkey2000 nur sehr schwache Hierarchien. Hier dient die Hierarchie zum Auffinden weiterer Peer-Rechner. Der Datenaustausch selbst wird selbständig zwischen gleichberech-

[9] Kommensalismus: Zusammenleben unter Nutzbringung für eine Seite ohne die Schädigung der einen oder der anderen Seite der Beziehung, siehe z.B. „Meyers grosses Taschenlexikon in 24 Bänden" [16].
[10] Zu Adobe PostScript siehe [59].
[11] Zum PDF-Format siehe [58].

tigten Rechnern abgewickelt.
Mit eDonkey2000 ist es möglich, beliebige Dateien zwischen allen teilnehmenden Tauschpartnern zu kopieren. Das System war ursprünglich für den Tausch großer Videodateien gedacht, kann inzwischen aber für Daten beliebigen Typs verwendet werden. Zum eDonkey-System siehe [38].

Audiogalaxy

Audiogalaxy war ein zentralisiertes Filesharingsystem, das sich auf den Tausch von Musik im mp3-Format spezialisiert hatte und die formatspezifischen Informationen durch Merkmalsextraktion direkt nutzen konnte. Der Dienst des dazu notwendigen zentralen Servers wurde inzwischen eingestellt. Zu Audiogalaxy siehe [39].

SQL-Datenbank

Eine SQL-Datenbank dient hier der Anschauung von möglicher freier Kategorisierung. Die in der Datenbank selbst speicherbaren Daten sind allerdings aus Effizienzgründen stark in der Größe begrenzt (siehe dazu die Abschnitte 2.2.2 und 4.4.1). Zu einem SQL-Tutorial siehe [40].

cdrdao (DOS)

Das Programm cdrdao aus dem Paket CDRDOS der Fa. Golden Hawk Technology steht hier Pate für ein System, das auf eine bestimmte Hardware (hier: SCSI CD-Brenner) angewiesen ist. Es ist ein hochspezialisiertes Programm das Abbilddaten von CD-ROMs mit Hilfe des CD-Brenners auf CD-Rohlinge schreiben kann. Zu CDRDOS siehe [41]. Es gibt gleichnamige Programme auf anderen Plattformen, die ähnliche Aufgaben erfüllen, welche aber nicht derartig stark spezialisiert sind.

AudioID

AudioID (siehe dazu [42]) ist ein in der Entwicklung befindliches Programm zum Extrahieren von Merkmalen aus Audiodaten mit dem Ziel, die errechneten Merkmale als

Identitätsschlüssel verwenden zu können. Ähnliche Systeme sind auch für den Videobereich in Entwicklung (siehe dazu Sietmanns „Licht ins Darknet - Multimediadaten suchen und finden” [8]).

2.4 Auswertung und Vergleich der Funktionalität

Tabelle 2.1 auf der nächsten Seite gibt eine Übersicht über die einzelnen untersuchten Funktionen der vorgestellten Systeme wieder.

Abbildung 2.4 zeigt die Einteilung der Systeme nach Verteilungs- und Spezialisierungsgrad und getrennt nach Archivierungs- und Vergleichssystemen. Der Großteil der vorgestellten, aber auch der dadurch vertretenen Systeme läßt sich nicht zu den verteilten Systemen rechnen. Es zeigt sich, dass mit zunehmender Verteilung auch der Grad der Spezialisierung bei Archivierungssystemen zunimmt.

Abbildung 2.5 auf Seite 28 zeigt die Einteilung der Systeme nach Automations- und Spezialisierungsgrad und getrennt nach Archivierungs- und Vergleichssystemen. Automation fasst hierbei die Funktionen Merkmalsextraktion und Formatübersetzung zusammen. Es zeigt sich hier, dass nur wenige Systeme eine benutzerfreundliche Auto-

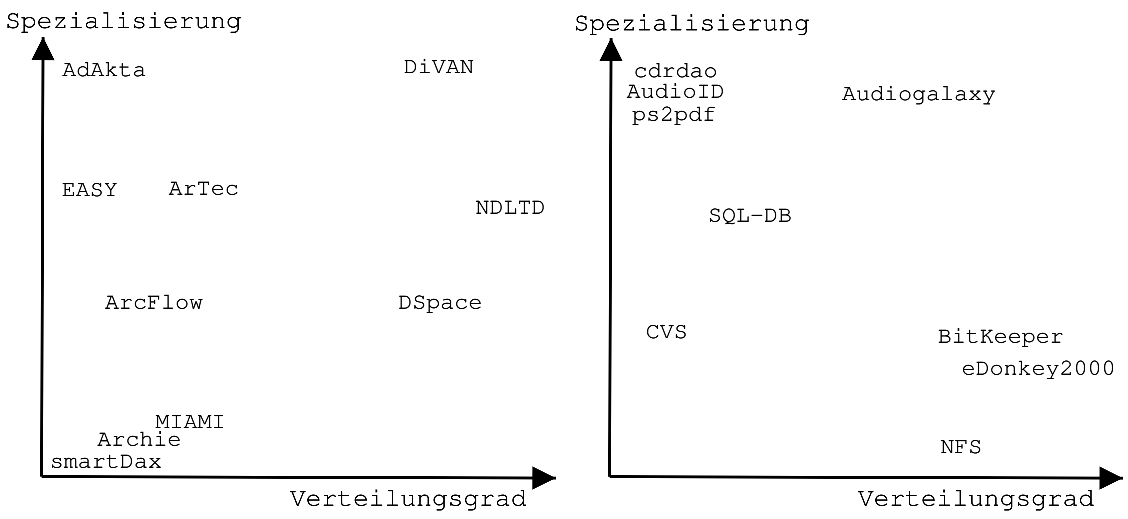

Abbildung 2.4: Verteilungs- zu Spezialisierungsgrad der vorgestellten Systeme: (links: Archivierungssysteme, rechts: Vergleichssysteme)

System	Verteiltes System	Versionskontrolle	Zugriffsverwaltung	Formatübersetzung	Kategorien	Spezialisierung	Merkmalsextraktion	Hardware
Archie	nein	ja	ja	nein	nein	nein	nein	nein
smartDax	nein	nein	ja	nein	nein	nein	nein	ja
MIAMI	nein	nein	ja	nein	nein	nein	nein	nein
EASY	nein	ja	ja	teilw.[a]	ja, fest	ja (Texte)	nein	nein
AdAkta®	nein	nein	nein	nein	ja, fest	ja (Bilder)	nein	ja[b]
Artec	nein	nein	ja	nein	nein	ja (Text)	nein	nein
ArcFlow	nein	nein	ja	nein	ja, frei	ja (Bild,Text)	ja (OCR)	nein
DiVAN	ja	nein	nein	ja	nein	ja (Video)	nein	nein
DSpace	ja	ja	ja	nein	ja, fest	nein	nein	nein
NDLTD	ja	nein	nein	nein	nein	ja (Texte mit RFC 1807-Beschreibung)	nein	nein
CVS	nein	ja	nein	nein	nein	teilw.[c]	nein	nein
BitKeeper	ja	ja	ja	nein	nein	ja (Quelltexte)	nein	nein
NFS-Dateisystem	nein	nein	ja	nein	nein	nein	nein	nein
ps2pdf-Konverter	nein	nein	nein	ja	nein	ja	nein	nein
P2P-Filesharing:								
– eDonkey2000	ja	nein	nein	nein	ja, fest	nein	teilw.[d]	nein
– Audiogalaxy	ja	nein	nein	nein	ja, fest	ja (mp3)	ja (id3[e])	nein
SQL-Datenbank	teilw.[f]	nein	ja	nein	ja, frei	ja, „kurze” Datenfelder	nein	nein
cdrdao (DOS)	nein	nein	nein	nein	nein	ja (iso[g])	nein	ja
AudioID	nein	nein	nein	nein	nein	ja (Audio)	ja	nein

Tabelle 2.1: Überblick über die Funktionen der vorgestellten Systeme

[a] EASY kann Texte bei der Erfassung in PDF umwandeln
[b] AdAkta® kann in der passenden Ausbaustufe direkt CD-Wechsler u.ä. ansteuern
[c] CVS kann mit Textdateien effizienter umgehen als mit Binärdaten. Letztere müssen dem System explizit bekanntgegeben werden
[d] eDonkey2000 kann aus bestimmten Daten (vornehmlich Videos) beispielsweise die Spieldauer auslesen
[e] id3 ist ein Metadatenformat, mit dem direkt in Musikdateien die zugehörige Information zu Interpret und Album gespeichert werden kann
[f] Verteilung ist nicht nötig für eine SQL-Datenbank, es gibt aber verteilte Exemplare
[g] ISO9660, oft kurz aber ungenau als „iso”-Format bezeichnet, ist das Standardformat für die Speicherung von Daten auf CD-ROM

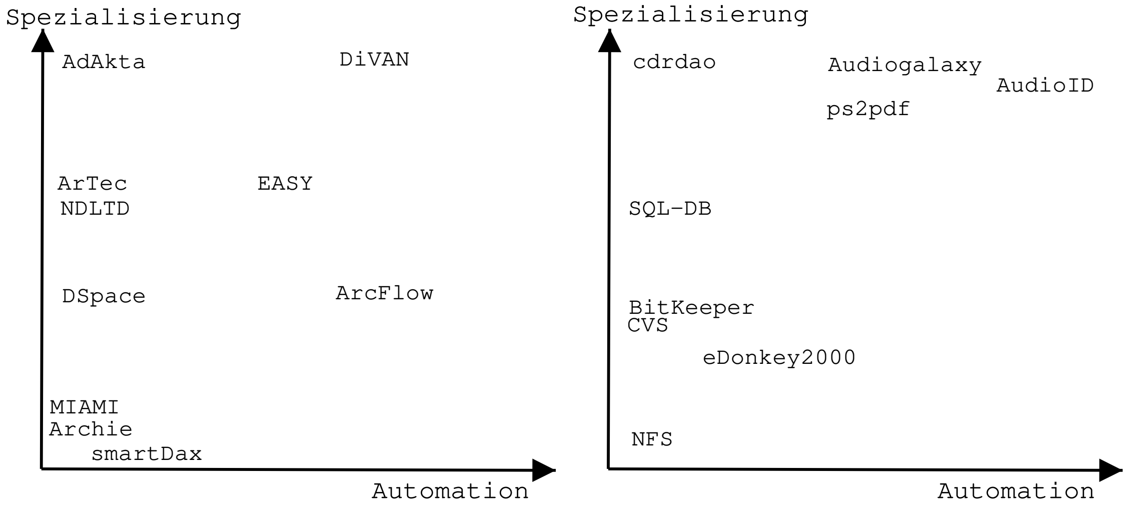

Abbildung 2.5: Automations- zu Spezialisierungsgrad der vorgestellten Systeme: (links: Archivierungssysteme, rechts: Vergleichssysteme)

mationsleistung anbieten. Bei den Systemen, die Automation bieten, geht dieses mit einer Spezialisierung des Gesamtsystems einher.

Abbildung 2.6 zeigt die Einteilung der Systeme nach Gestaltungsfreiheits- und Spezialisierungsgrad und getrennt nach Archivierungs- und Vergleichssystemen. Gestaltungsfreiheit fasst hierbei die Möglichkeiten der (freien) Kategorisierbarkeit und der

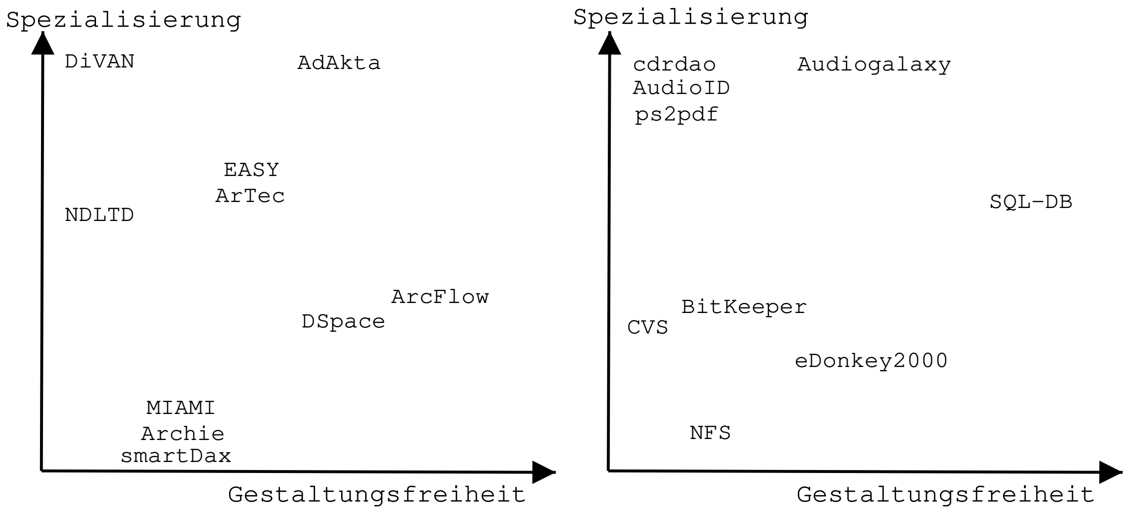

Abbildung 2.6: Gestaltungsfreiheits- zu Spezialisierungsgrad der vorgestellten Systeme: (links: Archivierungssysteme, rechts: Vergleichssysteme)

möglichen Rechtevergabe zusammen. Es gibt immerhin eine Reihe von Archivierungssystemen, die versuchen, den Benutzer durch Gestaltungsfreiheit zu unterstützen. Wie sich aus Tabelle 2.1 auf Seite 27 aber ergibt, geschieht dieses oft nur als Möglichkeit zur Zugriffsrechteverwaltung. Dieses unterstützt den Benutzer jedoch nicht bei der Organisation und beim Wiederauffinden der Daten. Auch hier zeigt sich, dass Gestaltungsfreiheit bei den untersuchten Systemen nur im Beisein von Spezialisierung und bei Verlust der Allgemeingültigkeit des Gesamtsystems vorkommt.

Keines der vorgestellten Systeme kann alle Anforderungen gemeinsam erfüllen, setzt man die zum Vergleich herangezogenen Funktionen als Anforderungen ein. Die Aufgabe der Arbeit und damit der folgenden Kapitel ist es nun, möglichst viele der positiven Eigenschaften unter Vermeiden der negativen Eigenschaften dieser Funktionen miteinander zu verbinden.
Besonders die Punkte der freien Kategorisierbarkeit und der automatischen Formatübersetzung wurden bislang nur unzureichend in Archivierungssysteme eingebaut. Diese sowie die Merkmalsextraktion konzentriert sich auf spezielle Datentypen und -formate. Mit dieser Spezialisierung geht bei den vorgestellten bestehenden Systemen auch immer eine Spezialisierung des gesamten Systems einher. Forschungsgegenstand wird daher auch sein, das Gesamtsystem mit möglicher Merkmalsextraktion auszustatten, *ohne* dieses dadurch in seiner Allgemeinheit einzuschränken.

2.5 Fazit aus dem aktuellen Entwicklungsstand

Es wurden Konzepte für verschiedene Bereiche eines verteilten Archivierungssystems vorgestellt. Dies betrifft die Gebiete der Gestaltung von Verteilten Systemen, der Konsistenzbewahrung in einem Verteilten System, der Datenhaltung und -langzeitarchivierung und der Zugriffsverwaltung. All diese vorgestellten Konzepte sind, teilweise aufeinander aufbauende, Einzelkonzepte, die jeweils versuchen, ein eng umgrenztes Problemgebiet zu behandeln. Einige geraten dabei mit anderen Konzepten in der Anwendbarkeit in

Konflikt. Teilweise werden auch neue Probleme mit der Anwendung eines der Konzepte geschaffen. Für viele Probleme gibt es zudem keine oder nicht weit genug reichende fertige Lösungskonzepte.

Dann wurden aktuelle Archivierungssysteme sowie für die einzelnen Funktionsgebiete ausgewählte Vergleichssysteme vorgestellt und anhand ihrer Funktionalität verglichen. Wie dieser Vergleich gezeigt hat, bereitet es existierenden Systemen anscheinend Probleme, die verschiedenen Wünsche an die funktionellen Möglichkeiten der Systeme zu erfüllen. Die folgenden Kapitel widmen sich daher diesen Wünschen, den Möglichkeiten und dem Zusammenspiel der hier vorgestellten und teilweise auch neuer Konzepte.

Kapitel 3

Anforderungen

Es soll ein System entstehen, mit dessen Hilfe sich Daten über längere Zeiträume verwalten lassen. Das System soll dabei den prinzipiellen Anforderungen eines herkömmlichen Archivs genügen. Diese herkömmlichen Aufgaben sind das Entgegennehmen, Ablegen und sichere Aufbewahren und das spätere Zurverfügungstellen von Daten. Dies umfasst neben der Aufnahme der zu archivierenden Daten auch die Aufnahme von Metadaten, die die archivierten Daten beschreiben.
Diese Grundanforderungen an ein Archiv erfüllen mehr oder weniger strikt bereits die im Übersichtskapitel vorgestellten Systeme in unterschiedlicher Ausprägung. Neu hingegen ist die Gesamtheit und der Umfang der Anforderungen, die in den folgenden, die Anforderungen konkretisierenden Abschnitten aufgeführt werden und die das System ebenfalls erfüllen soll.

3.1 Grunddefinitionen

Es werden die Begriffe „Daten" und „Metadaten" wie folgt verwendet:

$$\text{Daten} : \text{Die zu archivierenden Daten} \tag{3.1}$$

$$\text{Metadaten} : \text{Attribute, welche die zu archivierenden Daten beschreiben} \tag{3.2}$$

Die Metadaten können zum Teil aus den zu archivierenden Daten direkt gewonnen werden.

Zur Vereinfachung und Lesbarkeit wird folgende Abkürzung für zeitlich veränderliche Relationen verwendet:

$$\begin{gathered}\text{Sei } X(t) \text{ eine zeitlich veränderliche Relation mit Zeitparameter } t,\\ \text{so sei abkürzend geschrieben: } X \equiv X(t) \text{ und } X' \equiv X(t+1)\end{gathered} \tag{3.3}$$

Es bezeichne folgend $\mathcal{T}_X$ die Domäne zu X. Mit $\mathcal{P}(x)$ wird die Potenzmenge von x bezeichnet.

Zu jeder Domäne Y sei ferner ein Element $\perp$ disjunkt und kleiner aller anderen darin enthaltenen Elemente gegeben, so dass

$$Y^{\perp} := Y \cup \{\perp\} \tag{3.4}$$

die zu Y zugehörige gehobene Domäne[1] bezeichne.

Folgende grundlegende Domänen seien im System definiert:

$$\mathcal{T}_D \;:\; \text{Domäne der Daten}^2, \mathcal{T}_D = \{b^\star | b \in (2^8)\} \tag{3.5}$$

$$\mathcal{T}_I \;:\; \text{Domäne der Datenidentifikatoren} \tag{3.6}$$

$$\mathcal{T}_T \;:\; \text{Domäne der Metadatenidentifikatoren} \tag{3.7}$$

$$\mathcal{T}_W \;:\; \text{Domäne aller Metadaten} \tag{3.8}$$

$$\mathcal{T}_{W_t} \subseteq \mathcal{T}_W \;:\; \text{Domäne der Metadaten } t \in \mathcal{T}_T \tag{3.9}$$

Es sei weiterhin

$$T \subseteq \mathcal{T}_T \tag{3.10}$$

die im System definierte und zeitlich veränderbare Menge der aktuell[3] dort vorhandenen

[1] engl.: „lifted domain", siehe [60]: Diese Definition wird folgend benutzt, um den Rückgabewert bei einer fehlgeschlagenen oder nicht erfolgten Berechnung innerhalb einer Methode eindeutig zu kennzeichnen und so trotzdem den Wertebereich der betreffenden Methode eindeutig zu definieren.

[2] Daten seien wiederum gegeben als eine Menge von Bytes nach der heute üblichen Definition: 1 Byte bestehe aus 8 Bit, kann also 2^8 Zustände annehmen

[3] Die Formulierung dieser und auch folgender Mengen und Relationen als „zeitlich veränderbar" und im System „aktuell" definiert basiert auf den Aussagen von Lang und Lockemann in „Datenbankeinsatz" [2]. Die so definierten Mengen und Relationen beinhalten die jeweils aktuelle Systemsicht und werden durch Zuweisungen in der Form von Aussage 3.3 verändert und damit aktualisiert.

Metadatenidentifikatoren und

$$I \subseteq \mathcal{T}_I \tag{3.11}$$

die im System definierte und zeitlich veränderbare Menge der aktuell dort vorhandenen Datenidentifikatoren sowie

$$W_t \subseteq \mathcal{T}_{W_t} \text{ für } t \in T \tag{3.12}$$

die im System aktuell vorhandenen Werte der Metadaten t. Es sei außerdem die Menge der Kriterien definiert als

$$\begin{aligned} K &: \mathcal{T}_T \times \mathcal{T}_W \\ &= \{(t, w) | t \in \mathcal{T}_T \wedge w \in \mathcal{T}_{W_t}\} \end{aligned} \tag{3.13}$$

K^* bezeichne eine im System definierte und zeitlich veränderbare Menge mit mengenwertigen Elementen $k \subseteq K$. Die den Kriterien zugeordneten Wertelemente $w \in \mathcal{T}_{W_t}$ seien anhand der enthaltenen Metadatendomäne $t \in \mathcal{T}_T$ eindeutig mit $k_t = w$ für $k \in K^*$ zu bestimmen. Die $k \in K^*$ seien zudem durchnummeriert, so dass sie durch k_i für $1 \leq i \leq \arg\max_j k_j$ ebenfalls eindeutig bestimmt sind. Dies kann ohne Beschränkung der Allgemeinheit erfolgen: Im Konfliktfall werde t umbenannt.

So läßt sich die Datenbasis $\mathfrak{D}$ vom Typ $\mathcal{T}_{\mathfrak{D}}$, die dem System zugrundeliegen soll, nebst darin enthaltener Elemente δ definieren als

$$\mathfrak{D} \in \mathcal{P}(\delta) \wedge \mathfrak{D} \subseteq \delta \quad \text{mit} \tag{3.14}$$

$$\delta \in \mathcal{T}_D \times \bigtimes_{t \in \mathcal{T}_T} \mathcal{T}_{W_t} \tag{3.15}$$

Die Elemente sind also aus den eigentlichen Daten und den aus den Metadaten stammenden Werten aufgebaut, die diese Daten beschreiben. Auf die Komponenten eines Tupels δ wird folgend direkt mit $\delta.t$ für die Einträge aus $\mathcal{T}_{W_t}$ sowie mit $\delta.d$ für den Eintrag aus $\mathcal{T}_D$ Bezug genommen. Ferner bezeichne $i \in I$ die einzelnen Elemente $\delta_i \in \mathfrak{D}$ eindeutig, also $I = \{i \in \mathcal{T}_I | \exists x \in \mathfrak{D} : x = \delta_i\}$.

Es läßt sich nun die Grundfunktionalität definieren mit den Methoden ident und insert:

$$\begin{aligned} \text{ident} &: \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_D \to \mathcal{T}_I \qquad (3.16) \\ \text{ident}(\mathfrak{D}, d) &= i \\ \text{mit} & \quad i \in \mathcal{T}_I \wedge \begin{cases} I \cap \{i\} = \emptyset & \text{falls} \quad \forall j \in I, \delta_j \in \mathfrak{D} : \delta_j.d \neq d \\ \delta_i.d = d & \text{sonst} \end{cases} \end{aligned}$$

$$\begin{aligned} \text{insert} &: \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_D \times \mathcal{T}_{K^*} \to \mathcal{T}_{\mathfrak{D}}{}^{\perp} \qquad (3.17) \\ \text{insert}(\mathfrak{D}, d, k) &= \mathfrak{D}' \quad \text{mit} \\ & \begin{cases} \mathfrak{D}' := \mathfrak{D} \cup \delta_i \quad \text{mit} & \\ & i := \text{ident}(d) \\ & \wedge \quad \delta_i.d := d \\ & \wedge \quad \forall j : (1 \leq j \leq \arg\max_n k_n \wedge \exists a_j, b_j : \\ & \qquad (k_j = (a_j, b_j) : \delta_i.a_j := b_j)) \\ & \wedge \quad I' := I \cup \{i\} \\ & \text{falls} \quad I \cap \{i\} = \emptyset \\ \mathfrak{D}' := \mathfrak{D} & \text{sonst} \end{cases} \end{aligned}$$

3.2 Verlässlichkeit

Die Benutzer des Systems sollen sich auf das System verlassen können. Dies können sie, wenn gewährleistet ist, dass einmal eingegebene Daten sicher vor Verlust oder Verfälschung bewahrt werden und nur berechtigten Personen der Zugriff darauf gestattet wird.

3.2.1 Datensicherheit

Die erste Forderung zur Datensicherheit ergibt sich aus dem Wunsch, dass einmal in das System aufgenommene Daten über eine Suche mit den zugehörigen Metadaten oder auch Teilen davon wieder ausgeliefert werden können. Diese Forderung wird in den Definitionen 3.18 und 3.19 ausgedrückt.

Die Gleichung 3.19 drückt dabei implizit die zweite Forderung zur Datensicherheit, die der Unverfälschtheit der Daten, aus, indem genau das unverfälschte d wieder ausgeliefert wird, das mittels $\mathsf{insert}(\mathfrak{D}, d, \dots)$ einst in das System aufgenommen wurde. Dies ergibt sich aus der Forderung, dass $\forall x : \delta_x.d$ einzig mit der Methode insert verändert (nämlich einmalig vergeben) werden darf.

$$\begin{aligned} \text{search} &: \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_{K^*} \to \mathcal{P}(I) && (3.18)\\ \text{search}(\mathfrak{D}, k) &= \{i \in I | \forall j : (1 \leq j \leq \arg\max_n k_n \wedge \exists a_j, b_j : (k_j = (a_j, b_j) \wedge \exists \delta_i \in \mathfrak{D} : \delta_i.a_j = b_j))\} \end{aligned}$$

$$\begin{aligned} \text{deliver} &: \mathcal{T}_{\mathfrak{D}} \times I \to \mathcal{T}_D{}^{\perp} && (3.19)\\ \text{deliver}(\mathfrak{D}, i) &= \begin{cases} \delta_i.d & \text{falls} \quad \delta_i \in \mathfrak{D} \\ \perp & \text{sonst} \end{cases} \end{aligned}$$

3.2.2 Rechteverwaltung

Auf die im System gespeicherten Daten soll nicht jeder Benutzer zugreifen können, wenn die Administration als zur Rechtebestimmung ermächtigter Benutzerkreis ihm nicht dazu ein Recht einräumt. Ebenso darf das System neue Daten nur von Benutzern mit entsprechendem Recht aufnehmen. Diese Benutzerrechte sollen zur leichten Verwaltbarkeit von größeren Personenkreisen in Benutzergruppen organisierbar sein.[4]
Auf eine daneben mögliche Umgehung des Systems wird in dieser Arbeit nicht eingegangen, da nur die grundsätzliche Machbarkeit der Rechteverwaltung im Zuge eines Archivierungssystems, nicht aber die detaillierte Ausprägung einer entsprechenden Verwaltung zum Thema gehört. Es werden somit nur folgende Mindestforderungen als Demonstration der Machbarkeit der Rechteverwaltung erhoben:

[4] Es handelt sich um eine RBAC Rechteverwaltung (role based access control), die durch MAC (mandatory access control) ausgeführt wird, siehe dazu auch Colsmanns Projektbericht [17], „Getting a Grip on Access Control Terms" [18] aus TECS und Osborn et al. [19].

Es seien folgende Mengen für die Verwaltung der Benutzer definiert:

$$\begin{aligned} \mathfrak{U} &: \text{ Menge der Benutzer} && (3.20)\\ U &: \text{ Menge der Benutzergruppen} && (3.21)\\ &= \{u | u \subseteq \mathfrak{U}\} \\ \mathfrak{R}^{out} &: \text{ Menge der Auslieferungsrechte} && (3.22)\\ &= \{\text{darf suchen}, \text{darf ausliefern}\} \\ \mathfrak{R}^{in} &: \text{ Menge der Einlieferungsrechte} && (3.23)\\ &= \{\text{darf einfügen}\} \end{aligned}$$

Ferner seien folgende zeitlich veränderliche Relationen der Benutzerrechte aktuell im System definiert:

$$\begin{aligned} R^{out} &\subseteq I \times U \times \mathfrak{R}^{out} = \{(i, u, r) | i \in I \wedge u \in U \wedge r \in \mathfrak{R}^{out}\} && (3.24)\\ R^{in} &\subseteq U \times \mathfrak{R}^{in} = \{(u, r) | u \in U \wedge r \in \mathfrak{R}^{in}\} && (3.25) \end{aligned}$$

Weiterhin gehören folgende Funktionen zum Setzen der Rechte zum System:

$$\begin{aligned} \text{set}_{R^{out}} &: \mathcal{T}_{R^{out}} \times I \times \mathcal{T}_U \times \mathfrak{R}^{out} \to \mathcal{T}_{R^{out}} && (3.26)\\ \text{set}_{R^{out}}(R^{out}, i, u, r) &= R^{out\prime} \quad \text{mit} \\ & \forall x : R^{out\prime} := (R^{out} \setminus \{(i, u, x)\}) \cup \{(i, u, r)\} \end{aligned}$$

$$\begin{aligned} \text{set}_{R^{in}} &: \mathcal{T}_{R^{in}} \times \mathcal{T}_U \times \mathfrak{R}^{in} \to \mathcal{T}_{R^{in}} && (3.27)\\ \text{set}_{R^{in}}(R^{in}, u, r) &= R^{in\prime} \quad \text{mit} \\ & \forall x : R^{in\prime} := (R^{in} \setminus \{(u, x)\}) \cup \{(u, r)\} \end{aligned}$$

Somit können die bekannten Funktionen erweitert werden zu:

$$\mathrm{insert}_R \quad : \quad \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_{\mathfrak{U}} \times \mathcal{T}_D \times \mathcal{T}_{K^*} \rightarrow \mathcal{T}_{\mathfrak{D}}{}^{\perp} \tag{3.28}$$

$$\mathrm{insert}_R(\mathfrak{D}, u, d, k) \quad = \quad \begin{cases} \mathrm{insert}(\mathfrak{D}, d, k) & \text{falls} \quad \exists\, g : (u \in g \wedge \\ & (g, \text{darf einfügen}) \in R^{in}) \\ \perp & \text{sonst} \end{cases}$$

$$\mathrm{search}_R \quad : \quad \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_{\mathfrak{U}} \times \mathcal{T}_{K^*} \rightarrow \mathcal{P}(I)$$

$$\begin{aligned} \mathrm{search}_R(\mathfrak{D}, u, k) \quad = \quad & \{i | i \in \mathrm{search}(\mathfrak{D}, k) \\ & \wedge \exists\, g, r : (u \in g \\ & \wedge \{\text{darf suchen}\} \subseteq r \\ & \wedge (i, g, r) \in R^{out})\} \end{aligned} \tag{3.29}$$

$$\mathrm{deliver}_R \quad : \quad \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_{\mathfrak{U}} \times I \rightarrow \mathcal{T}_D{}^{\perp} \tag{3.30}$$

$$\mathrm{deliver}_R(\mathfrak{D}, u, i) \quad = \quad \begin{cases} \mathrm{deliver}(\mathfrak{D}, i) & \text{falls} \quad \exists\, g, r : (u \in g \\ & \wedge \{\text{darf ausliefern}\} \subseteq r \\ & \wedge (i, g, r) \in R^{out}) \\ \perp & \text{sonst} \end{cases}$$

3.3 Skalierbarkeit

Ein besonderer Schwerpunkt liegt auf der Skalierbarkeit des Systems. Das bedeutet, dass es auch durch die Aufnahme von vielen Datensätzen durch eine Ausweitung auf mehrere Rechensysteme benutzbar bleiben soll. Skalierbarkeit bedeutet ebenfalls eine erhöhte Effizienz bei der Verwendung mehrerer Rechensysteme, wenn die Datenmenge nicht im gleichen Ausmaß wie die Menge der Verarbeitungskapazität steigt. Das Gesamtsystem soll also nicht nur im kleinen sondern auch im großen Maßstab verwendbar sein.

3.3.1 Verteiltes System

Für die Skalierbarkeit spielt die mögliche Verteilung des Systems auf mehr als einen Rechner eine große Rolle. Die Verteilung dient zum einen der Anpassung an größere Datenmengenanforderungen im Laufe des Systemeinsatzes, zum anderen der Verringerung des Datenaufkommens zwischen zwei oder mehr Standorten. Als Standort wird hier ein von einem anderen Standort entfernt liegender Einsatzort des Systems gesehen, wie dies beispielsweise bei Firmen mit mehr als einer Filiale der Fall ist. Die Datenverbindung zwischen zwei Standorten zeichnet sich im allgemeinen dadurch aus, dass sie weniger leistungsfähig als die lokal an einem Standort vorhandenen Datenverbindungen sind. Das System soll daher

1) Daten nur bei Bedarf von einem an einen anderen Standort übermitteln,

2) Daten nach Bedarf nur einmal zwischen Standorten austauschen und Mehrfachübermittlungen vermeiden und

3) mit dem möglichen Ausfall sowie der Wiederaufnahme der Datenverbindung zwischen zwei Standorten rechnen und eine Weiterarbeit während der Trennung mit dem verbliebenen Systemteilen ermöglichen sowie nach Wiederherstellung der Verbindung die vollständige Funktionalität des Gesamtsystems wieder herstellen.

Neben der Behandlung von entfernten Standorten soll das System auch verteilt in einem lokalen Netzwerk funktionieren.
Es wird daher eine Menge der Standorte definiert als

$$\mathfrak{S} \quad : \quad \text{Menge der Standorte} \tag{3.31}$$

Weiterhin werden die Standortinformationen in den Metadaten aufgenommen. Dort werden alle Standorte, an denen das δ_i gespeichert wird, erfasst:

$$\begin{aligned} \forall i: \quad & \delta_i.\text{standort} \in \mathcal{T}_{W_\text{standort}} \\ \text{mit} \quad & \mathcal{T}_{W_\text{standort}} \equiv \mathcal{P}(\mathfrak{S}) \end{aligned} \tag{3.32}$$

Punkt 1) ist absichtlich sehr weiträumig definiert, je nach Interpretation von „Bedarf". Für das Erfüllen der Anforderungen reicht die Festlegung der verwendeten Interpretation und die dementsprechende Ausführung der Funktionalität. Eine Möglichkeit ist das Belassen der Daten am Einfügeort. Damit wird im Zuge von insert die Aktion aus Gleichung 3.33 ausgeführt. Durch eine Erweiterung von insert um einen Parameter $s \in \mathfrak{S}$ für den Zielort läßt sich auch das Speichern der Daten an jenem Zielort abbilden[5].

$$\begin{aligned} \delta_i.\mathrm{standort}' &:= \delta_i.\mathrm{standort} \cup \{s\} \\ \text{mit} \quad & s \in \mathfrak{S} : s \text{ bezeichnet lokalen Standort} \end{aligned} \tag{3.33}$$

Punkt 2) kann erreicht werden, wenn nach einer Übermittlung der Daten von einem anderen Standort diese Daten zusätzlich am lokalen Standort gespeichert werden. Die Aktion aus Gleichung 3.33 muss somit im Zuge von deliver ausgeführt werden.

Punkt 3) stellt einen Kernpunkt des zu erstellenden Systems dar. Solange die Verbindung zwischen einem oder mehreren Standorten nicht funktioniert muss jeder Systemteil für sich die lokal vorgenommenen Änderungen temporär zwischenspeichern und beim Wiederverbinden mit den restlichen Systemteilen synchronisieren. Damit Einfügeoperationen konfliktfrei stattfinden können, muss I und die Funktion ident derart gestaltet werden, dass sich keine Konflikte zwischen den zwischenzeitlich getrennten Standorten ergeben können. Die Funktionalität darf dergestalt eingeschränkt werden, dass Suchoperationen nur auf dem Stand der Daten bei der Trennung zuzüglich lokaler Erweiterungen und Auslieferoperationen nur mit lokal vorhandenen Daten möglich sein müssen.

3.3.2 Versionskontrolle

Das System soll nach entsprechender Benutzereingabe mehrere Versionen eines Datensatzes in einer den Daten zugeordneten Historie ablegen und wieder ausliefern können. Diese Historie kann innerhalb des normalen Metadatenraums geführt werden. Die Such-

[5] Neben dem Eintragen in die Metadaten muss die Speicherung am Ort s tatsächlich zuvor vorgenommen werden

funktion muss nicht erweitert werden. Es wird jedoch eine Erweiterung zu insert mit einer Hilfsfunktion $\mathsf{version}$ definiert als

$$\begin{aligned} \mathsf{version} &: \mathcal{T}_{\mathfrak{D}} \times I \rightarrow \mathcal{T}_{W_{\text{version}}} && (3.34) \\ \mathsf{version}(\mathfrak{D}, i) &= \begin{cases} \delta_i.\text{version} & \text{falls} \quad \exists\, \delta_i \in \mathfrak{D} \wedge \exists\, \delta_i.\text{version} \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

$$\begin{aligned} \mathsf{insert}_V &: \mathcal{T}_{\mathfrak{D}} \times I \times \mathcal{T}_D \times \mathcal{T}_{K^*} \rightarrow \mathcal{T}_{\mathfrak{D}} && (3.35) \\ \mathsf{insert}_V(\mathfrak{D}, j, d, k) &= \mathsf{insert}(\mathfrak{D}, d, k \cup k_{\text{version}} \cup k_{\text{vorgänger}}) \\ \text{mit} &\quad k_{\text{version}} := (\text{"version"}, \mathsf{version}(\mathfrak{D}, j) + 1) \\ \wedge &\quad k_{\text{vorgänger}} := (\text{"vorgänger"}, j) \end{aligned}$$

3.3.3 Hardwareeinbindung

Das System soll auf eine mögliche Hardwareeinbindung vorbereitet sein, indem es eine Schnittstelle für das Auslösen von Schreib- und Leseaktionen sowie für den Bericht über deren Fertigstellung anbietet. Gedacht ist dabei an die mögliche Steuerung eines Datensicherungsgerätes wie eines CD-Brenners oder eines Bandlaufwerks, welche zu gegebener Zeit eine Sicherung von neu angefallenen Daten, aber auch von älteren Speichermedien zwecks Erhalt der darauf abgelegten Daten, vornehmen können sollen.

3.4 Automation

Ein weiterer Schwerpunkt neben der Skalierbarkeit wird mit der Automation gesetzt. Die untersuchten Systeme zeichnen sich teilweise durch besondere Automation aus. Dieses geht dort aber einher mit der Spezialisierung des Gesamtsystems auf einen oder wenige Datentypen bzw. -formate. Die in das neue System einzubauenden Automationsteile sollen die allgemeine Anwendbarkeit des Gesamtsystems hingegen *nicht* einschränken.

3.4.1 Formatübersetzung

Ein Bereich, in dem dem Benutzer durch Automation Arbeit abgenommen werden kann, ist die Übersetzung von dem Datenformat, in dem die Daten abgelegt wurden, in eines, welches der Benutzer gerade benötigt oder verwenden kann. Die Formatübersetzung soll so gestaltet werden, dass sie immer wieder erweitert werden kann und nicht auf die zum Erstellungszeitpunkt implementierten Formate beschränkt ist. Wird vom Benutzer keine Formatumwandlung explizit gewünscht, so soll das System die Weiterarbeit auch mit einem dem System unbekannten Format ermöglichen.
Es sei die Menge der aktuell im System bekannten Formate gegeben als

$$\mathfrak{F} \subseteq \mathcal{T}_{W_{\mathrm{format}}} \tag{3.36}$$

Beim Einfügen werde nach Möglichkeit $\delta_i.\mathrm{format}$ gesetzt. Dieses kann durch automatisches Erkennen des Formats oder manuell durch den Benutzer als angegebenes Kriterium geschehen. Ist die Möglichkeit nicht gegeben oder erwünscht, so soll das System trotzdem mit dem unbekannten Datenformat weiterarbeiten können mit der Einschränkung, dieses eventuell nicht übersetzen zu können.
Zugehörig ist eine Relation, die die Übersetzbarkeit von einem in ein anderes Format kennzeichnet. Es werde damit verzeichnet, welche Übersetzungsfunktionen $\mathsf{translate}_y^x$ dem System bekannt sind:

$$\begin{aligned} F &\subseteq \mathfrak{F} \times \mathfrak{F} \qquad (3.37) \\ &= \{(p,q) | p,q \in \mathfrak{F} \wedge \exists\ \mathsf{translate}_q^p : \mathcal{T}_D \to \mathcal{T}_D \text{ Übersetzung von } p \text{ nach } q\} \end{aligned}$$

Somit kann $\mathsf{deliver}$ erweitert werden zu:

$$\begin{aligned} \mathsf{deliver}_F &: \mathcal{T}_{\mathfrak{D}} \times \mathcal{T}_{\mathfrak{F}} \times I \to \mathcal{T}_D{}^{\perp} \qquad (3.38) \\ \mathsf{deliver}_F(\mathfrak{D}, f, i) &= \begin{cases} \mathsf{translate}_f^e(\mathsf{deliver}(\mathfrak{D}, i)) & \text{falls} \quad (e,f) \in F \wedge \delta_i.\mathrm{format} = e \\ \perp & \text{sonst} \end{cases} \end{aligned}$$

Ferner soll das System geeignete Methoden zum Erweitern von F durch Zufügen weiterer Übersetzungsfunktionen $\mathsf{translate}_y^x$ anbieten, damit neue Übersetzungsmöglichkeiten in das System eingebunden werden können.

3.4.2 Merkmalsextraktion

Ein weiterer Bereich für das Abnehmen von Arbeit ist die Merkmalsextraktion. Aus den dem System bekannten Datenformaten sollen sich mögliche Metadaten automatisiert auslesen lassen. Da Merkmalsextraktionen möglicherweise nicht immer perfekt funktionieren, soll dem Benutzer eine Nachbearbeitung der so gewonnenen Metadaten nicht verweigert werden. Bei dem System unbekannten Formaten soll diese Funktion das Weiterarbeiten ohne die Automatikunterstützung mit dem unbekannten Format ermöglichen.

Es sei die Menge der Daten in bekannten Formaten, aus denen Merkmale automatisch gewonnen werden können, definiert als

$$E \subseteq \mathcal{T}_D \tag{3.39}$$

Zugehörig sei eine Methode $\mathsf{extract}$, welche die Merkmale passend für das Einfügen mit insert vorbereiten kann. Die Kontrolle, ob und welche der extrahierten Merkmale tatsächlich eingefügt werden, obliegt weiterhin dem Benutzer:

$$\begin{aligned} \mathsf{extract} &: \mathcal{T}_D \to \mathcal{T}_{K^*}{}^{\perp} && (3.40) \\ \mathsf{extract}(d) &= \begin{cases} \{k_l | k_l \in K : \text{extrahiertes Kriterium}\} & \text{falls} \quad d \in E \\ \perp & \text{sonst} \end{cases} \end{aligned}$$

Ferner soll das System geeignete Methoden zum Erweitern von E und $\mathsf{extract}$ anbieten, damit neuentstandene Extraktionsmöglichkeiten in das System eingebunden werden können.

3.5 Gestaltungsfreiheit

Der dritte Schwerpunkt des Systems soll eine weitgehende Gestaltungsfreiheit der Benutzer gemäß folgender Abschnitte sein.

3.5.1 Kategorisierbarkeit

Die Daten sollen sich in Kategorien einordnen lassen. Diese sollen dazu dienen, die Daten in geordneter Weise wiederaufzufinden. Dabei sollen

- sinnvolle feste Kategorien vorgegeben werden, welche sich aber erweitern lassen sowie
- Daten frei gruppierbar im Sinne von freier Festlegung von Gruppen und Gruppenzugehörigkeit sein.

Dazu sei die Menge der Kategorien $\mathfrak{C}$ definiert. Die Zuordnung zwischen den Kategorien soll gemäß folgender zeitlich veränderbarer und im System aktueller Relation festgelegt werden:

$$C \subseteq \mathfrak{C} \times \mathfrak{C} = \{(c,b)|c,b \in \mathfrak{C} \wedge c \text{ ist Oberkategorie zu } b\} \tag{3.41}$$

Dabei werde C mit geeigneten Kategorien initialisiert.
Die Kategorienzuordnung bilde einen Teil der Metadaten, so dass $\forall i : \delta_i.\text{category} \in \mathcal{T}_{W_{\text{category}}}$ die jeweilige Zuordnung anzeige.
Die Kategorienerweiterung soll mit der Methode $\mathsf{deducecategory}$ geschehen:

$$\begin{aligned} \mathsf{deducecategory} &: \mathcal{T}_C \times \mathcal{T}_{\mathfrak{C}} \times \mathcal{T}_{\mathfrak{C}} \to \mathcal{T}_C \\ \mathsf{deducecategory}(C,c,b) &= C' \quad \text{mit} \quad C' := C \cup \{(c,b)\} \end{aligned} \tag{3.42}$$

Für den sinnvollen Einsatz von Kategorien muss die Suchfunktion abgewandelt werden, damit nicht nur exakt gleiche Kategorien, sondern auch Unterkategorien davon im Suchergebnis auftauchen.

Für die Gruppierbarkeit seien die Menge der Gruppen $\mathfrak{G}$ und die aktuell im System bekannten Gruppen $G \subseteq \mathfrak{G}$ definiert. Die Gruppenzugehörigkeit werde als Teil der Metadaten geführt, so dass $\forall i : \delta_i.\text{group} \in \mathcal{T}_{W_{\text{group}}}$ die jeweilige Zugehörigkeit angebe.

Zur Erweiterung der im System vorhanden Gruppen werde die Methode $\mathsf{creategroup}$ verwendet:

$$\begin{aligned} \mathsf{creategroup} &: \mathcal{T}_G \times \mathcal{T}_{\mathfrak{G}} \to \mathcal{T}_G \\ \mathsf{creategroup}(G,g) &= \begin{cases} G' \quad \text{mit} \quad G' := G \cup \{g\} & \text{falls} \quad g \notin G \\ G & \text{sonst} \end{cases} \end{aligned} \tag{3.43}$$

Die Zugehörigkeit zu einer Gruppe $g \in G$ soll somit durch ein Kriterium ("group", g) im Zuge des Einfügevorgangs deklariert werden können.

3.5.2 Allgemeinheit bei Spezialisierbarkeit

Das System soll mit Daten allgemeiner Art umgehen können und nicht beschränkt sein auf die Verwendung nur bestimmter Datentypen bzw. -formate. Ist aber das Datenformat bekannt, so soll das System darauf mit der Erweiterung von Benutzungsmöglichkeiten (beispielsweise weitere Suchkriterien) oder der Erweiterung der Automatisierung reagieren.
Weiterhin soll das System so gestaltet sein, dass weitere Datenformate als Spezialisierung dem System bekannt gegeben und daraufhin auch mit weiteren Programmmodulen auf die Ausgestaltung der neuen Spezialisierung Einfluss genommen werden kann.

3.6 Abschluss der Anforderungen

Es wurden die einzelnen Aufgabenfelder und Anforderungen an ein verteiltes Archivierungssystem vorgestellt. Diese umfassen

- Verlässlichkeit,
- Skalierbarkeit,
- Automation und
- Gestaltungsfreiheit.

Zu der Motivation und Begründung wurde jeweils eine möglichst exakte Formulierung zu jeder Teilanforderung gegeben.

Das System wird die Anforderungen erfüllen, wenn alle Teilanforderungen aus diesem Kapitel gemeinsam und gleichzeitig erfüllt werden.

Das System werde somit zusammengesetzt aus der Datenbasis, die Daten und Metadaten beinhaltet sowie der Rechte-, Format- und Gruppenbasis nebst zugehöriger Methoden als Teil des Datenverwaltungssystems. Die Versionsverwaltung geschehe als Teil der Metadaten und werde somit innerhalb der Datenbasis geschaffen.

Kapitel 4

Konzeption

4.1 Überblick

In diesem Kapitel wird ein System konzipiert, das das Archivieren von Daten ermöglicht, diese Aufgabe als Verteiltes System durchführen kann und dabei die weiteren Anforderungen aus Kapitel 3 erfüllt.
Zunächst wird das generelle Architekturkonzept vorgestellt. Darin werden neben einem Architekturüberblick zu aufkommenden Problemen verschiedene Lösungswege gesucht, aufgezeigt und beschrieben. Es schließt sich der Entwurf der Datenbasis und Datenbank an. Danach werden die einzelnen Anwendungsfälle noch einmal ausführlich dargestellt. Die Abbildungen und Diagramme in diesen Abschnitten richten sich nach der Notation der Unified Modeling Language (UML) 1.4 (siehe z.B. die Notationsübersicht [64] der oose.de GmbH).

4.2 Das Architekturkonzept

Das konzipierte Archivierungssystem setzt auf der Applikationsebene (siehe Tabelle 4.1 auf der nächsten Seite) an, um darunterliegende wohlbekannte Dienste nutzen zu können. Seine Dienste kann es wiederum direkt einem Benutzer anbieten. Der Benutzer muss nicht notwendigerweise ein Mensch aus Fleisch und Blut sein, es kann sich auch um ein anderes Programm handeln, welches sich auf die Dienste des Archivs stützt. Für die weitere Betrachtung ist dieser Unterschied jedoch irrelevant.

Ebene	Beispiele für verteilte Techniken
physikalische Ebene	RAID-Verbund
Dateisystemebene	NFS, Intermezzo
Datenbankebene	Zwei-Phasen-Commit-Protokoll
Applikationsebene	verteiltes Archiv

Tabelle 4.1: Abstraktionsebenen zur Datenspeicherung mit Beispieltechniken

Als verteilte Anwendung basiert das System auf dem Verbund mehrerer, folgend als *Dämonen*[1] bezeichneter Dienstprogramme, die nach Möglichkeit ständig laufen und die miteinander die Basis des verteilten Archivierungssystems bilden.

Das System wird vervollständigt durch ein die eigentliche Benutzerschnittstelle implementierendes Programm, folgend *GUI* [2] genannt, welches sich mit einem der Dienstprogramme verbinden kann, um Benutzeraktivitäten durchzuführen.

Diese Teilung ermöglicht die Einrichtung eines Dämons an einem fest erreichbaren Ort an dem dieser durchgängig betrieben werden kann ohne dass an demselben Ort die Benutzerschnittstelle lauffähig sein muss. Der Teil des Systems, der ständig erreichbar sein muss, ist somit vom nur zeitweilig benötigten Teil des Systems getrennt.

Die Dämonen sind für die Kommunikation zwischen den Systemstandorten zuständig. Pro Standort läuft genau ein solcher Dämondienst. Sie übernehmen auch die Aufgabe der Hardwareeinbindung, die bei Bedarf an verschiedenen Standorten verschieden ausfallen kann. Die Dienste stützen sich auf die mit der Hardwareeinbindung realisierte

[1] Dämon (engl.: Daemon) steht ursprünglich für „Disk And Execution Monitor" und ist die gängige Bezeichnung für ein Dienstprogramm, welches ständig im Hintergrund läuft und bei bestimmten Ereignissen aktiv wird, ohne dass dafür eine Benutzeraktivität nötig ist, siehe [62].

[2] GUI: (engl.: Graphical User Interface) Graphische Benutzerschnittstelle.

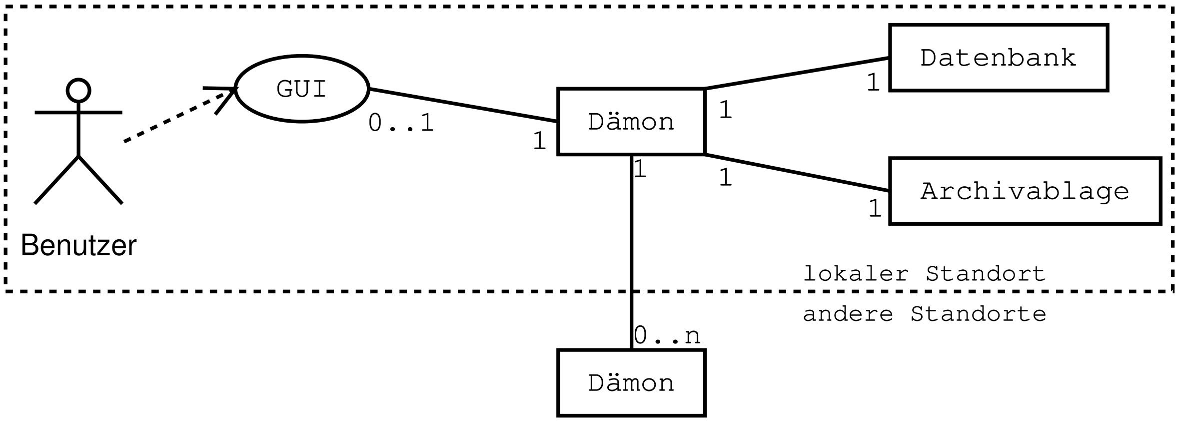

Abbildung 4.1: Übersicht über die Verknüpfungen der Untersysteme an einem Standort und die Verknüpfungskardinalitäten bei n Standorten

Archivablage sowie auf eine eigene Instanz einer relationalen Datenbank, mit deren Hilfe die Metadaten sowie weitere Verwaltungsdaten gespeichert und wieder abgerufen werden können. Dadurch bilden sie die Mittlerschicht (vgl. Abschnitt 2.2.4) zwischen Benutzerschnittstelle und der verteilten Gesamtdatenablage bestehend aus Archivablage und Datenbank an jedem Standort. Die Ablage für die eigentlichen Daten und die Meta- und übrigen Verwaltungsdaten erfolgt also getrennt (vgl. Abschnitt 2.2.1). Der prinzipielle Aufbau ähnelt somit dem aus Abbildung 2.1 auf Seite 10, jedoch werden die Teile der Mittlerschicht und der Datenablage verteilt realisiert. Das prinzipielle Umfeld eines Standortes ist in Abbildung 4.1 dargestellt.

4.2.1 On-Demand-Hierarchien

Es ergibt sich nun die Frage, in welcher Weise die Dienste zur Kommunikation mit ihresgleichen eine Hierarchie aufbauen sollen und werden. Wie in Kapitel 2.2 beschrieben wurde, bieten mehr oder weniger stark ausgeprägte Hierarchien unterschiedliche Vor- und Nachteile.

Das System wird daher so gestaltet, dass die einzelnen Dienste prinzipiell gleichberechtigt sind, jedoch Hierarchien bei Notwendigkeit für einen kurzen Zeitraum aufgebaut

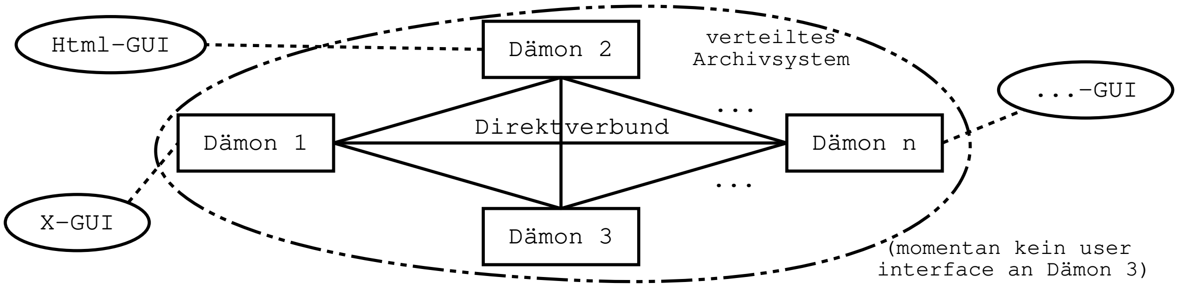

Abbildung 4.2: Verbund von verteilten Dienstprogrammen (Dämonen) und angehängte Benutzerschnittstellen (GUIs)

werden. In Abbildung 4.2 ist ein Verbund gleichwertiger Dienstprogramme miteinander dargestellt.

Der Verbund der Dienste bildet dabei das Rückgrat des Archivierungssystems: Die Programme an den einzelnen Standorten wissen voneinander. Daten, die nicht am lokalen Standort gespeichert sind, können von einem Dämon bei einem anderen angefordert werden. Die Metainformationen liegen zudem lokal vor (dies wird ausführlich in Abschnitt 4.4 besprochen), so dass dafür nicht einmal ein anderer als der lokale Dienst herangezogen werden muss. Notwendig sind Hierarchien jedoch, wenn es um die Einhaltung von einheitlichen Konventionen geht. Dieses umfasst hier im Einzelnen die Meta- und Verwaltungsdaten aus den Bereichen:

- Kategorien
- Benutzer
- Benutzergruppen
- Benutzergruppenrechte
- Formatübersetzungsmöglichkeiten
- Merkmalsextraktionsmöglichkeiten

Diese müssen eindeutig identifizierbar und überall im System gleichlautend sein. Die ebenfalls eindeutig zu identifizierenden Daten und zugehörige Metadaten wurden in diese Liste nicht aufgenommen, da an sie nicht so strenge Forderungen geknüpft sind: Zum einen kann die Eindeutigkeit über eine – lokal aufwendigere – andere Methode, die Erschaffung und Erhaltung eines gemeinsamen Namensraumes über die ident-Funktion sichergestellt werden (vgl. Abschnitt 2.2.8) und zum anderen braucht die Verteilung nicht unmittelbar stattfinden. Sie kann somit auch noch funktionieren, wenn Teile des Systems von anderen Teilen temporär nicht erreicht werden können. All den in der Liste aufgenommenen Punkten ist gemein, dass sie bei der vorgesehenen Verwendung des Archivierungssystems prinzipiell erheblich weniger oft in Anspruch genommen werden als die Funktionen zum Datenein- und -ausliefern, sowie der Suche. Es wird daher hier ein konventioneller Weg über ein Zwei-Phasen-Commit-Protokoll, genauer einer Erweiterung davon, dem in dieser Arbeit erstellten Robusten Commit-Protokoll (siehe Anwendungsfallbeschreibungen und Abschnitt 4.2.3) gewählt, um die Einheitlichkeit zu garantieren. Bei diesem Protokoll wird eine Hierarchie gebildet. Hier wird dabei der Standort, an dem die Benutzeraktion, die zum Ändern der Daten führt, aufgerufen wurde, zum übergeordneten und überwachenden Glied[3] einer nur für die Zeit der Protokollausführung existierenden Hierarchie, bei der alle anderen Glieder dem Überwacher direkt gleichberechtigt untergeordnet sind. Es können zu einem Zeitpunkt mehrere solcher Hierarchien gleichzeitig parallel erstellt werden und bestehen, jedoch ist es genau Sache des Commit-Protokolls, die Sperrung der zu ändernden Datensätze auszuhandeln und im Konfliktfall die Gesamtänderung abzubrechen bzw. wieder rückgängig zu machen.

4.2.2 Persistent gesicherte Warteschlangen

In Abwandlung des Konzepts der persistenten Nachrichten von Lowell und Chen aus [10] (vgl. Abschnitt 2.2.12) werden für die abgesichert[4] stattfinden zu habende Kom-

[3] dieses Glied wird als Commit-Manager bezeichnet, siehe Abschnitt 4.2.3

[4] abgesichert im Sinne von Sicherstellen, dass eine Nachricht genau einmal und mit unverfälschten Daten den Empfänger erreicht und dieses bei garantierter Zustellung

munikation zwischen den Systemteilen Warteschlangen benutzt, welche durch das Verwenden persistenten Speichers gesichert werden. Dabei werden die Funktionen des lokal vorhandenen Dateisystems benutzt, welches für das Funktionieren dieser Methode die Anforderungen des direkten Schreibens erfüllen muss. Es darf also kein Dateisystem für diese Funktion verwendet werden, das zu schreibende Daten zuerst für einige Zeit in flüchtigem Speicher bewahrt (cached/delayed writes). Ebenfalls muss die dem Dateisystem zugrundeliegende verwendete Hardware dieses Kriterium erfüllen, andernfalls kann die Sicherheit der Kommunikation und somit das Funktionieren des Gesamtsystems nicht gewährleistet werden.

Die persistent gesicherten Warteschlangen erweitern das Konzept der persistenten Nachrichten auch in Hinsicht auf die Reihenfolge der eintreffenden Nachrichten an sich (vgl. Abschnitt 2.2.11), da diese nicht nur in einer FIFO-Struktur[5] beim Sender abgelegt werden sondern durch Kontrollnachrichten auch ebendiese Reihenfolge für das Eintreffen der Nachrichten beim Empfänger garantiert. Eine Warteschlange sichert die Kommunikation zwischen genau zwei Systemteilen auf diese Weise ab. Der Absicherung der Kommunikation zwischen mehr als zwei Kommunikationspartnern widmet sich dieses Konzept nicht.

4.2.3 Robustes Commit-Protokoll

Zur Absicherung komplexerer Funktionalität, welche die Kommunikation von potentiell mehr als zwei Systemteilen erfordert, reichen persistente Warteschlangen nicht aus. Hierzu wurden verschiedene Protokolle erdacht. Ein sehr weit verbreitetes Protokoll ist das Zwei-Phasen-Commit-Protokoll (2PC-Protokoll, siehe übernächsten Abschnitt, Bacon et al. [1] und Risnes [4]). Haupteinsatzgebiet ist die Absicherung von verteilten Transaktionen mit Schreib- bzw. Änderungsvorgängen.

[5] FIFO steht für „first in first out" – dem Prinzip einer Warteschlange

Verteilte Transaktionen

Die Anforderungen an eine Transaktion werden mit dem Begriff „ACID” charakterisiert. Dies steht abkürzend für vier Forderungen:

- *Atomizität (atomicity)*

  „Atomizität bedeutet [...], daß eine Transaktion bis zu ihrem erfolgreichen Abschluß überhaupt keine Wirkung hinterläßt, die von fremden Transaktionen beobachtbar wäre, nach ihrem erfolgreichen Abschluß aber ihre Wirkung allgemein sichtbar ist” (aus Lang/Lockemann: „Datenbankeinsatz” [2] S. 620). Durch Atomizität im Zusammenspiel mit der Isolation können mit geringem zusätzlichen Aufwand Störungskaskaden vermieden werden, welche sonst umfangreiche Störungsbehebungsmechanismen erforderlich machten. Bei einer Störung der Transaktionsausführung ist ein *Rücksetzen* (engl.: *Rollback*) eine wenig kommunikationsaufwendige Lösung der festgefahrenen Situation.

  Diese Überlegungen gelten sowohl für lokale wie auch für verteilte Transaktionen. Bei lokalen Transaktionen wären zudem *Sicherungspunkte* (engl.: *Savepoints*) eine Möglichkeit, um nicht die gesamte Transaktion im Fall einer Störung zurücksetzen zu müssen, bei verteilten Transaktionen hingegen ist vom Gebrauch dieser Möglichkeit wegen des hohen Kommunikationsbedarfs zwischen den Transaktionsteilnehmern abzuraten solange nicht durch andere spezielle Forderungen dieser Aufwand gerechtfertigt werden kann. In dieser Arbeit wird auf die Verwendung von Sicherungspunkten nicht weiter eingegangen.

- *Konsistenz (consistency)*

  Konsistenzhaltung ist eine Grundforderung an Aktionen in einem Datenbanksystem. Sie umfasst die Schemakonsistenz der Daten, die referentielle Konsistenz zwischen den Daten sowie deren Erhaltung bei Zustandsänderung (Zustandskonsistenz) und Sichten (Sichtkonsistenz). Diese Forderung kann durch die Verwendung eines gemeinsamen Modells an allen Standorten durch die jeweilige lokale Konsistenzforderung ersetzt werden. Nachdem eine Synchronisierung der Datenmodelle zu einem konsistenten Ursprungszustand einmalig stattgefunden hat, braucht diese Forderung auf der Verteilungsebene nicht weiter betrachtet werden, solange sichergestellt ist, dass die Datenmodelle nicht mehr verändert werden.

- *Isolation (isolation)*
  Als Isolation wird die wechselwirkungsfreie Koordination von nebenläufigen Transaktionen bezeichnet. Wechselwirkungsfrei sind Transaktionen, wenn Zugriffskonflikte auf gemeinsame Daten verhindert werden. Dies kann durch Zwangsserialisierung von Transaktionen erfolgen, welche auf gemeinsame Daten zugreifen müssen. Isolation läßt sich auf der lokalen Ebene der Systeme umsetzen. Zur Konfliktfreiheit reicht dabei die jeweilige Isolationskoordination der jeweils zugrundeliegenden lokalen Transaktionen gegeneinander aus, wenn zusätzlich Atomizität gefordert wird.

- *Dauerhaftigkeit (durability)*
  „Hinter der Dauerhaftigkeit steht die Forderung, daß die Wirkung einer erfolgreich abgeschlossenen Transaktion selbst unter den widrigsten Umständen nicht mehr verlorengeht, es sei denn sie wird durch eine weitere Transaktion ausdrücklich widerrufen" (aus Lang/Lockemann: „Datenbankeinsatz" [2] S. 622). Langfristige Sicherung über Jahre hinweg muss durch ein umgebendes Datensicherungskonzept gewährleistet werden (vgl. Abschnitte 2.2.6 und 2.2.7) und ist nicht Teil der weiteren Betrachtung. Kurzfristig wird diese Forderung durch das *Festschreiben* (engl.: *Commit*) der Wirkung der Transaktion in der Datenbasis erfüllt. Dieses auch in einem Verteilten System zu gewährleisten ist Gegenstand des verwendeten Commit-Protokolls. Lokale Datenbanksysteme alleine können diese Forderung im Gegensatz zu den anderen drei Forderungen nicht erfüllen.

Kommt somit an den verteilten Standorten jeweils eine Datenbank zum Einsatz, welche ihrerseits die lokalen Transaktionen nach dem ACID-Modell absichert, so kann ein System für verteilte Transaktionen auf deren Funktionalität aufbauen. Die Koordination zwischen den verteilten Standorten kann dann allein mit einem Commit-Protokoll abgewickelt werden, welches die Dauerhaftigkeit der Transaktion zuzusichern vermag.

Das Zwei-Phasen-Commit-Protokoll

Das Zwei-Phasen-Commit-Protokoll dient der Synchronisation potentiell mehrerer parallel ablaufender Schreib- bzw. Änderungsvorgänge in einem Datenbestand, indem alle betroffenen Systemteile zu ihrem Zustand befragt werden und gegebenenfalls ein Veto gegen die Ausführung des Schreibvorgangs einlegen können. Im Detail besteht es aus folgenden zwei Phasen:

1. Der Commit-Manager[6] fordert die betroffenen anderen Systemteile, den Commit-Teilnehmern (engl.: Commit Participants) zur Abstimmung auf und sammelt die Stimmen ein, welche auf „Commit" oder „Abort" lauten müssen

2. Der Commit-Manager entscheidet auf Basis der Stimmen aus Phase 1 auf „Commit" oder „Abort" und teilt das Ergebnis den anderen Systemteilen mit

Die erste Phase dient der Sicherung der Wiederholbarkeit. Beim Abschluss der ersten Phase ist bei den Teilnehmern sichergestellt, dass die Wirkung der Transaktion notfalls aus eigener Kraft wiederholt hergestellt werden kann, falls eine Störung eintreten sollte. Die zweite Phase dient der Herstellung der Atomizität, indem die Wirkung sichtbar gemacht wird.
Bacon und Harris beschreiben in [1] S. 590ff. ausführlich die vom Protokoll handhabbaren Fälle der Störung eines oder mehrerer Systemteile inklusive des Commit-Managers bzw. der Verbindung dazwischen und zeigen, dass in einigen Konstellationen das 2PC-Protokoll unzureichend ist. Diese Unzulänglichkeit besteht insbesondere in der Unfähigkeit des Protokolls, das sichere und garantierte Zustellen des Entscheidungsergebnisses in Phase 2, also die Dauerhaftigkeit zu leisten. Hierfür wurden inzwischen verschiedene Anstrengungen unternommen, welche detaillierte und meist leider auch aufwendige und umfangreiche Abschlussprotokolle als Erweiterung des 2PC-Protokolls beschreiben bis hin zu einem Drei-Phasen-Commit-Protokoll.

[6] Commit-Manager wird derjenige Systemteil genannt, der die Protokollausführung überwacht

Robustheit

Hier wird jedoch ein anderer Weg beschritten: Anstelle der Erweiterung des Protokolls selbst, welches bis auf den störanfälligen Abschluss sehr gut funktioniert, wird die für die Praxistauglichkeit nötige Robustheit des Commit-Protokolls durch das Verwenden von persistenten Nachrichten (siehe Lowell und Chen [10] und Abschnitt 2.2.12) bzw. persistent gesicherten Warteschlangen (siehe Abschnitt 4.2.2) gepeichert und somit die Dauerhaftigkeit bei Abschluss der Transaktion an allen Teilnehmerstandorten garantiert. Diese Kombination erweitert die Fähigkeiten des 2PC so, dass das Gesamtkonzept als „Robustes Commit-Protokoll" ohne zusätzlichen Kommunikationsaufwand im Protokoll und ohne eine aufwendige Protokollerweiterung auskommt.

Abbildung 4.3 auf der nächsten Seite zeigt den Aufbau des Robusten Commit-Protokolls inklusive der Wiederherstellungspositionen.[7] Als ersten Schritt speichert der Commit-Manager die Transaktionsdaten und Teilnehmerliste und vermerkt den Beginn im persistenten Speicher. Daraufhin sendet der Commit-Manager die Transaktionsdaten an alle Teilnehmer und wartet auf deren Stimmabgabe. Mit den Transaktionsdaten versucht daraufhin jeder Teilnehmer die Transaktion bis auf das abschließende Festschreiben durchzuführen. Tritt dabei eine Störung auf, so stimmt dieser Teilnehmer mit „Abort", vermerkt dabei vor dem Absenden diese Entscheidung in seinem persistenten Speicher und schließt die Bearbeitung dann mit dem Eintrag „Done" ab. In diesem Zustand kann er weiterhin etwaig doppelte Anfragen des Commit-Managers beantworten. Tritt hingegen keine Störung auf, so stimmt der Teilnehmer mit „Commit". Vor der Stimmabgabe werden die Transaktionsdaten und die Entscheidung im persistenten Speicher vermerkt. Danach wartet der Teilnehmer seinerseits auf die globale Entscheidung des Commit-Managers.

Dieser schreibt unterdessen bei jedem Eintreffen einer auf Abort lautenden Stimme einen entsprechenden Eintrag in den persistenten Speicher. Dauert die Stimmabgabe

[7] Der grundlegende Diagrammaufbau selbst basiert auf der von Risnes in [4] ab Seite 15 für kanalbasierte verteilte Speichersysteme vorgestellten Variante des 2PC-Protokolls und wurde für das hier beschriebene Vorgehen, mit der Absicherung durch das Verwenden persistenten Speichers, abgewandelt und erweitert.

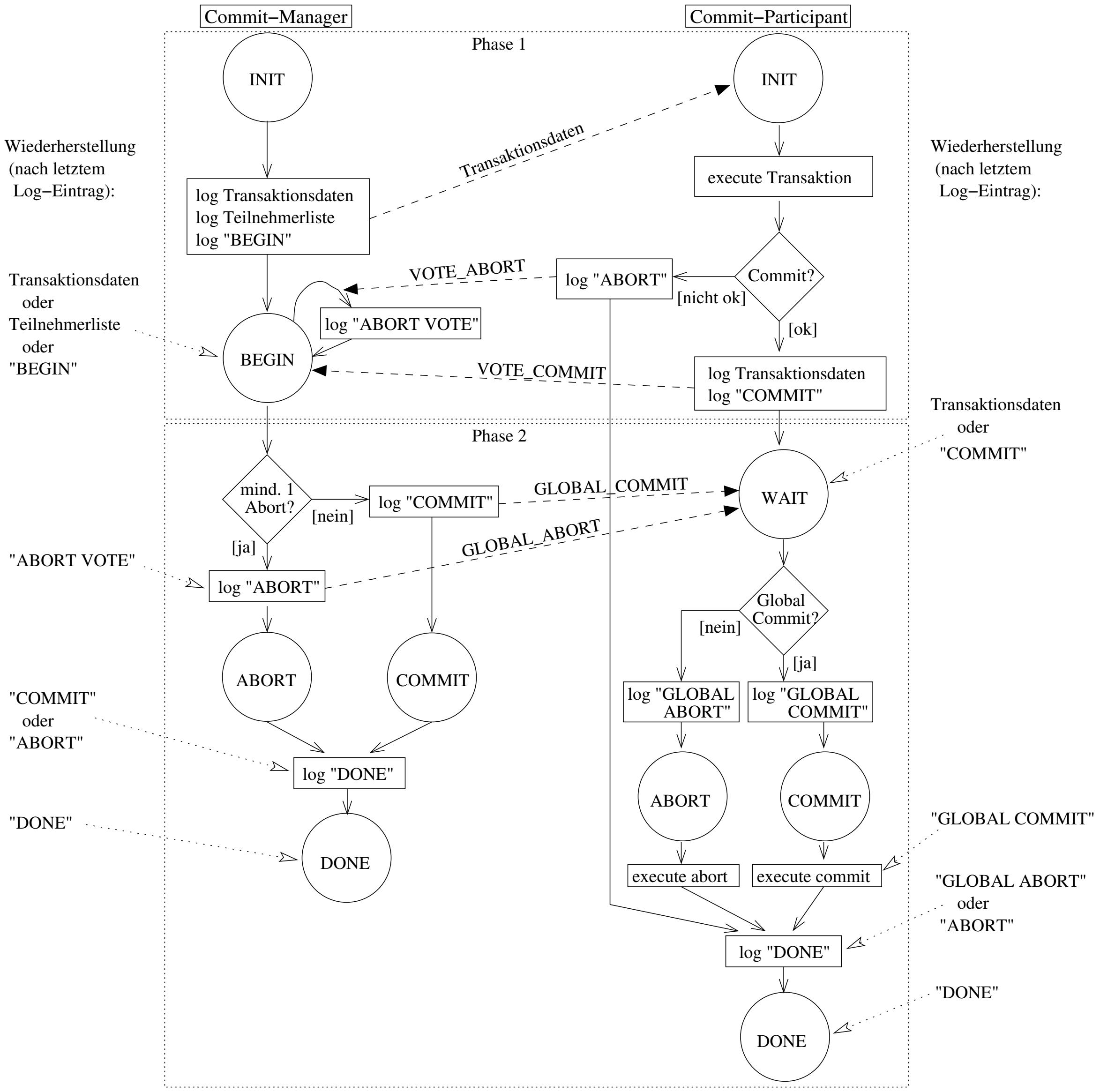

Abbildung 4.3: Das Robuste Commit-Protokoll inklusive Wiederherstellungspositionen

eines Teilnehmers zu lange, so werden diesem nach angemessener Wartezeit (Timeout) die Transaktionsdaten erneut zugesendet. Wurden alle Stimmen eingesammelt, so wird die Entscheidung gefällt und ebenfalls im persistenten Speicher vermerkt. Anschließend wird die Entscheidung über die persistent abgesicherten Warteschlangen den Teilnehmern mitgeteilt. Mit dem Eintrag „Done" schließt der Commit-Manager seine Arbeit ab.

Sobald die Teilnehmer die Entscheidung des Commit-Managers erhalten, wird diese im persistenten Speicher hinterlegt. Es folgt entweder die Festschreibung oder das Rücksetzen der Transaktion und der Abschluss wird mit „Done" wiederum im persistenten Speicher vermerkt.

Durch das zwischenzeitliche Speichern der auf „Abort" lautenden Stimmen bzw. des Teilnehmers beim Commit-Manager kann eine Optimierung erreicht werden, denn diese Standorte brauchen in der zweiten Phase nicht mehr benachrichtigt werden. Sie haben bereits die Ausführung abgeschlossen. Außerdem ist diese Information für eine Wiederherstellung im Falle des Ausfalls des Commit-Managers wertvoll.[8]

Sobald die Transaktionsdaten gespeichert wurden, ist sichergestellt, dass die Gesamttransaktion ausgeführt wird, auch wenn ein oder mehrere Systemteile den Dienst zeitweilig versagen sollten. Das Ergebnis der Gesamttransaktion hingegen wird erst in der zweiten Phase bestimmt.
Im Falle eines Ausfalls eines Teilnehmers oder des Commit-Managers muss beim Neustart des Dienstes eine Wiederherstellung erfolgen. Diese kann sich dabei auf die im persistenten Speicher hinterlegten Informationen stützen. Abbildung 4.3 auf der vorherigen Seite zeigt jeweils am Rand die Position zu der fortgeschritten werden muss, wenn der letzte Eintrag im persistenten Speicher wie angegeben lautet. Die Maßnahmen für die Wiederherstellung richten sich nach dem jeweiligen Ausfallszenario. Die Ausfallszenarien für den Commit-Manager sind folgende:

1. *Der Commit-Manager fällt im Initialzustand aus*: Der Commit-Manager braucht nur neu gestartet und initialisiert zu werden.

2. *Der Commit-Manager fällt nach dem Schreiben der Transaktionsdaten und vor dem ersten Senden aus*: Der Commit-Manager weiß, dass der Verarbeitungsprozess begonnen hat, jedoch nicht, ob bereits Transaktionsdaten versendet wurden. Er liest die Transaktionsdaten und fährt mit der Bearbeitung im Zustand „Begin"

[8] Dieses wurde bereits von Risnes in [4] für kanalbasierte verteilte Speichersysteme nachgewiesen.

fort, d.h. er beginnt mit dem Senden der Transaktionsdaten. Dies kann auf Teilnehmerseite zu doppelten Nachrichten führen. Es stellt jedoch kein unlösbares Problem dar und wird weiter unten behandelt.

3. *Der Commit-Manager fällt beim Warten auf die Stimmabgabe aus*: Falls es mindestens einen Eintrag über eine „Abort"-Stimme gibt, wurden die Transaktionsdaten bereits erfolgreich übermittelt und aufgrund der „Abort"-Stimme muss die Entscheidung ebenfalls auf „Abort" lauten. Der Commit-Manager vermerkt demnach die Entscheidung und übermittelt sie an alle Teilnehmer. Falls kein Stimmeneintrag vorliegt, werden die Transaktionsdaten im Zustand „Begin" versendet, eventuell unwissentlich doppelt.

4. *Der Commit-Manager fällt nach der Entscheidungshinterlegung aber vor dem Informieren der Teilnehmer über die Entscheidung aus*: Die Entscheidung steht fest, sie braucht nur noch an alle Teilnehmer übermittelt zu werden. Nach der Übermittlung wird die Bearbeitung mit dem Vermerk „Done" im persistenten Speicher abgeschlossen.

5. *Der Commit-Manager fällt nach der Entscheidungsübermittlung aus*: Wurde der Abschluss vermerkt, so braucht nichts weiter getan werden. Ansonsten wird die Entscheidung (möglicherweise doppelt) an alle Teilnehmer übermittelt.

Die Ausfallszenarien der Teilnehmer sind folgende:

1. *Der Teilnehmer fällt im Initialzustand aus*: Nach einem Timeout wird der Commit-Manager die Transaktionsdaten erneut an diesen Standort versenden, so dass der Teilnehmer seine Arbeit durchführen kann.

2. *Der Teilnehmer fällt nach dem Schreiben seiner „Abort"-Entscheidung aber vor der Entscheidungsübermittlung an den Commit-Manager aus*: Der Teilnehmer wechselt in den Abschlusszustand und hält sich für die Beantwortung etwaiger Anfragen des Commit-Managers bereit. Dieser wird nach Ablauf der Wartezeit den Teilnehmer erneut nach seiner Stimme fragen.

3. *Der Teilnehmer fällt im Wartezustand aus*: Durch die persistente Nachrichtenübermittlung wird sichergestellt, dass eine Entscheidung des Commit-Managers den Teilnehmer erreichen wird. Dieser bestätigt den Erhalt der Nachricht erst, wenn die Entscheidung in seinem persistenten Speicher vermerkt wurde.

4. *Der Teilnehmer fällt im Abort- oder Commit-Zustand aus*: Die Entscheidung des Commit-Managers wurde im persistenten Speicher vermerkt und kann direkt durchgeführt werden.

5. *Der Teilnehmer fällt im Endzustand aus*: Der Teilnehmer hält sich wie im Endzustand üblich für etwaige (doppelte) Anfragen des Commit-Managers bereit, denn der Commit-Manager könnte ebenfalls zwischenzeitlich ausgefallen sein und Anfragen stellen. Diese werden aus den im persistenten Speicher hinterlegten Daten beantwortet.

Es zeigt sich, dass eine Wiederherstellung in allen Fällen mit geringem Mehraufwand von eventuell doppelt versendeten Nachrichten möglich ist. Das Protokoll kann somit als robust angesehen werden.

Das eventuelle Auftreten doppelter Nachrichten kann ein Problem darstellen, wenn diese nicht als doppelt erkannt werden. Dies könnte dazu führen, dass Transaktionen an einigen aber nicht notwendigerweise allen Standorten mehrfach ausgeführt würden und somit in einem inkonsistenten Zustand der Datenmengen untereinander enden. Dies muss verhindert werden.
Eine Möglichkeit dazu wäre das Vermerken jeder noch so kleinen Aktion im persistenten Speicher. Damit könnte die Wiederherstellung den Zustand vor dem Ausfall exakter wieder herbeiführen, exakt genug, um bereits das Versenden doppelter Nachrichten zu verhindern. Diese Vorgehensweise jedoch bringt den nicht unerheblichen Nachteil mit sich, dass sehr viel mehr Daten im persistenten Speicher abgelegt und sehr viel mehr Schreibaktionen mit dem persistenten Speicher durchgeführt werden müssen. Da zudem der Ausfall nicht zum Regelfall gerechnet werden kann, ist dies eine unzumutbare

Einschränkung.

Eine andere Möglichkeit ist das Ignorieren der Ausführung von Aktionen aus doppelten Nachrichten beim Empfänger. Zwar werden einige Nachrichten zuviel zwischen den Standorten ausgetauscht, da dies aber nicht dem Regelfall entspricht, dürften diese zu tolerieren sein. Um doppelte Nachrichten ignorieren zu können, müssen diese jedoch zuerst als doppelt erkannt werden. Eine einfache Möglichkeit ist die Vergabe von eindeutigen Nummern für jede Transaktion. Tatsächlich braucht nicht einmal jede Transaktion eine eindeutige Nummer bekommen, es reicht beim vorgestellten System, wenn jeweils die Transaktionen eines Standorts eine für den Standort eindeutige Nummer bekommen, da die Leitung des Commit-Protokolls immer in Händen des auslösenden Standorts liegt. Somit ist auch das Herstellen der Eindeutigkeit dieser Nummern nicht schwer, da bei der Generierung nur der lokale Standort mit dem Commit-Manager involviert ist.

Sollte nun ein Teilnehmer des Commit-Protokolls eine Nachricht doppelt erhalten, so kann er diese anhand der mitgesendeten Identifikationsnummer als doppelt erkennen und entsprechend reagieren. Bereits berechnete Ergebnisse können direkt, ohne die Berechnung erneut durchzuführen, an den Anfragenden übermittelt werden.

Mit dieser Behandlung von doppelten Nachrichten ist das Robuste Commit-Protokoll somit abschließend betrachtet. Es kommt also ohne ein zusätzliches aufwendiges Abschlussprotokoll aus.

4.3 Gestaltung der ident Funktion

An die Funktion ident werden hohe Anforderungen gestellt (vgl. Gleichung 3.16 auf Seite 34). Diese muss eine Identifikation auf Basis der gegebenen Daten leisten. Diese Funktion muss sie in diesem verteilten Archivierungssystem zudem an jedem Standort des Systems unabhängig von anderen Standorten erfüllen. Außerdem soll diese Identifikation nicht allzu viel Speicherplatz benötigen, da sie von vielen Stellen aus referenziert werden wird.

Die Eindeutigkeit an einem lokalen Standort läßt sich beispielsweise herstellen, indem eine für diesen Standort fortlaufende Nummer gewählt wird. Durch solch eine Nummer ist jedoch noch kein Wiedererkennen bei gleichlautender Dateneingabe gegeben, denn diese Nummer wäre kontextunabhängig. Eine kontextabhängige Berechnung mit einem Ergebnis fester Speichermenge können sogenannte *Message-Digest*-Funktionen herstellen. Sie stellen einen Bereich aus den Hash-Funktionen dar, welche einer Eingabe einen festgelegten Adressierungsbereich zuweisen. Es handelt sich um sogenannte Einwegberechnungen, da eine Umkehrberechnung einer solchen Funktion nicht wieder zu einem eindeutigen, sondern zu einem mehrdeutigen Ergebnis führt. Es wird durch den Berechnungsaufbau garantiert, dass bei gleichlautender Eingabe immer wieder die gleiche Ausgabe erreicht wird. Vertreter dieser Funktionsgattung sind beispielsweise MD5 oder SHA-1, welche eine 128 bzw. 160 Bit umfassende Ausgabe liefern. Bestimmte Message-Digest-Funktionen – MD5 und SHA-1 gehören zu dieser Gruppe – können einen zusätzlichen Nutzen bieten. Sie sind so geschaffen, dass sie zur Prüfung der Datenintegrität verwendet werden können. Störungen bei einer Datenübertragung oder während der Langzeitspeicherung führen zu einer Veränderung der Bits und Bytes in den Daten. Eine Berechnung mit dieser Gruppe der Message-Digest-Funktionen führt bereits bei kleinsten Änderungen – kleinstmöglich ist die Veränderung eines einzelnen Bits – zu einem deutlich anderen Ergebnis. Durch diese Eigenschaft können sie zur Prüfung der Daten auf Verfälschungen eingesetzt werden. Die Kontextabhängigkeit sowie eine zusätzliche Absicherung der Datenübertragung kann somit mit dem Verwenden eines solchen Algorithmus bei relativ sparsamem Umgang mit Speicher erreicht werden. Es bleibt noch die Forderung, diese Funktion müsse die Eindeutigkeit im gesamten verteilten System zusichern. Dies kann durch stete Synchronhaltung aller Standorte geschehen. Dieses jedoch führt zu einigem Mehraufwand bei der Kommunikation zwischen den Standorten, welche nach Möglichkeit vermieden werden soll, damit das System skalierungsfähig bleibt und sich nicht durch eine überhandnehmende für die Synchronisation nötige Kommunikation selbst behindert (vgl. Abschnitt 3.3). Dies kann zwar bei als weniger häufig anzunehmenden Operationen tolerierbar sein, ist es bei einer häufigen Operation wie dem Dateneinfügen jedoch nicht.

Die systemweite Eindeutigkeitsforderung auf lokaler Basis könnte bei einer fortlaufenden Nummerierung durch das Zuweisen von disjunkten Nummernbereichen zu jedem Standort geschehen. Bei einer kontextabhängigen Funktion aber kann ein solcher Nummernbereich nur schwer abgegrenzt werden. Die hier gewählte Lösung ist das Kombinieren von Message-Digest-Ergebnis mit einer für den Einfügestandort eindeutigen Nummer. Hierzu wird an das Ergebnis der Message-Digest-Funktion die ohnehin bereits eindeutig vergebene Standortnummer angehängt. Mit einem auf diese Weise bestimmten Identifikator wird die geforderte Funktionalität bei lokaler Berechenbarkeit erreicht.

4.4 Datenbankentwurf

Das konzipierte Archivierungssystem verwendet eine Datenbank zur Speicherung von anfallenden Daten. Im Folgenden wird nun geklärt, *welche* der anfallenden Daten überhaupt für die Speicherung in der Datenbank vorgesehen werden sowie *wie* und in welchen Zusammenhängen diese Daten gespeichert werden müssen und auch *was* mit den Daten geschehen soll, wie sie beispielsweise verändert oder ausgetauscht werden können. Die entworfene Datenbasis muss den an sie gestellten Anforderungen in einem Archivierungssystem über eine lange Zeit genügen. Um eine hohe Qualität des Entwurfs zu gewährleisten, wird nach einem bewährten Entwurfsprozeß vorgegangen, wie ihn Lang/Lockemann in [2] ab Seite 291ff. beschreiben. Der Entwurfsprozeß gliedert sich in vier Phasen von denen zwei in das Stadium der Konzeption und eine in das Stadium der Implementation fallen:

1 *Anforderungsanalyse.* In dieser Phase wird die Ausgangsbasis für die folgenden Phasen geschaffen, indem abgegrenzt und verzeichnet wird, welche Daten wie und mit was für Mitteln und mit welchen Zusammenhängen in die Datenbasis aufgenommen werden. Als Mittel dienen Daten-, Operations- und Ereignisverzeichnisse, die die Daten selbst, die mit den Daten möglichen Operationen und die Auslöser für die Operationen erfassen.

2 *Konzeptueller Entwurf.* In dieser Phase werden die Sachverhalte und Gesetzmäßigkeiten aus der Anforderungsanalyse formal gestaltet und mit einem semantischen Schema beschrieben. Als Darstellungsmittel wird in dieser Arbeit die Entity-Relationship-Modellierung verwendet, welche wegen ihrer „intuitiv begründeten (a priori-) Semantik” (Lang/Lockemann: „Datenbankeinsatz” [2], S. 333) und der Verwendung von graphischer Modellierung gewählt wurde.

3 *Logischer Entwurf.* Der logische Entwurf schließlich übersetzt die Erkenntnisse aus dem semantischen Schema in ein konkretes Datenbasisschema. In dieser Arbeit wird dazu ein relationales Datenbasisschema verwendet.

4 *Physischer Entwurf.* Der Physische Entwurf dient der Leistungsoptimierung bezüglich der speziellen Anwendung, hier des Archivierungssystems.

Die Entwurfstätigkeiten sind miteinander verzahnt und die Phasen 2 bis 4 werden üblicherweise mehrfach in Verifikations-, Bewertungs- und Validierungszyklen durchlaufen. Hier wird nun das damit erzielte Endergebnis vorgestellt.

4.4.1 Anforderungsanalyse

In der Anforderungsanalyse wird zunächst dargestellt, welche Daten in die Datenbank aufgenommen werden müssen:

Binärdaten in die Datenbank oder ins Dateisystem?

Zu dieser Fragestellung hat Käster [3] ab Seite 19 für Bilddaten ausführliche Überlegungen angestellt. Die dort durchgeführten Überlegungen treffen größtenteils für die allgemeinen Daten und potentiell nicht kleinen Datenmengen in einem Archivierungssystem zu:

Für die Speicherung in der Datenbank spricht die mögliche Einhaltung der referentiellen Konsistenz. Diese besagt, dass wenn Daten aus der Datenbank gelöscht oder umbenannt werden, dann auch die auf sie verweisenden Referenzen aus den Metadaten

entfernt oder umbenannt werden und der Datenbestand somit in einem konsistenten Zustand verbleibt. Jedoch ist diese Funktionalität im konzipierten Archivierungssystem nicht von Belang und braucht nicht sichergestellt zu werden, da im System gar keine Funktionalität zum Entfernen oder Umbenennen der archivierten Daten vorgesehen und auch nicht erwünscht ist.
Ein weiterer Vorteil der Speicherung in der Datenbank ist, dass die Daten mit einem Datenbankzugriff direkt in den Hauptspeicher gelesen und dort bearbeitet werden können. Ein dateiorientierter Zugriff unter Verwendung von Dateideskriptoren ist nicht notwendig. Dieses stellt aber gleichzeitig einen Nachteil dar, da für Standardprogramme, wie z.B. das in Abschnitt 2.3.2 vorgestellte Formatumwandlungsprogramm „ps2pdf", die Daten erst in eine temporäre Datei im Dateisystem kopiert werden müssen, um sie anschließend mit einem entsprechenden Systemaufruf zu konvertieren.
Ein weiterer Nachteil ist, dass viele Datenbanksysteme eine Größenbeschränkung für die zu speichernden Daten besitzen, welche im Falle des Fehlens eines Datentyps mit genügend großer Speicherkapazität das Speichern in der Datenbank unmöglich oder sehr umständlich machen würde.

Das Speichern im Dateisystem besitzt für den nötigen Einsatzzweck hingegen einige Vorteile. Wie bereits als Nachteil der Speicherung in der Datenbank erwähnt, lassen sich Standardprogramme wie „ps2pdf" mit direktem Dateizugriff leichter betreiben.
Die häufigeren Suchanfragen werden zudem nicht durch Auslesen der Daten, sondern der Metadaten beantwortet. Die Metadaten gehören somit eindeutig in den Datenbankspeicher, nicht jedoch die Daten.
Die fehlende Absicherung der referentiellen Integrität spielt für diese Anwendung keine Rolle.

Als Fazit ergibt sich daher, die Daten selbst im Dateisystem unterzubringen, Metadaten sowie Verwaltungsdaten, bei denen die referentielle Integrität sichergestellt sein muss, hingegen in der Datenbank.

Datenmodell

Aus den vorhergehenden Überlegungen ergibt sich, dass folgende Daten in dem zu erstellenden Datenmodell enthalten sein müssen:

- Daten über die Ablage der Daten
- Metadaten zu den Daten
- Daten über die verschiedenen Standorte
- Daten über die Verfügbarkeit der Daten an den verschiedenen Standorten
- bekannte Datenkategorien
- bekannte Datengruppen
- bekannte Formate
- Daten über bekannte Formatübersetzungsfunktionen
- Daten über bekannte Merkmalsextraktionsfunktionen
- Benutzer
- Benutzergruppen
- Benutzergruppenrechte
- Zugehörigkeitsdaten der Benutzer zu den Benutzergruppen

Ein besonderer Schwerpunkt des Archivierungssystems liegt in der herausragenden Stellung, die die flexible Verwaltung der Metadaten einnimmt.
Das Ziel des Archivierungssystems ist es, die Verwaltung der archivierten Daten komfortabel zu ermöglichen. Dazu gehört in erster Linie die Möglichkeit der effizienten Suche über die Metadaten zum Wiedererlangen der archivierten Daten.

Datenverzeichnis

Die zu den in der Datenbasis auftretenden Daten gehörenden *Fakten* werden formal im Datenverzeichnis (Tabelle 4.2) für eine übersichtliche Darstellung erfasst. Zu den nach ihrer Identifikationsnummer geordneten Daten sind Beschreibungen sowie Metainformationen gegeben, die Zweck und Verwendung charakterisieren. Die Kardinalitätsangaben sind dabei nur sehr grobe Schätzungen. Sie dienen der groben Abschätzung der Verhältnisse zwischen den Daten.

Nr.	Bezeichnung	Beschreibung	Metainformationen
D01	Daten	Identifikation: Identifikator Klassifikation: Archivdaten Daten: Identifikator, Repertorien-eintrag	Kardinalität: ca. 10000 Oberbegriff: Archivalie
D02	Metadaten	Identifikation: Bezeichnung, Identifikator, Klassifikation: Metadaten Daten: Bezeichnung, Identifikator, Wert	Kardinalität: ca. 50 mal die von D01 Oberbegriff: Repertorie
D03	Standort-daten	Identifikation: Nummer Klassifikation: Verwaltungs-daten Daten: Nummer, IP-Adresse, Port	Kardinalität: ca. 100 Oberbegriff: Verwaltung
D04	Verfügbar-keit	Identifikation: Nummer, Identifikator Klassifikation: Verwaltungs-daten Daten: Nummer, Identifikator	Kardinalität: ca. $\frac{1}{2}$ mal die von D01 mal die von D03 Oberbegriff: Verwaltung
D05	Kategorien	Identifikation: Bezeichnung Klassifikation: Klassifikation Daten: Bezeichnung, Name, Position	Kardinalität: ca. 100 Oberbegriff: Kategori-sierung
D06	Datengruppen	Identifikation: Bezeichnung Klassifikation: Klassifikation Daten: Bezeichnung	Kardinalität: ca. 1000 Oberbegriff: Gruppierung

Nr.	Bezeichnung	Beschreibung	Metainformationen
D07	Formate	Identifikation: Bezeichnung Klassifikation: Klassifikation Daten: Bezeichnung	Kardinalität: ca. 1000 Oberbegriff: Datenformat
D08	Format-übersetzungen	Identifikation: Eingabeformat, Ausgabeformat Klassifikation: Übersetzung Daten: Eingabeformat, Ausgabeformat, Übersetzungs-funktionsname	Kardinalität: ca. 2 mal die von D07 Oberbegriff: Format-übersetzung
D09	Merkmals-extraktionen	Identifikation: Eingabeformat Klassifikation: Merkmals-berechnung Daten: Eingabeformat, Extraktions-funktionsname	Kardinalität: ca. 100 Oberbegriff: Merkmals-berechnung
D10	Benutzer	Identifikation: Bezeichnung Klassifikation: Verwaltung Daten: Bezeichnung, Name, Passwort	Kardinalität: ca. 100 Oberbegriff: Benutzer-verwaltung
D11	Benutzer-gruppen	Identifikation: Bezeichnung Klassifikation: Verwaltung Daten: Bezeichnung, Status	Kardinalität: ca. 10 Oberbegriff: Benutzer-verwaltung
D12	Benutzer-gruppen-zugehörigkeit	Identifikation: Benutzer, Benutzergruppe Klassifikation: Verwaltung Daten: Benutzer, Benutzergruppe	Kardinalität: ca. die von D10 mal die von D11 Oberbegriff: Benutzer-verwaltung
D13	Benutzer-gruppen-einlieferungs-rechte	Identifikation: Bezeichnung Klassifikation: Verwaltung Daten: Benutzergruppe, Recht	Kardinalität: wie bei D11 Oberbegriff: Zugriffs-verwaltung
D14	Benutzer-gruppen-such- und -auslieferungs-rechte	Identifikation: Identifikator, Benutzergruppe Klassifikation: Verwaltung Daten: Benutzergruppe, Identifikator, Suchrecht, Auslieferrecht	Kardinalität: wie bei D01 Oberbegriff: Zugriffs-verwaltung

Tabelle 4.2: Datenverzeichnis

Basiselemente in der Datenbank sind Daten D01. Sie werden durch einen eindeutigen Identifikator (vgl. Gleichung 3.16 auf Seite 34 und Abschnitt 4.3) und den Repertorieneintrag beschrieben. Dieser Repertorieneintrag wird von der Hardwareeinbindung vergeben und dient dieser zum Wiederauffinden der Daten. Die Datenmenge des Systems wird mit 10000 angenommen, diese Menge ist allerdings nicht begrenzt und kann beliebig erweitert werden. Für die Datensuche dienen die Metadaten D02, welche neben freien Einträgen auch Einträge aus dem Bereich der Kategorien D05, Datengruppen D06 und Formaten D07 beinhalten können. Die Metadateneinträge umfassen den Identifikator der Daten, denen sie zugeordnet sind, ihre Bezeichnung sowie den eingetragenen Wert. Für jeden Dateneintrag existieren im Normalfall ein oder mehrere Metadateneinträge mit unterschiedlichen Bezeichnungen.

Die Standortinformationen D03 entsprechen der in Gleichung 3.31 auf Seite 38 geforderten Menge der Standorte und nehmen das Wissen der Dämonen über die Erreichbarkeit anderer Dämonen im Netzwerk auf. Sie werden durch eine eindeutige Standortnummer beschrieben. Zudem werden sie aus den Verfügbarkeitsdaten D04 referenziert. Diese wiederum stellen das in Gleichung 3.32 auf Seite 38 geforderte Wissen über die Verfügbarkeit von Daten an den jeweiligen Standorten dar. Da diese Verknüpfung im Gegensatz zu den übrigen Metadaten mengenwertige Einträge aufnimmt, wird sie zur in der Folge leichteren Handhabung getrennt von den übrigen Metadaten geführt.
Die Kategorien D05 umfassen die Daten zur Bezeichnung einer Kategorie und zu deren Position im Ableitungsbaum. Die genaue Ausprägung dieser Positionsbeschreibung wird in Abschnitt 4.4.3 beleuchtet.

Die Datengruppen D06 sind direkt durch ihre Bezeichnung gekennzeichnet und besitzen keine weiteren Beziehungen. Die Formate D07 werden sowohl von den Einträgen zur Formatübersetzung D08 wie auch der Merkmalsextraktion D09 referenziert und können zudem in die Metadaten aufgenommen werden. Jene wiederum speichern neben dem

Quellformat und im Falle von D08 dem Zielformat die jeweilige Funktionsbezeichnung.

Die folgenden Daten D10 bis D14 umfassen die Verwaltungsdaten des Systems zur Rechteverwaltung (vgl. Abschnitt 3.2.2). Die Benutzerdaten D10 werden durch ihre Bezeichnung eindeutig beschrieben und besitzen zur Authentifizierung ein Passwort. Benutzergruppendaten D11 werden durch ihre Bezeichnung beschrieben und benannt. Ihnen ist außerdem ein Status zugeordnet, welcher die Daten zu Standardrechten der Benutzergruppe sowie zur Kennzeichnung einer Benutzergruppe als zur Administration zugehörig zusammenfasst. Die Zugehörigkeit eines Benutzers zu einer Benutzergruppe wird mit den Daten der Benutzergruppenzugehörigkeit D12 ausgewiesen, welche zu diesem Zweck die Bezeichnungen von Benutzer und Benutzergruppe referenziert. Die eigentlichen Rechte einer Benutzergruppe werden mit den Daten D13 und D14 gespeichert. Neben der Referenz durch die Benutzergruppenbezeichnung wird im Falle von D13 das Einlieferrecht und im Falle von D14 das Such-, das Auslieferrecht sowie der dem Such- und Auslieferrecht zugeordnete Datenidentifikator als Referenz auf die Daten D01 aufgenommen.

Operationsverzeichnis

Die im Datenverzeichnis verzeichneten Daten bilden die Grundlage des Archivierungssystems. Neben den Daten existieren zur Vervollständigung der Funktionsfähigkeit die Operationen, mit denen die Daten verwendet oder erstellt werden. Die Operationen werden formal im Operationsverzeichnis (Tabelle 4.3) erfasst und dargestellt. Die einzelnen nach ihrer Identifikationsnummer geordneten Operationen werden durch die Beschreibung der Ein- und Ausgaben und der Metainformationen zu geschätzter Häufigkeit, Bezugsdaten und Auswirkungen auf die Datenbasis genauer gekennzeichnet. In dieses Verzeichnis werden also die *Aktionen* aufgenommen, die mit den Daten durchgeführt werden können.

Nr.	Bezeichnung	Beschreibung		Metainformationen	
Op01	Dateneingabe (Storing)	Eingabe:	Daten Metadaten	Häufigkeit:	häufig
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D01, D02, D04, D05, D06, D09
				Auswirkung:	Einfügen
Op02	Datensuche (Query)	Eingabe:	Metadaten-kriterien	Häufigkeit:	sehr häufig
		Ausgabe:	Daten-nummern	Bezugsdaten:	D02, D05, D06, D07
				Auswirkung:	keine
Op03	Daten-auslieferung (Retrieval)	Eingabe:	Daten-identifikator	Häufigkeit:	sehr häufig
		Ausgabe:	Daten	Bezugsdaten:	D01, D07, D08
				Auswirkung:	keine
Op04	Einlieferungs-rechteprüfung	Eingabe:	Benutzer	Häufigkeit:	wie Op01
		Ausgabe:	Rechte-status	Bezugsdaten:	D12, D13
				Auswirkung:	keine
Op05	Suchrechte-prüfung	Eingabe:	Benutzer Datennummer	Häufigkeit:	wie Op02
		Ausgabe:	Rechte-status	Bezugsdaten:	D12, D14
				Auswirkung:	keine
Op06	Auslieferungs-rechteprüfung	Eingabe:	Benutzer Datennummer	Häufigkeit:	wie Op03
		Ausgabe:	Rechte-status	Bezugsdaten:	D12, D14
				Auswirkung:	keine
Op07	Kategorien-erweiterung	Eingabe:	Kategorie Bezeichnung	Häufigkeit:	mittel
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D05
				Auswirkung:	Einfügen
Op08	Gruppen-vergabe	Eingabe:	Bezeichnung	Häufigkeit:	mittel
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D06
				Auswirkung:	Einfügen
Op09	Formateingabe	Eingabe:	Format	Häufigkeit:	selten
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D07
				Auswirkung:	Einfügen
Op10	Formatüber-setzungs-möglichkeiten-erweiterung	Eingabe:	Quellformat Zielformat Übersetzungs-funktion	Häufigkeit:	selten
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D07, D08
				Auswirkung:	Einfügen, Löschen
Op11	Merkmals-extraktions-möglichkeiten-erweiterung	Eingabe:	Extraktions-funktion	Häufigkeit:	selten
		Ausgabe:	„Erfolgs-meldung”	Bezugsdaten:	D07, D09
				Auswirkung:	Einfügen, Löschen

Nr.	Bezeichnung	Beschreibung		Metainformationen	
Op12	Benutzer-zugang	Eingabe:	Bezeichnung Name Status Passwort	Häufigkeit: Bezugsdaten: Auswirkung:	selten D10 Einfügen,
		Ausgabe:	„Erfolgs-meldung”		
Op13	Benutzer-anmeldung	Eingabe:	Bezeichnung Passwort	Häufigkeit: Bezugsdaten: Auswirkung:	häufig D10 keine
		Ausgabe:	Benutzerstatus		
Op14	Benutzer-gruppen-zugang	Eingabe:	Benutzerliste Benutzergruppe	Häufigkeit: Bezugsdaten: Auswirkung:	selten D10, D11, D12 Einfügen
		Ausgabe:	„Erfolgs-meldung”		
Op15	Benutzer-gruppen-änderung	Eingabe:	Benutzerliste Benutzergruppe	Häufigkeit: Bezugsdaten: Auswirkung:	selten D10, D11, D12 Ändern
		Ausgabe:	„Erfolgs-meldung”		
Op16	Benutzer-gruppen-abgang	Eingabe:	Benutzergruppe	Häufigkeit: Bezugsdaten: Auswirkung:	selten D11, D12 Löschen
		Ausgabe:	„Erfolgs-meldung”		
Op17	Einlieferungs-rechtevergabe	Eingabe:	Benutzergruppe Rechte	Häufigkeit: Bezugsdaten: Auswirkung:	selten D13 Ändern
		Ausgabe:	„Erfolgs-meldung”		
Op18	Such- und Auslieferungs-rechtevergabe	Eingabe:	Benutzergruppe Nummer Rechte	Häufigkeit: Bezugsdaten: Auswirkung:	selten D14 Ändern
		Ausgabe:	„Erfolgs-meldung”		
Op19	Standort-erweiterung	Eingabe:	Standortdaten	Häufigkeit: Bezugsdaten: Auswirkung:	sehr selten D03 Einfügen
		Ausgabe:	„Erfolgs-meldung”		
Op20	Standort-suche	Eingabe:	Standort-nummer	Häufigkeit: Bezugsdaten: Auswirkung:	häufig D03 keine
		Ausgabe:	Adresse, Port		
Op21	Verfügbar-keits-erweiterung	Eingabe:	Datennummer Standort	Häufigkeit: Bezugsdaten: Auswirkung:	häufig D04 Einfügen
		Ausgabe:	„Erfolgs-meldung”		

Nr.	Bezeichnung	Beschreibung	Metainformationen
Op22	Verfügbar-keits-prüfung	Eingabe: Daten-identifikator Ausgabe: Standort-nummernliste	Häufigkeit: häufig Bezugsdaten: D04 Auswirkung: keine

Tabelle 4.3: Operationsverzeichnis

Die Operationen Op01, Op02 und Op03 entsprechen den lokalen Ausgaben der Funktionen der im Anforderungskapitel aufgestellten Forderungen an insert, search und deliver. Die Rechteprüfung übernehmen dabei die Operationen Op04, Op05 und Op06. Durch das Programm muss die Funktionalität auch für den verteilten Einsatz garantiert werden. Durch den lokalen Charakter der Datenablage reicht es dazu, die Anfrage bzw. Eingabe zum passenden Standort weiterzuleiten und dort diese lokalen Operationen auszuführen. Hinzu kommt im Fall von insert und deliver noch das Ansprechen der Hardwareeinbindung, welche selbst aber die Garantie für die Ein- bzw. Auslieferung der eigentlichen Daten gemäß der mit D01 gespeicherten Repertorieneinträge gibt. Zur Durchführung der korrekten Verteilung dieser Aufgaben an sich kann sich das Programm der Operationen Op19 bis Op22 bedienen.
Op07 und Op08 entsprechen ebenso wie die gerade besprochenen Operationen den lokalen Ausgaben für deducecategory bzw. creategroup. Die funktionale Absicherung dieser Funktionalität für die verteilte Operation wird mit dem Robusten Commit-Protokoll gesichert. Ebenso abgesichert werden die verteilten Funktionen zur Erweiterung der Formatumwandlungs- und der Merkmalsextraktionsmöglichkeiten, deren lokale Operationen Op10 bzw. Op11 jeweils mit Rückgriff auf Op09 ausgestattet sind.
Mit dem Robusten Commit-Protokoll werden auch die Änderungen, Zu- sowie Abgänge und Rechteänderungen von Benutzergruppen gesichert, bei denen Op14, Op15 bzw. Op16 sowie Op17 bzw. Op18 die jeweils lokalen Operationen aus der mit dem Commit-Protokoll überwachten Transaktion darstellen.
Die verbleibenden Operationen werden im Zuge der lokalen Benutzeranmeldung eines neuen (Op12) oder bestehenden (Op13) Benutzers verwendet.

Ereignisverzeichnis

Ein Ereignisverzeichnis (Tabelle 4.4) dient der Beschreibung aller *Auslösebedingungen* für die im Operationsverzeichnis niedergeschriebenen Operationen. Dadurch werden indirekt die möglichen Abläufe in Bezug auf die mit den Daten in der Datenbank möglichen Operationen beschrieben. Die Ereignisse decken sich bis auf die Abmeldung, welche keine Datenbankoperationen erfordert, mit den Anwendungsfällen, die in Abschnitt 4.5 genauer beschrieben werden. In allen Fällen ist die Ereignissyntax elementar und konditional, da diese allesamt ohne zusätzliche Bedingungen direkt vom Benutzer ausgelöst werden und keine temporalen Ereignisse, wie dies beispielsweise bei einer Inventur anfiele, im System vorkommt.

Nr.	Bezeichnung	Bedingung		Bezug
E01	Dateneingabe (Storing)	Syntax Semantik	elementar konditional	Op01, Op04 Op21
E02	Datensuche (Query)	Syntax Semantik	elementar konditional	Op02, Op05
E03	Datenausgabe (Retrieval)	Syntax Semantik	elementar konditional	Op03, Op06, Op20, Op22
E04	Kategorien-erweiterung	Syntax Semantik	elementar konditional	Op07
E05	Gruppen-vergabe	Syntax Semantik	elementar konditional	Op08
E06	Rechte-verwaltung	Syntax Semantik	elementar konditional	Op17, Op18
E07	Rechtegruppen-verwaltung	Syntax Semantik	elementar konditional	Op14, Op15, Op16
E08	Formatumwandlungs-möglichkeitenerweiterung	Syntax Semantik	elementar konditional	Op09, Op10
E09	Merkmalsextraktions-möglichkeitenerweiterung	Syntax Semantik	elementar konditional	Op09, Op11
E10	Anmeldung	Syntax Semantik	elementar konditional	Op12, Op13
E11	Standorterweiterung	Syntax Semantik	elementar konditional	Op19

Tabelle 4.4: Ereignisverzeichnis

4.4.2 Konzeptueller Entwurf

Aus den Ergebnissen der Anforderungsanalyse wird anschließend ein semantisches Schema in Entity-Relationship-Modellierung (kurz ER-Modell) erstellt. Das Entity-Relationship-Modell ist ein semantisches Modell, das auf Formalisierung verzichtet, dafür aber die eingängigere graphische Modellierung des Konzepts ermöglicht (siehe Lang/Lockemann: „Datenbankeinsatz" [2] S. 333ff.).

Ein ER-Modell basiert auf *Gegenständen* (engl.: Entities), die die Objekte in der Modellwelt repräsentieren. Diese werden im ER-Diagramm durch Rechtecke symbolisiert. Zur Beschreibung der Gegenstände werden *Attribute* verwendet, welche im ER-Diagramm als Ellipsen dargestellt werden. Zwischen den Gegenständen können Beziehungen existieren. Diese werden graphisch als Rauten notiert. „Eine *Beziehung* (Relationship) beschreibt einen Zusammenhang zwischen mehreren Gegenständen" (aus Lang/Lockemann: „Datenbankeinsatz" [2] S. 335). Beziehungen selbst wiederum können auch durch Attribute ergänzt werden. Die Verbindungen zwischen Attributen, Gegenständen und Beziehungen werden im Diagramm durch Linien hergestellt. An diesen Linien werden zudem die in den Beziehungen vorkommenden Kardinalitäten der Gegenstände mit jeweils einem Tupel, welches aus Mindest- und Höchstmenge der zugeordneten Gegenstände besteht, eingetragen. Ein Stern steht dabei für beliebig viele Gegenstände. Falls rekursive Beziehungen auftreten – dies ist hier bei den Kategorien der Fall – werden die jeweiligen Rollen, die die Gegenstände in der Beziehung wahrnehmen, an den Linien vermerkt.

Abbildung 4.4 auf der nächsten Seite zeigt die im Modell enthaltenen Gegenstände nebst zugehörigen Attributen sowie die Beziehungen zwischen den Gegenständen. Es zeigt sich, dass das Datenmodell auf drei Schwerpunkten basiert sein wird, den Daten, den Benutzergruppen und den Formaten, wobei den Daten die hauptsächliche Konzentrationsfunktion zukommt. Dies deckt sich mit den bisherigen Erwartungen, dass die Daten die Hauptrolle im gesamten System spielen und durch die Funktionalität

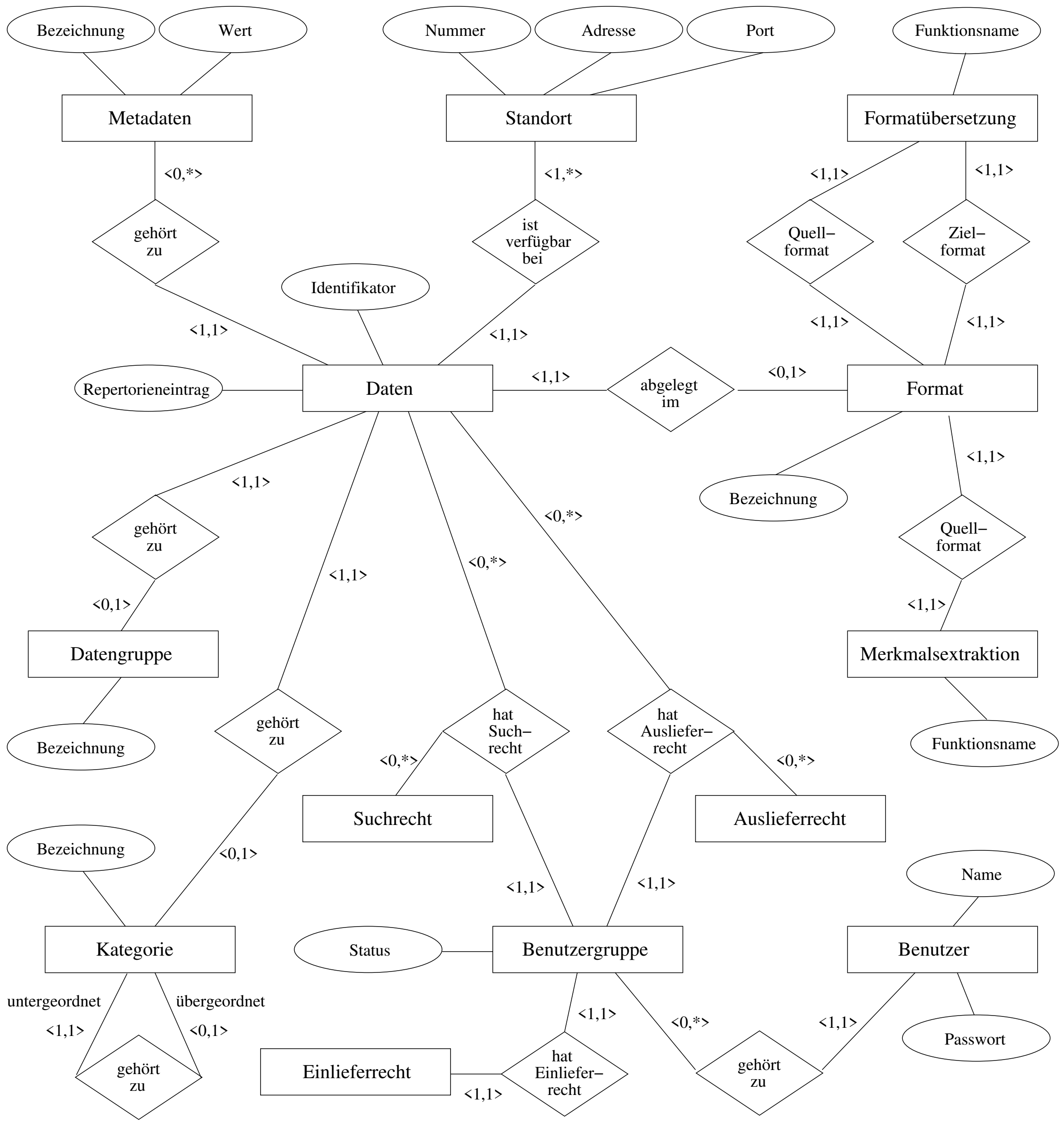

Abbildung 4.4: Entity-Relationship-Diagramm

über die Formate mit Übersetzung zwischen diesen sowie Merkmalsextraktion mit diesen ergänzt und über die Rechtevergabe basierend auf den Benutzergruppen verwaltet werden.

4.4.3 Logischer Entwurf

Das relationale Datenmodell und die dieses Modell implementierenden Datenbanksysteme können sich auf langjährige Erfahrungen stützen (siehe dazu Miners „New Advances in the Filesystem Space" [52]). Dabei stößt die Mächtigkeit dieses Modells in Bezug auf die speicherbaren Datentypen und -beziehungen manchmal an die Modellgrenzen. Das relationale Datenmodell vermag keine mengenwertigen Datentypen direkt in einer Tabelle zu speichern, ebensowenig kann eine Baumstruktur direkt abgebildet werden. Jedoch gibt es Mittel und Wege, diese Beschränkungen zu umgehen, wenn auch durch einen gewissen Mehraufwand. Für den Großteil des bei diesem Archivierungssystem zu speichernden Daten reicht das relationale Modell direkt aus und auch für die Baumstruktur der Kategorien läßt sich ein gangbarer Weg finden. Dies begründet neben der Ausgereiftheit heutiger relationaler Datenbanksysteme die Wahl dieses und keines anderen Datenmodells, wie beispielsweise dem NF^2-Modell oder einem objektorientierten Modell (siehe Lang/Lockemann: „Datenbankeinsatz" [2] S. 97ff. und 175ff.).

Speichern einer Baumstruktur in einem relationalen Datenmodell

Für die Umsetzung der Kategorienbeziehungen sind noch einige Überlegungen anzustellen. Diese prinzipiell für baumartige Strukturen allgemeingültigen Überlegungen werden hier beispielhaft anhand der in diesem System verankerten Kategorien durchgeführt.

Eine – nach einer im Rahmen dieser Arbeit durchgeführten Umfrage unter mehreren Datenbankprogrammierern in Bielefeld – anscheinend intuitive Vorgehensweise ist die Folgende: Die Kategorien werden in einer Relation gespeichert, welche neben dem Bezeichnungsattribut noch ein Attribut für die Darstellung der Beziehung zur Oberkategorie als Selbstreferenz auf das Bezeichnungsattribut derselben Tabelle enthält. In der Sprache SQL[9] formuliert würde eine entsprechende Tabelle mit der Anweisung

```
create table KATEGORIEN (
KATEGORIENBEZEICHNUNG character varying(32) primary key,
```

[9] Im Folgenden wird der SQL92-Standard für die Formulierung von Datenbankanweisungen verwendet.

```
OBERKATEGORIE character varying(32) references
KATEGORIEN(KATEGORIENBEZEICHNUNG));
```

erstellt. Hiermit lassen sich nun baumartige Strukturen schnell und mit wenig Platzaufwand in der Datenbank speichern. Der Wurzelknoten läßt sich schnell finden, denn bei ihm enthält das Attribut `OBERKATEGORIE` den SQL-Wert `NULL`. Auch die Blattknoten des Baumes, welche als Eigenschaft aufweisen, dass sie nirgends im Attribut `OBERKATEGORIE` eines in der Relation enthaltenen Tupels referenziert werden, lassen sich wegen dieser Eigenschaft leicht heraussuchen. Bei der Betrachtung der Problemstellung zeigt sich jedoch, dass in der Praxis des Archivierungssystems entweder nach der Gesamtliste (für deren Darstellung bei der Dateneingabe) gesucht wird oder nach der Eigenschaft „ist Unterkategorie von". Letzteres geschieht im Zuge der Datensuche, welche zusammen mit der Datenauslieferung die voraussichtlich häufigst vorkommende Operation im Archivierungssystem darstellt (vgl. Operationsverzeichnis).

Eine Anfrage nach dieser Eigenschaft läßt sich jedoch mit den Mitteln von SQL bei dieser Tabellendefinition nicht mehr allgemeingültig formulieren. Für den Sonderfall, dass die Tiefe der Baumstruktur bekannt und vor allem fest ist, kann mit ineinander geschachtelten `select`-Abfragen (mit genau so vielen Schachtelungen, wie die Baumtiefe beträgt) noch eine Lösung erfolgen. Die freie Kategorisierung, die mit diesem Archivierungssystem umgesetzt werden soll, erlaubt aber gerade keine festgelegte Baumtiefe. Ein anderer Ansatz ist also nötig.

Ein anderer Vorschlag aus der Umfrage lautete, anstelle der Oberkategorien die Unterkategorien zu jeder Oberkategorie zu speichern. Dies wäre in zwei Ausprägungen möglich: Zum einen könnten die *direkten* Unterkategorien gespeichert werden. Dieses hält zwar den Speicheraufwand in Grenzen, führt jedoch wieder auf das gerade aufgetauchte Problem der ungewiß oft zu schachtelnden `select`-Abfragen. Zum anderen könnten *alle* Unterkategorien mit ihrem gesamten Pfad gespeichert werden. Dies würde zwar das Abfrageproblem lösen, aber neben einem noch tolerierbaren Mehraufwand beim Einfügen einer Unterkategorie führte dies zu einem inakzeptablem Speicheraufwand.

Es gibt aber noch eine weitere Möglichkeit, die aber in der Umfrage von keinem Teilnehmer gefunden wurde und daher hier ausführlich vorgestellt wird: Eine Baumstruktur läßt sich nicht nur durch die direkten Beziehungen zwischen zwei Knoten beschreiben, es läßt sich auch seine Struktur als sogenannter *Baumdurchlauf* linearisieren. Baumdurchläufe gibt es in drei Ausführungen, als Beispiel wird der in Abbildung 4.5 gezeigte Kategorienbaum verwendet:

- *Preorder-Durchlauf*: Das mit dem Knoten assoziierte Attribut wird ausgegeben, bevor die Zweige betrachtet werden. Es ergibt sich für das Beispiel die Reihenfolge Allgemeine Daten, Musik, Graphik, PNG-Graphik, Text, TeX-Text, ASCII-Text.
- *Inorder-Durchlauf* (bei Binärbäumen): Das mit dem Knoten assoziierte Attribut wird ausgegeben, nachdem der linke Zweig betrachtet wurde, aber noch bevor der rechte Zweig betrachtet wird. Ein Beispieldurchlauf mit dem um den Knoten Musik gekürzten und so auf einen Binärbaum zurechtgestutzten Beispielbaum ergibt PNG-Graphik, Graphik, Allgemeine Daten, TeX-Text, Text, ASCII-Text.
- *Postorder-Durchlauf*: Das mit dem Knoten assoziierte Attribut wird ausgegeben, nachdem die Zweige betrachtet wurden. Ein Postorder-Durchlauf durch den Beispielbaum ergibt die Aufzählung Musik, PNG-Graphik, Graphik, TeX-Text, ASCII-Text, Text, Allgemeine Daten.

Diese Linearisierung läßt sich nun ausnutzen: Es ist ausreichend, wenn bei einem Durchlauf mitgezählt, bei jedem Abstieg und jedem Aufstieg der Zähler um den Wert 1 inkrementiert wird und der jeweils aktuelle Wert an der Preorder- sowie an der Postorderposition notiert wird. Den Beispielbaum mit derartig nummerierten Knoten zeigt

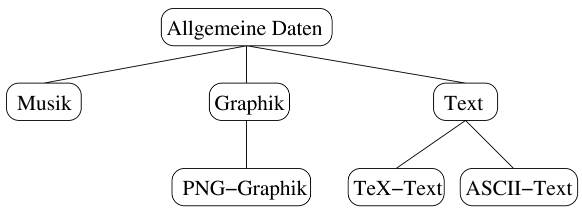

Abbildung 4.5: Kategorienbaum mit Beispielkategorien

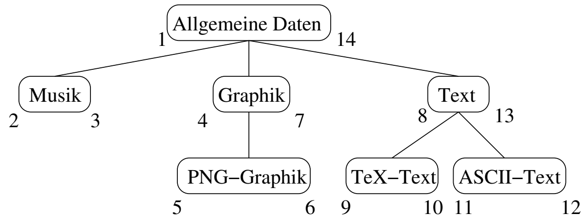

Abbildung 4.6: Kategorienbaum mit nummerierten Beispielkategorien: Die L-Werte sind links, die R-Werte rechts neben den Knoten vermerkt.

Abbildung 4.6. Es werden also zu jedem Knoten zusätzlich zwei Nummern gespeichert. Diese werden mit L für die bei der Preorderposition und mit R für die bei der Postorderposition notierte Nummer benannt. Der zur intuitiven Methode zusätzliche Speicherbedarf dieser Methode liegt mit zwei zusätzlichen Werten pro Knoten noch in einem erträglichem Rahmen. Eine Tabelle mit diesen Daten kann mit SQL wie folgt erstellt werden:

```
create table KATEGORIEN (
KATEGORIENBEZEICHNUNG character varying(32) primary key,
L integer not null, R integer not null);
```

Wie lassen sich nun die bekannten Abfragen und vor allem die Abfrage nach allen Unterkategorien einer Kategorie gestalten? Der Wurzelknoten ist immer derjenige Knoten, bei dem der L-Wert 1 ist bzw. der R-Wert der maximale in der Relation vorkommende Wert ist. Ein Blattknoten besitzt die Eigenschaft, dass dort der R-Wert um genau 1 größer als der L-Wert ist. Die in den anderen Modellen schwierige Abfrage nach der Kategorienzugehörigkeit läßt sich nun problemlos abfragen, da alle L- und R-Werte von Unterkategorien einer Kategorie *zwischen* deren L- und R-Wert liegen. Dies läßt sich mit einer einfachen Anfrage in SQL wie folgt ausnutzen, wobei `:OBERKATEGORIE` durch die gewünschte Kategorienbezeichnung ersetzt werden muss:

```
select KAT1.KATEGORIENBEZEICHNUNG
from KATEGORIEN as KAT1, KATEGORIEN as KAT2
where KAT1.L between KAT2.L and KAT2.R
and KAT2.KATEGORIENBEZEICHNUNG = :OBERKATEGORIE;
```

Diese Abfrage ist ohne weitere Veränderungen immer einsetzbar und muss dabei nicht beliebig tief verschachtelt werden.
Diese Modellierung hat jedoch auch Nachteile. Einfüge-, Änderungs- und Löschoperationen sind aufwendiger als bei den zuvor vorgestellten Methoden. Änderungen und Löschungen kommen in diesem Archivierungssystem bei den Kategorien selbst nicht zum Einsatz, so dass nun noch das Einfügen betrachtet werden muss.
Das Einfügen der ersten Kategorie gestaltet sich noch recht einfach, der L-Wert muss auf 1, der R-Wert auf 2 lauten. Beim Einfügen einer Kategorie unterhalb einer bestehenden Kategorie müssen jedoch bei allen Kategorien oberhalb der Einfügemarke die L- und R-Werte inkrementiert werden. Für das Einfügen muss der R-Wert der Oberkategorie vor der Änderung bekannt sein:

```
:ORIENTIERUNGSWERT = select R from KATEGORIEN
where KATEGORIE = :OBERKATEGORIE;
```

In SQL formuliert sieht die Einfügeoperation (unter Einsetzung der entsprechenden Werte in die Variablen `:NEUEKATEGORIE` und `:ORIENTIERUNGSWERT`) dann so wie folgt aus:

```
update KATEGORIEN
set R= case when R >= :ORIENTIERUNGSWERT then R+2 else R end,
set L= case when L > :ORIENTIERUNGSWERT then L+2 else L end
where R >= :ORIENTIERUNGSWERT;
insert into KATEGORIEN (KATEGORIENBEZEICHNUNG, L, R)
values (:NEUEKATEGORIE, :ORIENTIERUNGSWERT,
(:ORIENTIERUNGSWERT+1));
```

Dies ist ein nicht unerheblicher Aufwand. Insbesondere muss sichergestellt werden, dass diese Änderungen atomar erfolgen. Diese Forderung wird von einem Datenbanksystem, welches die bereits in Abschnitt 4.2.3 beschriebenen ACID-Anforderungen für Transaktionen unterstützt, aber erfüllt. Da jedoch das Einfügen neuer Kategorien als nicht allzu häufig stattfindende Operation angesehen wird (vgl. Operationsverzeichnis), die

Suche hingegen erheblich häufiger und damit nach Möglichkeit weniger aufwendig zu gestalten ist, erscheint der Mehraufwand bei der Einfügeoperation vertretbar. Mit der Auswahl der vorgestellten Methode ist nun die letzte verbleibende Hürde vor der Erstellung des Tabellenmodells genommen.

Die Datenspeicherung

Aus den Ausführungen zum konzeptuellen Entwurf läßt sich nun ein konkretes Datenbasisschema erstellen, indem die Beziehungs- und Gegenstandstypen des ER-Modells direkt in ein relationales Datenmodell überführt werden. Dabei werden in den Relationen aus der zugehörigen Attributmenge *Primärschlüssel* bestimmt, welche in der jeweiligen Relation einen enthaltenen Datensatz eindeutig bestimmen. Es sollten für den Primärschlüssel eine möglichst geringe Anzahl an Attributen verwendet werden. Alle nicht zu diesem Schlüssel gehörenden Attribute einer Relation nennt man Nichtschlüsselattribut.
Alle vorgestellten Gegenstände und Beziehungsattribute lassen sich direkt oder nach den Betrachtungen zur Speicherung der Kategorienbaumstruktur in das relationale Datenmodell überführen.

Es stellt sich an dieser Stelle die Frage, welche im relationalen Modell vorhandenen Datentypen für die zu speichernden Attribute geeignet sind. Einen wichtigen Punkt bilden die als Primärschlüssel dienenden Attribute. Für die in der Betrachtung der Ausgestaltung der ident-Funktion erzielten Ergebnisse ist der Speicherplatzbedarf der verwendeten Message-Digest-Funktion und zusätzlich der Speicherplatz für die Standortnummer nötig. Dafür ist der SQL-Datentyp `character` geeignet. Für den Platzbedarf bei Verwendung der SHA-1 Funktion erscheinen mit der Textrepräsentation der Standortnummer 32 Zeichen als angemessen. Diese Anzahl wird folglich auch für alle übrigen Bezeichnungen verwendet. Bei dem potentiell höheren Zeichenanzahlbedarf von frei definierbaren Attributen wird der dafür geeignetere SQL-Datentyp `text` verwendet. Dies betrifft die Benennung des Formats und der Formatübersetzungs- und Merkmalsextraktionsfunktionsnamen, den Kategoriennamen, das Wertattribut der Metadaten, den Repertorieneintrag und den Benutzernamen. Nummernwertige Attribute werden als `integer` ins Datenmodell aufgenommen. Der boolsche Charakter der Einfüge-, Such- und Auslieferrechte wird durch den Datentyp `boolean` abgedeckt. Es verbleibt noch das Adressattribut der Standortdaten, für welches sich der Datentyp `inet`[10] eignet.

Normalisierung

Bei der Überführung aus dem ER-Modell wird eine Normalisierung durchgeführt, um Redundanz und Anomalien in den sich ergebenden Relationen zu vermeiden. Dazu werden die im ER-Modell durch Beziehungen dargestellten bestehenden funktionalen Abhängigkeiten untersucht. Man unterscheidet mehrere, aufeinander aufbauende Normalformen, die immer restriktivere Forderungen an die Relationen stellen:

- *Erste Normalform*: „Eine Relation ist in *erster Normalform* (1NF) genau dann, wenn die Domänen aller Attribute elementar sind, d.h. sich nicht als Potenzmenge oder kartesisches Produkt aus einfacheren Wertevorräten darstellen lassen" (aus Lang/Lockemann: „Datenbankeinsatz" [2] S. 320). Die ursprünglich in den Anforderungen geforderte Menge der Standorte, an denen Daten verfügbar sind und die im ER-Diagramm als „ist verfügbar bei" ausgedrückt wird, verstieße somit gegen diese Forderung. Diese Beziehung läßt sich aber auch in einer separaten Verknüpfungsrelation ausdrücken, welche den Einzelbezug zwischen den Daten und dem jeweiligen Standort ausdrückt, an dem diese verfügbar sind. Diese Lösung greift auch an den anderen Problemstellen, wie der Zugehörigkeit von Benutzern zu Benutzergruppen.

[10] `inet` ist ein speziell für die Aufnahme von IP-Adressen geeigneter Datentyp, welcher nicht bei allen Datenbanksystemen, jedoch in der von der im nächsten Kapitel besprochenen Implementation verwendeten Datenbank PostgreSQL definiert ist. Alternative Implementationen unter der Verwendung anderer Datenbanksysteme können dieses Attribut anstelle dessen durch eine Textrepräsentation und den Datentyp `character` speichern.

- *Zweite Normalform*: „Eine Relation ist in *zweiter Normalform* (2NF) genau dann, wenn sie in 1NF ist und jedes Nichtschlüsselattribut vom Primärschlüssel voll funktional abhängig ist" (ebenfalls aus [2] S. 321). Jede Relation mit einattributigem Primärschlüssel und jede Relation, welche keine Nichtschlüsselattribute besitzt, ist in 2NF. Die für die 1NF herangezogenen Verknüpfungsrelationen besitzen keine Nichtschlüsselattribute und sind somit auch in 2NF. Alle übrigen Relationen sind bereits von dieser Form, da die Nichtschlüsselattribute nicht nur in ihrer Gesamtheit, sondern jedes einzeln vom Primärschlüssel voll funktional abhängig ist.

- *Dritte Normalform*: „Eine Relation ist in *dritter Normalform* (3NF) genau dann, wenn sie in 2NF ist und kein Nichtschlüsselattribut transitiv vom Primärschlüssel abhängt" (auch aus [2] S. 322). Mit der dritten Normalform werden also funktionale Abhängigkeiten innerhalb der Menge der Nichtschlüsselattribute verboten. Auch mit dieser Forderung ergeben sich für das hier betrachtete System keine Probleme bis auf die zwei Relationen Formate und Kategorien, bei denen die jeweilige Bezeichnung eine für die Praxis relevante Abkürzung für den kompletten Format- bzw. Kategoriennamen ist (die letztendlich diese Abbildung durchführende Operation ist eine Message-Digest-Berechnung). Es gibt also zwei Schlüsselkandidaten. Die Formate weisen keine weiteren Nichtschlüsselattribute auf, so dass sie dennoch in 3NF vorliegen. Bei den Kategorien hängen die Attribute `L` und `R` jeweils von den Schlüsselkandidaten ab, so dass eine transitive Beziehung besteht. Für eine Umformung in 3NF wird die Kategorienrelation also in zwei Relationen zerlegt, von denen eine die Attribute `Kategorienbezeichnung` und `Kategorienname` und die andere die `Kategorienbezeichnung` sowie `L` und `R` aufnehmen. Es sei an dieser Stelle angemerkt, dass durch den physischen Entwurf diese Zerlegung im Zuge der dort stattfindenden Denormalisierung wieder aufgehoben wird.

- *Weitere Normalformen*: Über die vorgestellten Normalformen hinaus existieren noch weitere, die Relationen stärker einschränkende Normalformen, wie die Boyce-

Codd-Normalform. Bei der Anwendung der sich mit höheren Normalformen ergebenden Forderungen läßt sich eine abhängigkeitsbewahrende Zerlegung aber nicht mehr garantieren. Sie werden hier nicht weiter betrachtet. Redundanz und Anomalien in den Relationen, die mit den Regeln der verschiedenen Normalformen vermieden werden sollen, stören vor allem bei der Behandlung von Änderungsoperationen in der Datenbasis.

Das sich aus diesen Überlegungen ergebene Datenmodell ist in Abbildung 4.7 auf der nächsten Seite dargestellt, wobei die Zerlegung der Kategorienrelation wieder rückgängig gemacht wurde und die Abbildung somit den endgültigen Entwicklungsstand des Modells nach dem physischen Entwurf zeigt. Die referentiellen Beziehungen, welche direkt durch das Datenbanksystem sichergestellt werden können, sind mit durchgezogenen Linien markiert. Darüberhinaus existieren schwächere logische Beziehungen, die nicht direkt durch referentielle Beziehungen in diesem relationalen Modell abgedeckt sind. Sie sind anhand der gepunkteten Linien ersichtlich. Eine Begründung wird beim physischen Entwurf geliefert.

4.4.4 Physischer Entwurf

Nachdem mit dem Tabellenmodell der logische Entwurf abgeschlossen wurde, gilt es nun Überlegungen zur Leistungsfähigkeit des Modells und etwaiger Optimierungen zu führen. Lang und Lockemann führen dazu in [2] auf Seite 442 aus: „Ein herausragender Leistungsengpaß ist auch heute noch bei Datenbanksystemen der Zugriff auf die Hintergrundspeicher. Der Minimierung der Anzahl dieser Zugriffe dienen daher alle Maßnahmen der Leistungsoptimierung". Die heutzutage hauptsächlich genutzte Möglichkeit zur Zugriffsminimierung ist das Erstellen von Indizes über häufig benötigte Attribute. Indizes selbst benötigen jedoch wiederum für ihre Pflege Zugriffe, so dass weniger häufig benötigte Attribute nicht indiziert werden sollten. Für die Indizierung bieten sich die jeweiligen Primärschlüssel der Tabellen an. Sie müssen durch die Schlüsselbedingung ohnehin eindeutig gehalten werden. Heutige Datenbanksysteme erstellen deswegen implizit einen Index für jeden Primärschlüssel bei der Tabellenerstellung.

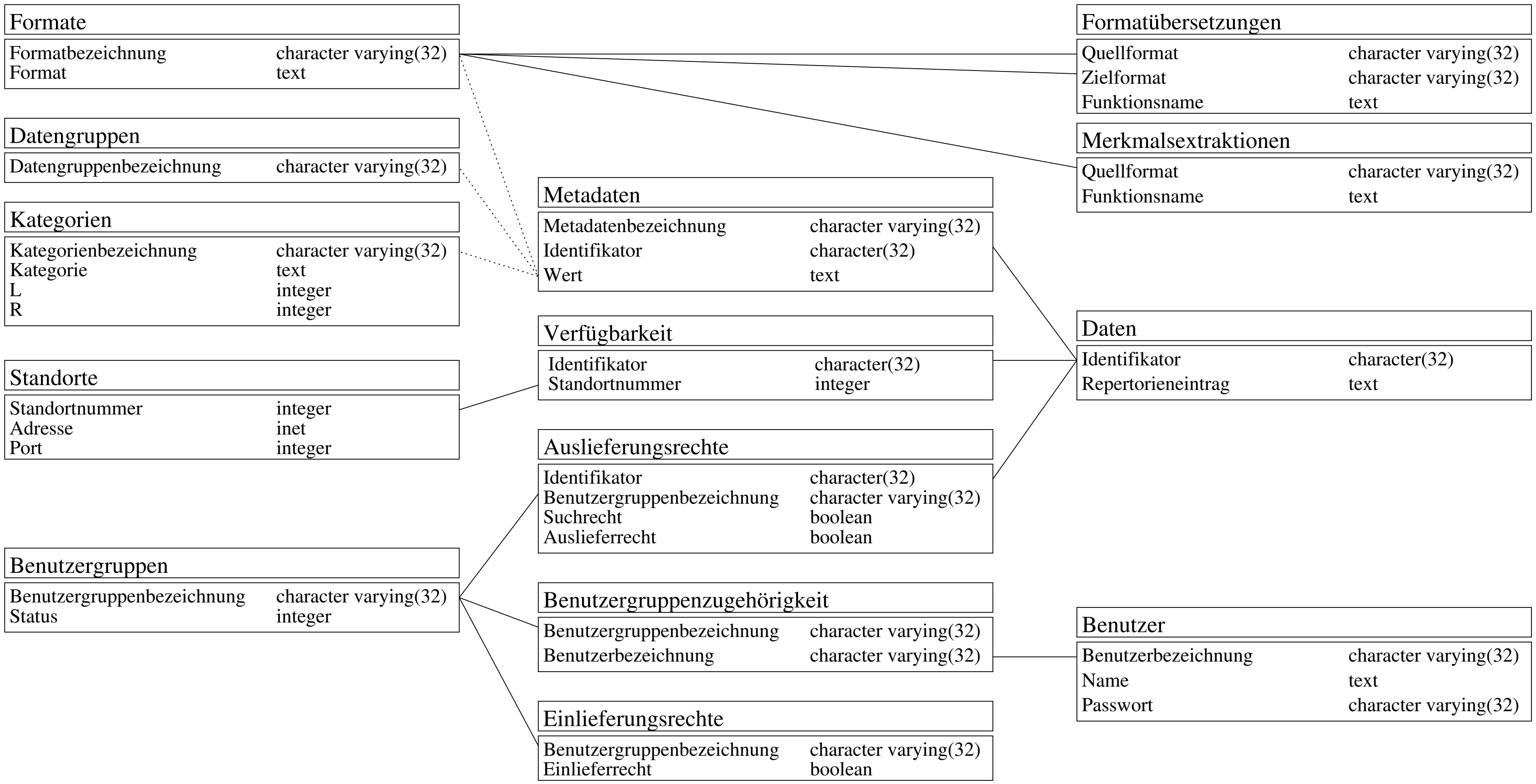

Abbildung 4.7: Relationales Tabellenmodell mit Beziehungen zwischen den Tabellen:
Die durchgezogenen Linien markieren referentielle Beziehungen, die gepunkteten Linien schwächere logische Beziehungen zwischen den jeweils verknüpfenden Attributen der Tabellen.

Weiterhin müssen Zugriffe für die Pflege von referentiellen Verweisen getätigt werden. Wo diese durch andere Mittel konsistent gehalten werden können, kann dem Datenbanksystem diese Pflege erspart werden. Bei der Beschreibung des Tabellenmodells wurde angemerkt, dass in dem vorgestellten Modell nicht alle möglichen referentiellen Beziehungen ausgeführt wurden. Diese wurden in Abbildung 4.7 auf der vorherigen Seite gepunktet dargestellt. Eine Formulierung im relationalen Modell wäre zwar möglich, würde jedoch einen Mehraufwand mit zusätzlichen Tabellen und referentiellen Beziehungen untereinander bedeuten. Da diese Beziehungen jedoch bei diesem Archivierungssystem allesamt während der Einfügephase von Operation Op01 gesetzt werden und an keiner anderen Stelle mehr verändert werden – Datenidentifikatoren, Datengruppen, Kategorien und Formate werden immer nur zugefügt, jedoch nicht wieder entfernt oder umbenannt – kann diese Prüfung auch vor dem Einfügen direkt im Programm erfolgen und braucht nicht durch das zugrundeliegende Datenbanksystem zugesichert werden. Dies spart entsprechende Zugriffe ein.

Im Zuge der im logischen Entwurf durchgeführten Normalisierung wurde die Relation Kategorien zerlegt, um der dritten Normalform zu entsprechen. In der Praxis führt diese Trennung zu einem höheren Aufwand des Datenbanksystems, welches die beiden Relationen beim Zugriff wieder verknüpfen muss. Da für das Einfügen neuer Kategorien – und das ist die einzig dort mögliche Änderungsoperation in diesem System – eine den ACID-Regeln folgende Transaktion verwendet wird, stört die sonst mögliche Anomalienbildung in der Praxis nicht. Die Trennung wird wieder aufgehoben (denormalisiert), um letztendlich eine bessere Leistung des Systems zu gewinnen. Somit liegen letztendlich alle Relationen in 3NF vor, bis auf die Kategorienrelation, die in 2NF ist.

4.5 Anwendungsfallanalyse

Nach der ausführlichen Betrachtung der verwendeten Einzelkonzepte und der Erstellung eines Datenmodells für die in der Datenbank gespeicherten Meta- und Verwaltungsdaten steht nun abschließend für das Konzeptionskapitel eine Analyse der Anforderungen und die Beschreibung der Umsetzung aus Anwendersicht aus. Aus den Anforderungen aus Kapitel 3 ergeben sich dafür folgende, hier zunächst kurz umrissene Anwendungsfälle:

AF1 Dateneingabe (Storing)
: Die Dateneingabe umfasst den Vorgang des Einfügens neuer Daten und Metadaten in das Archiv. In diesen Prozess ist die Merkmalsextraktion zur automatischen Gewinnung der gesamten oder von Teilen der Metadaten eingebunden.

AF2 Datensuche (Query)
: Die Datensuche beinhaltet den Vorgang des Suchens im Datenbestand des Archivs mittels gegebener Kriterien aus den Metadaten. Sie liefert alle aufgefundenen Datenidentifikatoren zurück, zu welchen die Kriterien und Suchrechte passen.

AF3 Datenauslieferung (Retrieval)
: Die Datenauslieferung umspannt den Vorgang des Auslieferns von Daten mittels eines gegebenen Datenidentifikators. In diesen Prozess ist die Formatübersetzung zur Umwandlung der Daten in ein gewünschtes Format eingebunden.

AF4 Kategorienerweiterung
: Die Kategorienerweiterung beschreibt den Vorgang der Ableitung einer neuen Kategorie aus einer bestehenden Kategorie.

AF5 Gruppenvergabe
: Die Gruppenvergabe beschreibt den Vorgang des Anlegens einer neuen Gruppendefinition für die Gruppenzuordnung innerhalb der Metadaten.

AF6 Rechteverwaltung (Administration)
: Die Rechteverwaltung hat den Vorgang des Einstellens von Zugriffsrechten für eine Rechtegruppe als Inhalt. Dies ist eine Aufgabe der Administration.

AF7 Rechtegruppenverwaltung (Administration)
: Die Rechtegruppenverwaltung beinhaltet den Vorgang des Änderns von Benutzergruppenzugehörigkeiten. Dies ist eine Aufgabe der Administration.

AF8 Erweiterung der Formatumwandlungsmöglichkeiten (Administration)

Die Erweiterung der Formatumwandlungsmöglichkeiten umfasst den Vorgang der Erweiterung des Archivierungssystems um eine neue Übersetzungsfunktion für die Umwandlung von Daten aus einem in ein davon verschiedenes Format. Dies ist eine Aufgabe der Administration.

AF9 Erweiterung der Merkmalsextraktionsmöglichkeiten (Administration)

Die Erweiterung der Merkmalsextraktionsmöglichkeiten umspannt den Vorgang der Erweiterung des Archivierungssystems um eine neue Berechnungsfunktion für das automatische Extrahieren von Kriterien aus Daten. Dies ist eine Aufgabe der Administration.

AF10 Anmeldung

Die Anmeldung beinhaltet den Vorgang der Authentifizierung[11] des Benutzers und im Fall eines neuen Benutzers dessen Aufnahme in die Verwaltungsdaten.

AF11 Abmeldung

Die Abmeldung beschreibt den Vorgang des Abmeldens des angemeldeten Benutzers.

AF12 Standorterweiterung

Die Standorterweiterung umfasst den Vorgang der Bekanntgabe eines neuen Systemstandorts sowie des Synchronisierens des neuen Standorts mit den an den alten Standorten bereits vorhandenen Daten. Dies ist eine Aufgabe der Administration.

Die in den Anforderungen getrennt behandelten Funktionen der Formatübersetzung und der Merkmalsextraktion sowie der Hardwareeinbindung sind systemimmanente Funktionen der Anwendungsfälle AF1 und AF3 und werden daher hier nicht als eigenständige Anwendungsfälle gesehen und aufgeführt.

Die nun kurz vorgestellten Anwendungsfälle werden im Folgenden detailliert beschrieben. Zu jedem Anwendungsfall ist ein Diagramm zugeordnet, das den Vorgangsablauf

[11] Für eine Erklärung siehe [61]

darstellt und die jeweiligen Aktivitäten einem Verantwortungsbereich unterstellt. An Verantwortungsbereichen wird dabei in Benutzer bzw. Administrator, lokalen Systemteil, lokale Datenbasis und andere, nicht lokale Systemteile unterschieden.

4.5.1 AF1 Dateneingabe (Storing)

Auslöser und Vorbedingungen

Die Dateneingabe wird vom Benutzer ausgelöst. Der Benutzer muss dem System bekannt (angemeldet) sein. Die einzufügenden Daten müssen vorliegen.

Nachbedingungen

Die Daten wurden mitsamt der angegebenen Metadaten in das Archiv unter einem eindeutigen Datenidentifikator aufgenommen. Die übrigen Standorte wurden über die Vergabe des Datenidentifikators und die neuen Metadaten informiert, falls sie sofort erreichbar waren oder werden später informiert. Der letztere Fall wird durch die Verwendung einer persistenten Warteschlange sichergestellt.

Vorgangsbeschreibung

Abbildung 4.8 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System überprüft die Einfügerechte:
   Falls der Benutzer nicht über das Recht verfügt, Daten einzufügen, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

2. Das System zeigt die Dateneingabemaske.

3. Der Benutzer trägt die Position der Daten im Dateisystem in die Maske ein.

4. Der Benutzer trägt Metadaten in die Maske ein.

5. Der Benutzer bestätigt die eingegebenen Daten.

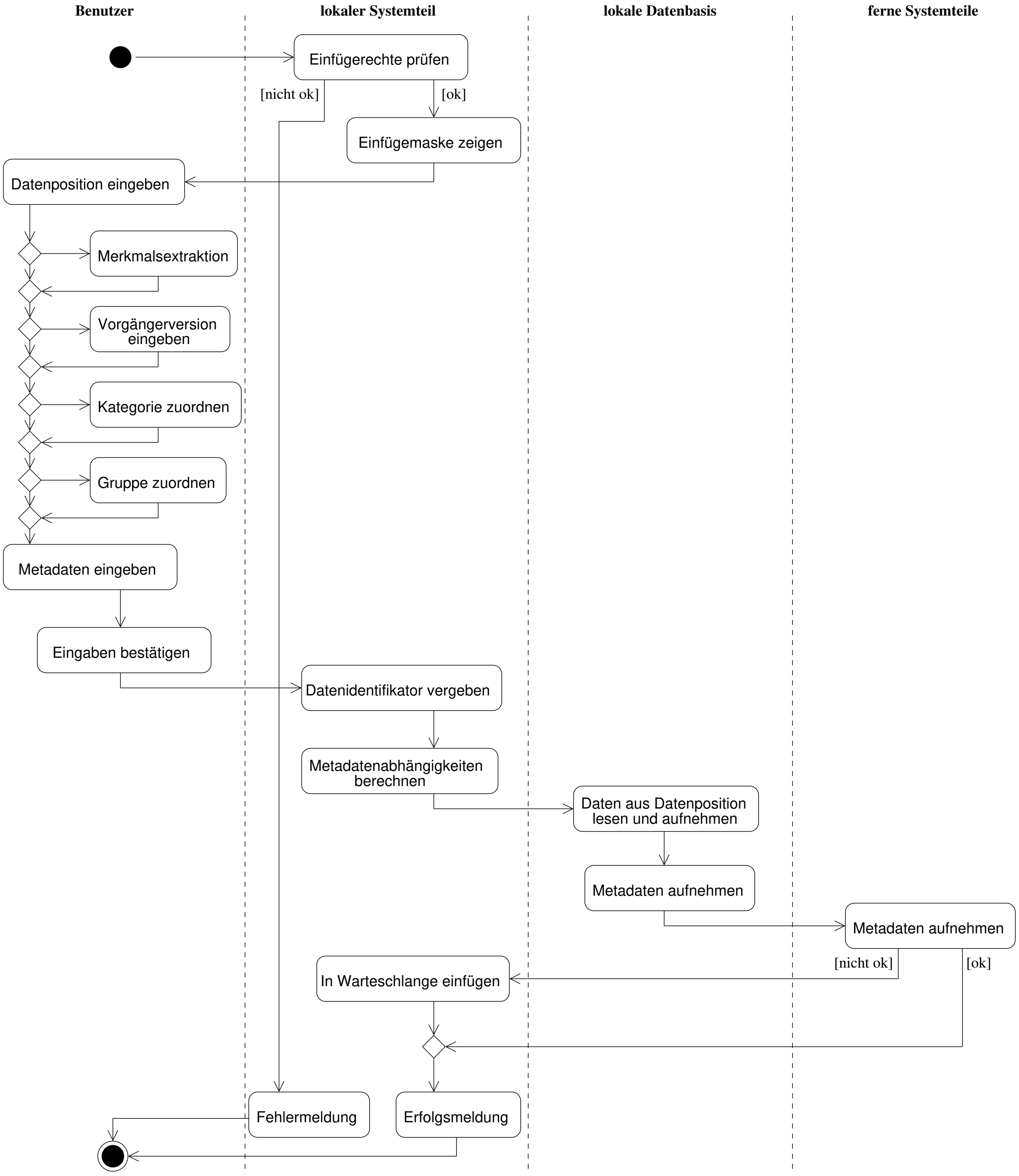

Abbildung 4.8: Aktivitätsdiagramm zu AF1 Dateneingabe (Storing)

6. Das System vergibt einen eindeutigen Datenidentifikator.

7. Das System berechnet Metadatenabhängigkeiten:

   Das System berechnet die neue Versionsnummer, falls eine Vorgängerversion deklariert wurde. Das System setzt die Standardrechte für den Zugriff auf diesen Datensatz.

8. Das System nimmt die Daten auf:

   Das System liest die Daten von der angegebenen Position und speichert sie. Im Zuge der Speicherung wird die für den Standort eingerichtete Hardwareeinbindung zum Speichern von Daten durchgeführt. Das System erweitert nach erfolgreicher Speicherung den Metadateneintrag der Standorte um den Speicherort.

9. Das System nimmt die Metadaten auf.

10. Das System informiert die übrigen Standorte:

    Das lokale System kontaktiert die übrigen Systemstandorte und teilt diesen die Vergabe des Datenidentifikators und die neu aufgenommenen Metadaten mit. Die übrigen Standorte nehmen die Metadaten auf.

11. Das System meldet den erfolgreichen Abschluss der Dateneingabe.

Variationen

4a. Merkmalsextraktion:

Statt die Metadaten selbst einzugeben, wird vom Benutzer die Merkmalsextraktion ausgelöst. Die davon gelieferten Daten werden automatisch in die Maske eingetragen und dabei eventuell bereits eingegebene Einträge überschrieben. Weiter bei Schritt 4.

4b. Versionsverwaltung:

Als Teil der Metadaten kann der Benutzer einen bereits im Archiv vorhandenen Datensatz als Vorgängerversion deklarieren. Weiter bei Schritt 4.

4c. Kategorienvergabe:

Als Teil der Metadaten kann der Benutzer den Daten eine Kategorie zuweisen. Die Kategorie muss zuvor im Zuge von AF4 angelegt worden sein. Weiter bei Schritt 4.

4d. Gruppenvergabe:

Als Teil der Metadaten kann der Benutzer die Daten als zu einer oder mehreren Gruppen zugehörig erklären. Die Gruppen müssen zuvor im Zuge von AF5 angelegt worden sein. Weiter bei Schritt 4.

10a. Standort nicht erreichbar:

Sollte ein anderer Standort nicht sofort erreichbar sein, so wird die Informationsübermittlung in eine persistente Warteschlange eingereiht[12]. Es wird von einem anderen Systemteil regelmäßig versucht, den nicht erreichbaren Standort zu kontaktieren. Sobald ein Kontakt hergestellt wurde, wird die Warteschlange abgearbeitet und abgebaut, indem die Informationen an den anderen Standort übermittelt werden. Weiter bei Schritt 11.

4.5.2 AF2 Datensuche (Query)

Auslöser und Vorbedingungen

Die Datensuche wird vom Benutzer ausgelöst. Der Benutzer muss dem System bekannt (angemeldet) sein.

Nachbedingungen

Der Benutzer erhielt eine Liste mit Datenidentifikatoren, die zu den angegebenen Suchkriterien passen.

[12] Persistent sei diese Warteschlange, damit die Information auch bei einem Ausfall oder Neustart des Systemteils am lokalen Standort erhalten bleibt (siehe 4.2.2).

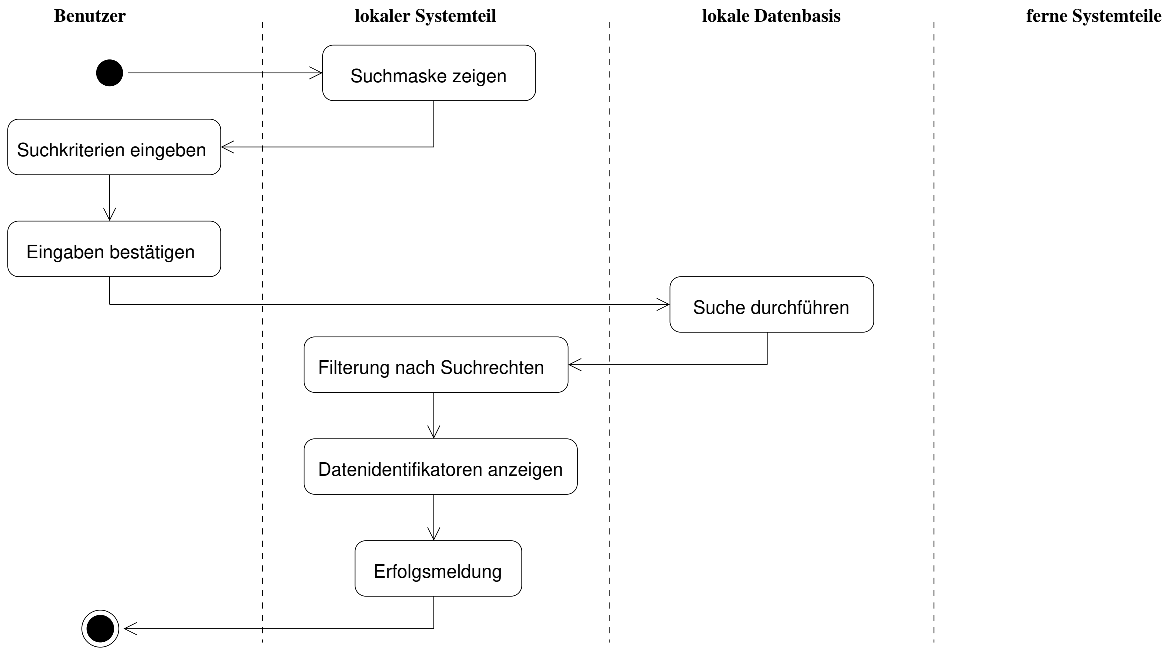

Abbildung 4.9: Aktivitätsdiagramm zu AF2 Datensuche (Query)

Vorgangsbeschreibung

Abbildung 4.9 zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System zeigt die Suchmaske.
2. Der Benutzer trägt Suchkriterien in die Maske ein.
3. Der Benutzer bestätigt die eingegebenen Kriterien.
4. Das System sucht im Metadatenbestand:
   Das System sucht nach Datensätzen, bei denen alle eingegebenen Kriterien zutreffen und erstellt eine Liste der zugehörigen Datenidentifikatoren.
5. Das System überprüft die Suchrechte:
   Das System überprüft jeden Eintrag in der Liste der Datenidentifikatoren daraufhin, ob der Benutzer ein Recht hat, diesen Eintrag zu finden. Falls er das Recht nicht innehat, wird der entsprechende Eintrag aus der Liste entfernt.

6. Das System meldet den erfolgreichen Abschluss der Datensuche:
   Das System meldet den Abschluss und liefert dem Benutzer die Liste der Datenidentifikatoren.[13]

Variationen

-keine-

4.5.3 AF3 Datenauslieferung (Retrieval)

Auslöser und Vorbedingungen

Die Datenauslieferung wird vom Benutzer ausgelöst. Der Benutzer muss dem System bekannt (angemeldet) sein. Der Benutzer muss den Datenidentifikator der gewünschten Daten kennen.

Nachbedingungen

Der Benutzer erhielt die angeforderten Daten.

Vorgangsbeschreibung

Abbildung 4.10 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System zeigt die Auslieferungsmaske.

2. Der Benutzer trägt die Position der erwarteten Daten ein:
   Der Benutzer trägt die Speicherposition ein, an der die Daten abgelegt werden sollen.

3. Der Benutzer trägt den Datenidentifikator ein.

4. Der Benutzer bestätigt seine Eingaben.

[13] Falls keine zu den eingegebenen Kriterien passenden Datensätze gefunden oder alle gefundenen Datensätze durch die Suchrechtefilterung entfernt werden, wird eine leere Liste zurückgeliefert.

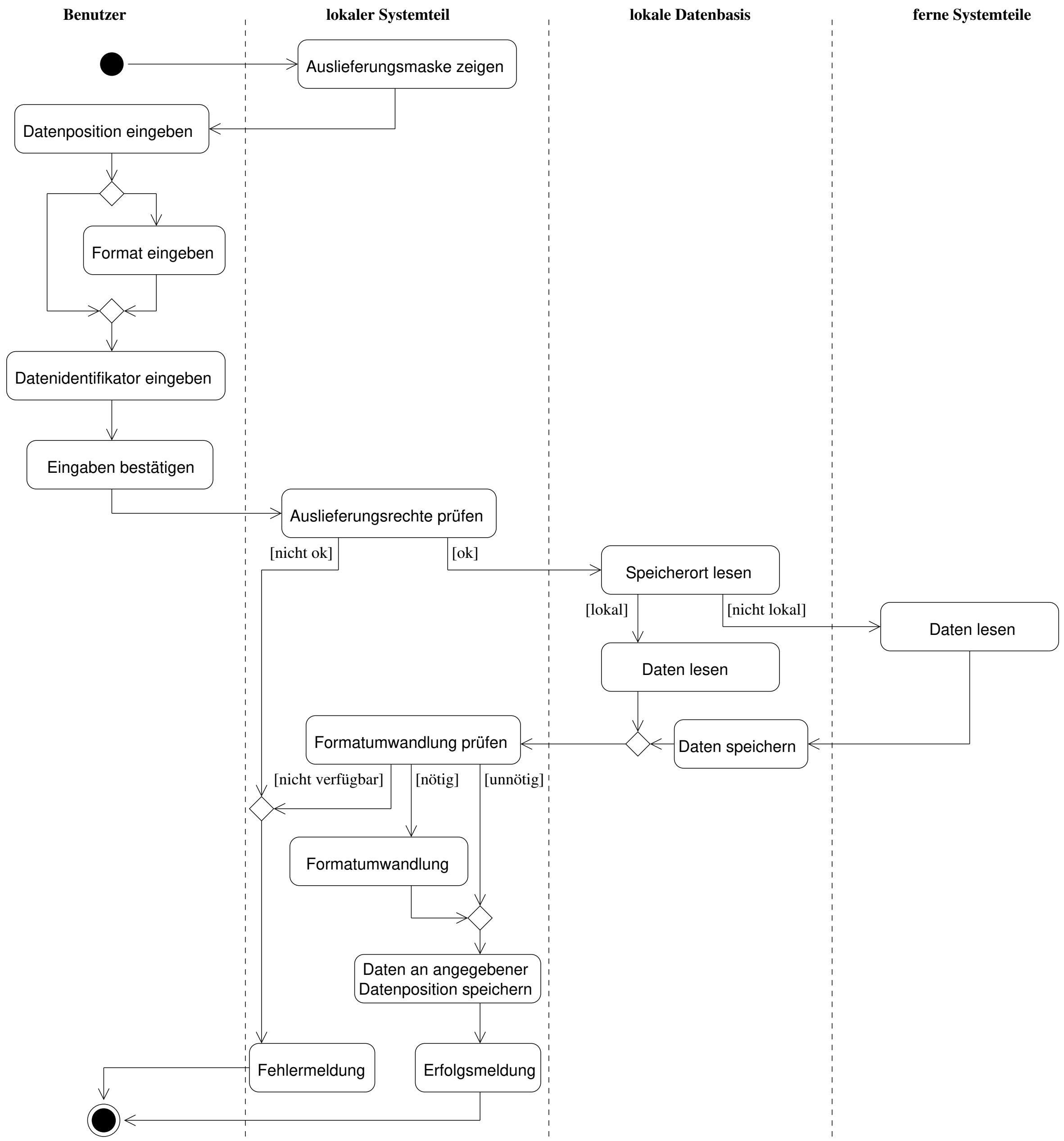

Abbildung 4.10: Aktivitätsdiagramm zu AF3 Datenauslieferung (Retrieval)

5. Das System überprüft die Auslieferungsrechte:

   Falls der Benutzer nicht über das nötige Recht an der Auslieferung der angeforderten Daten verfügt, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

6. Das System speichert die Daten:
   Das System liest die interne Datenpositon, liest hieraus die angeforderten Daten und speichert diese an der angegebenen Position. Im Zuge des Leseprozesses wird die für den die Daten speichernden Systemteil eingerichtete Hardwareeinbindung zum Lesen von Daten durchgeführt.

7. Das System meldet den erfolgreichen Abschluss der Datenauslieferung.

Variationen

3a. Der Benutzer gibt das gewünschte Auslieferungsformat an. Weiter bei Schritt 3.

6a. Ferner Standort:
Liegen die Daten nicht lokal vor, so werden die Daten von einem Standort, an dem diese verfügbar sind, gelesen und durch die lokale Hardwareeinbindung auch an diesem Standort zur Verfügung gestellt. Weiter bei Schritt 6.

6b. Formatübersetzung:
Hat der Benutzer ein anderes als das vorhandene Format als Auslieferungsformat angegeben, so überprüft das System, ob es eine Übersetzungsfunktion für die Umwandlung vom vorhandenen in das gewünschte Format kennt. Ist das der Fall, so werden die Daten mit dieser Funktion umgewandelt, weiter bei Schritt 6. Ist dem System keine Umwandlungsfunktion bekannt, so endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

4.5.4 AF4 Kategorienerweiterung

Auslöser und Vorbedingungen

Die Kategorienerweiterung wird vom Benutzer ausgelöst. Der Benutzer muss dem System bekannt (angemeldet) sein. Der Benutzer muss die Oberkategorie kennen, von der er eine neue Kategorie ableiten möchte.

Nachbedingungen

Die gewünschte neue Kategorie wurde als Unterkategorie der angegebenen Oberkategorie im System verzeichnet.

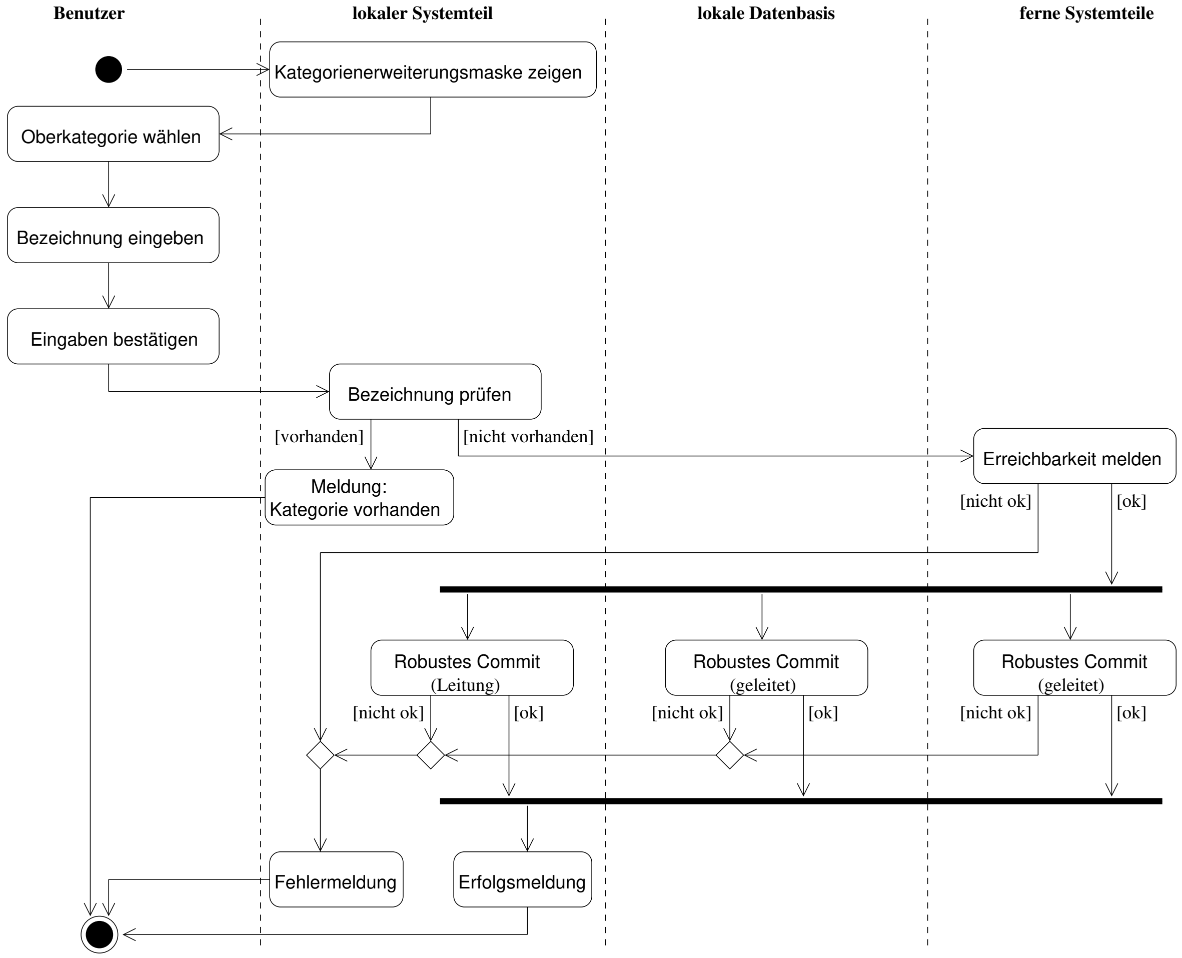

Abbildung 4.11: Aktivitätsdiagramm zu AF4 Kategorienerweiterung

Vorgangsbeschreibung

Abbildung 4.11 zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System zeigt die Kategorienerweiterungsmaske.

2. Der Benutzer wählt die Oberkategorie aus.

3. Der Benutzer gibt die Bezeichnung der neuen Kategorie ein.

4. Der Benutzer bestätigt die eingegebenen Daten.

5. Das System überprüft vorhandene Kategorien:
   Das System überprüft, ob bereits eine Kategorie mit der eingegebenen Bezeichnung existiert. Falls dieses zutrifft, endet der Vorgang hier mit einer entsprechenden Meldung.

6. Das lokale System informiert die übrigen Standorte:
   Das lokale System kontaktiert die übrigen Standorte. Falls mindestens ein Standort nicht erreichbar ist, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

7. Vom lokalen System wird ein Robustes Commit-Protokoll[14] geführt:
   Das System führt ein Robustes Commit für das Einfügen der neuen Kategorie mit den übrigen Standorten durch. Im Falle der Abweisung endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

8. Das System meldet den erfolgreichen Abschluss der Kategorienerweiterung.

Variationen

-keine-

4.5.5 AF5 Gruppenvergabe

Auslöser und Vorbedingungen

Die Gruppenerweiterung wird vom Benutzer ausgelöst. Der Benutzer muss dem System bekannt (angemeldet) sein.

[14] siehe Kapitel 4.2.3

Nachbedingungen

Die gewünschte Gruppe wurde im System verzeichnet. Die übrigen Standorte wurden über die Vergabe der Gruppe informiert, falls sie sofort erreichbar waren oder werden später informiert. Der letztere Fall wird durch die Verwendung einer persistenten Warteschlange sichergestellt.

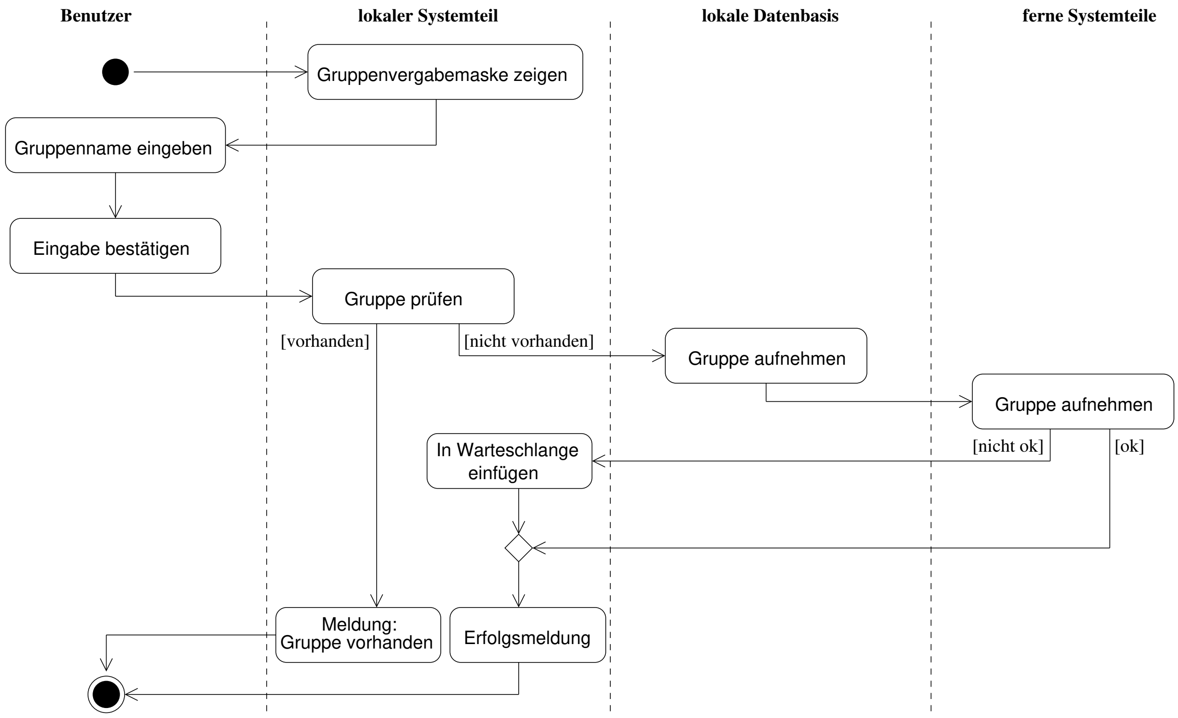

Abbildung 4.12: Aktivitätsdiagramm zu AF5 Gruppenvergabe

Vorgangsbeschreibung

Abbildung 4.12 zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System zeigt die Gruppenvergabemaske.
2. Der Benutzer gibt die Bezeichnung der neuen Gruppe ein.
3. Der Benutzer bestätigt die eingegebenen Daten.

4. Das System überprüft vorhandene Gruppen:
   Das System überprüft, ob bereits eine Gruppe mit der eingegebenen Bezeichnung existiert. Falls dieses zutrifft, endet der Vorgang hier mit einer entsprechenden Meldung.
5. Das System nimmt die Gruppe auf.
6. Das lokale System informiert die übrigen Standorte:
   Das lokale System kontaktiert die übrigen Systemstandorte und teilt diesen die Vergabe der Gruppe mit. Die übrigen Standorte nehmen die Gruppe auf.
7. Das System meldet den erfolgreichen Abschluss der Gruppenvergabe.

Variationen

6a. Standort nicht erreichbar:
Sollte ein anderer Standort nicht sofort erreichbar sein, so wird die Informationsübermittlung in eine persistente Warteschlange eingereiht. Es wird von einem anderen Systemteil regelmäßig versucht, den nicht erreichbaren Standort zu kontaktieren. Sobald ein Kontakt hergestellt wurde, wird die Warteschlange abgearbeitet und abgebaut, indem die Informationen an den anderen Standort übermittelt werden. Weiter bei Schritt 7.

6b. Gruppe schon vorhanden:
Falls an einem oder mehreren der übrigen Standorte bereits eine Gruppe mit gleichlautender Bezeichnung vorhanden ist, wird die Aufnahme der Gruppe dort ignoriert. Weiter bei Schritt 7.

4.5.6 AF6 Rechteverwaltung (Administration)

Auslöser und Vorbedingungen

Die Rechteverwaltung wird vom Administrator aufgerufen. Der Administrator muss dem System bekannt (angemeldet) sein. Der Administrator muss die Bezeichnung der

Benutzergruppe kennen, deren Rechte gesetzt werden sollen.

Nachbedingungen

Die Rechte der angegebenen Benutzergruppe wurden gesetzt.

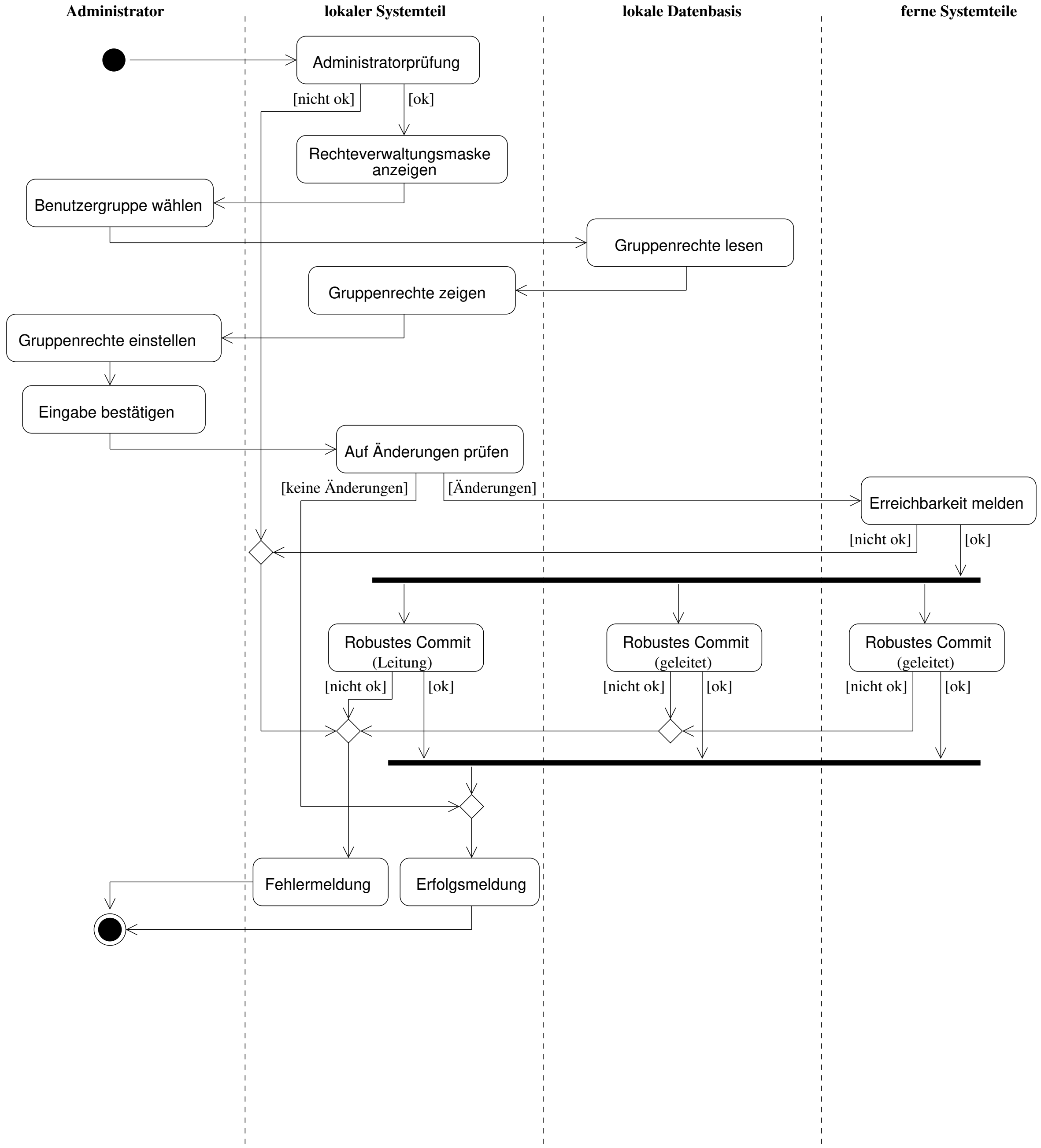

Abbildung 4.13: Aktivitätsdiagramm zu AF6 Rechteverwaltung

Vorgangsbeschreibung

Abbildung 4.13 auf der vorherigen Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System prüft auf Administrationsrechte:
   Das System prüft, ob der Benutzende ein Administrator ist. Falls dies nicht zutrifft, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

2. Das System zeigt die Rechteverwaltungsmaske.

3. Der Administrator wählt eine Benutzergruppe aus.

4. Das System zeigt die aktuellen Rechte:
   Das System liest die aktuell eingestellten Rechte der gewählten Benutzergruppe und stellt sie in der Maske dar.

5. Der Administrator stellt die Rechte ein.

6. Der Administrator bestätigt die Eingabe.

7. Das System prüft auf Änderungen:
   Das System prüft, ob die Rechte verändert wurden. Falls das nicht der Fall ist, weiter bei Schritt 10.

8. Das lokale System informiert die übrigen Standorte:
   Das lokale System kontaktiert die übrigen Standorte. Falls mindestens ein Standort nicht erreichbar ist, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

9. Vom lokalen System wird ein Robustes Commit-Protokoll geführt:
   Das System führt ein Robustes Commit für das Ändern der Rechte mit den übrigen Standorten durch. Im Falle der Abweisung endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

10. Das System meldet den erfolgreichen Abschluss der Rechteverwaltung.

Variationen

-keine-

4.5.7 AF7 Rechtegruppenverwaltung (Administration)

Auslöser und Vorbedingungen

Die Rechtegruppenverwaltung wird vom Administrator aufgerufen. Der Administrator muss dem System bekannt (angemeldet) sein. Der Administrator muss die Bezeichnung der Benutzergruppe kennen, deren Mitgliederliste aktualisiert werden sollen.

Nachbedingungen

Die Mitgliederliste der angegebenen Benutzergruppe wurden vom System aktualisiert.

Vorgangsbeschreibung

Abbildung 4.14 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System prüft auf Administrationsrechte:
   Das System prüft, ob der Benutzende ein Administrator ist. Falls dies nicht zutrifft, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

2. Das System zeigt die Rechtegruppenverwaltungsmaske.

3. Der Administrator wählt eine Benutzergruppe aus.

4. Das System zeigt die aktuellen Mitglieder:
   Das System liest die aktuell verzeichneten Mitglieder der gewählten Benutzergruppe und stellt sie in der Maske dar.

5. Der Administrator stellt die neue Mitgliederliste zusammen:
   Der Administrator wählt aus allen Benutzern die Liste der neuen Mitglieder aus.

6. Der Administrator bestätigt die Eingabe.

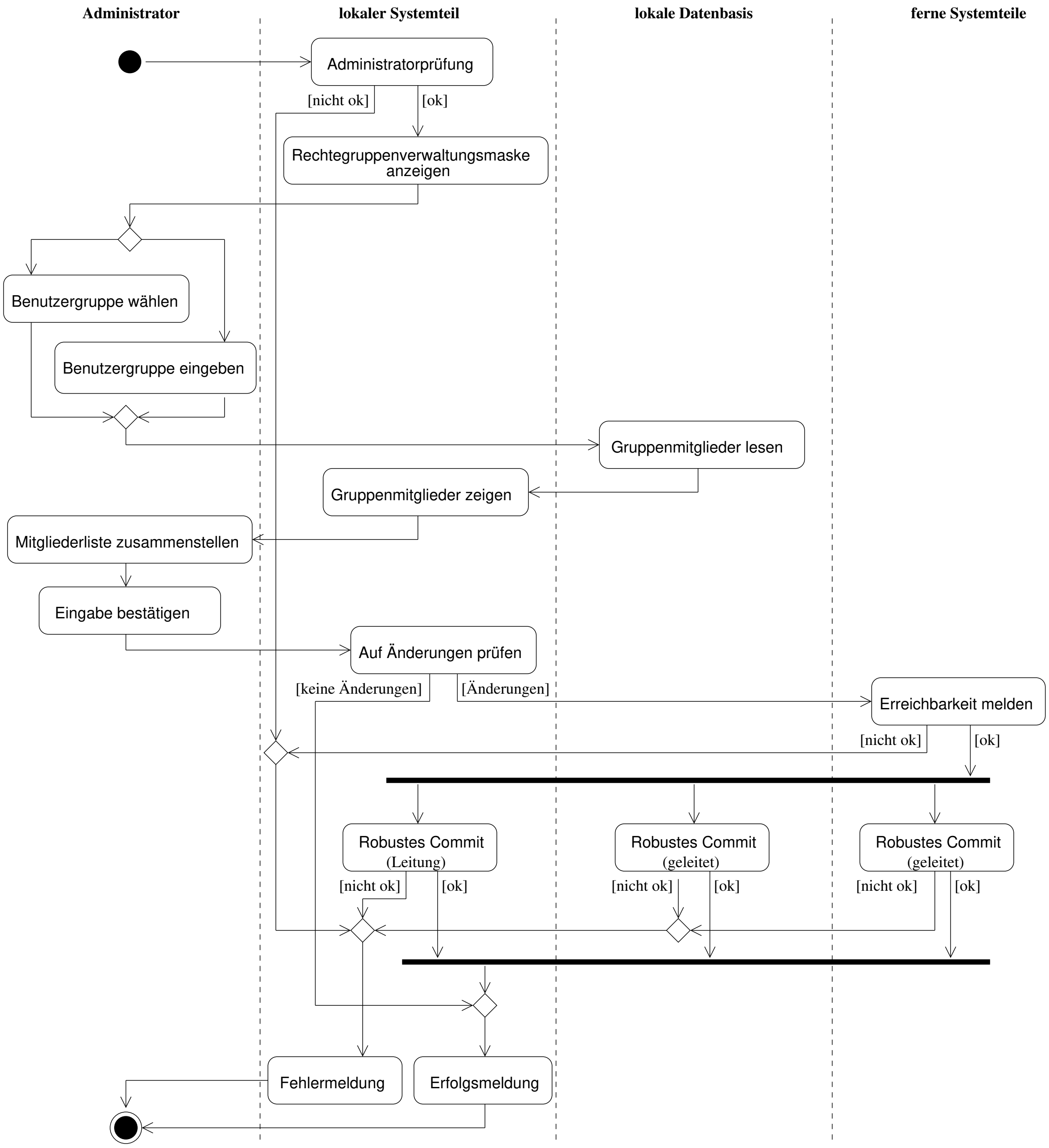

Abbildung 4.14: Aktivitätsdiagramm zu AF7 Rechtegruppenverwaltung

7. Das System prüft auf Änderungen:

   Das System prüft, ob die Mitgliederliste verändert wurde. Falls das nicht der Fall ist, weiter bei Schritt 10.

8. Das lokale System informiert die übrigen Standorte:

   Das lokale System kontaktiert die übrigen Standorte. Falls mindestens ein Standort nicht erreichbar ist, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

9. Vom lokalen System wird ein Robustes Commit-Protokoll geführt:

   Das System führt ein Robustes Commit für das Ändern der Mitgliederliste der Benutzergruppe mit den übrigen Standorten durch. Im Falle der Abweisung endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

10. Das System meldet den erfolgreichen Abschluss der Rechtegruppenverwaltung.

Variationen

3a. Neue Benutzergruppe:

Statt eine vorhandene Benutzergruppe zu wählen, kann der Administrator auch eine neue Benutzergruppenbezeichnung eingeben, um eine neue Gruppe anzulegen. Weiter mit Schritt 4.

5a. Leere Benutzergruppe:

Falls der Administrator die Mitgliederliste komplett leert, wird folgend keine Änderung sondern das Löschen der Benutzergruppe durchgeführt. Weiter bei Schritt 6.

7a. Änderungsprüfung einer neuen Benutzergruppe:

Falls eine neue Benutzergruppenbezeichnung eingegeben wurde, so gilt sie als unverändert im Sinne von Schritt 7, wenn die Mitgliederliste leer ist. In diesem Fall weiter bei Schritt 10, sonst weiter bei Schritt 8.

9a. Commit einer neuen Benutzergruppe:

Das Robuste Commit-Protokoll aus Schritt 9 umfasst nicht nur die Änderung der Mitgliederliste sondern auch das vorherige Anlegen der neuen Benutzergruppe. Weiter bei Schritt 10.

9b. Commit einer geleerten Benutzergruppe:
Ist die Mitgliederliste leer, so umfasst das Robuste Commit-Protokoll aus Schritt 9 nicht das Ändern der Mitgliederliste sondern das Entfernen der Benutzergruppe, es sei denn, es handelt sich um die spezielle Benutzergruppe „newusers", welche nicht gelöscht wird[15]. Dabei ist die referentielle Integrität der den Daten zugeordneten Rechte zu beachten und zu erhalten. Weiter bei Schritt 10.

4.5.8 AF8 Erweiterung der Formatumwandlungsmöglichkeiten (Administration)

Auslöser und Vorbedingungen

Die Erweiterung der Formatumwandlungsmöglichkeiten wird vom Administrator aufgerufen. Der Administrator muss dem System bekannt (angemeldet) sein. Die neue Formatumwandlungsfunktion muss im Formatumwandlungspluginverzeichnis des Systems abgelegt worden sein.

Nachbedingungen

Die neue Formatumwandlungsfunktion ist dem System bekannt.

Vorgangsbeschreibung

Abbildung 4.15 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System prüft auf Administrationsrechte:
   Das System prüft, ob der Benutzende ein Administrator ist. Falls dies nicht zutrifft, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

2. Das System zeigt die Maske der Formatumwandlungsmöglichkeitenerweiterungen.

[15] Siehe dazu auch die Vorgangsbeschreibung zu AF10.

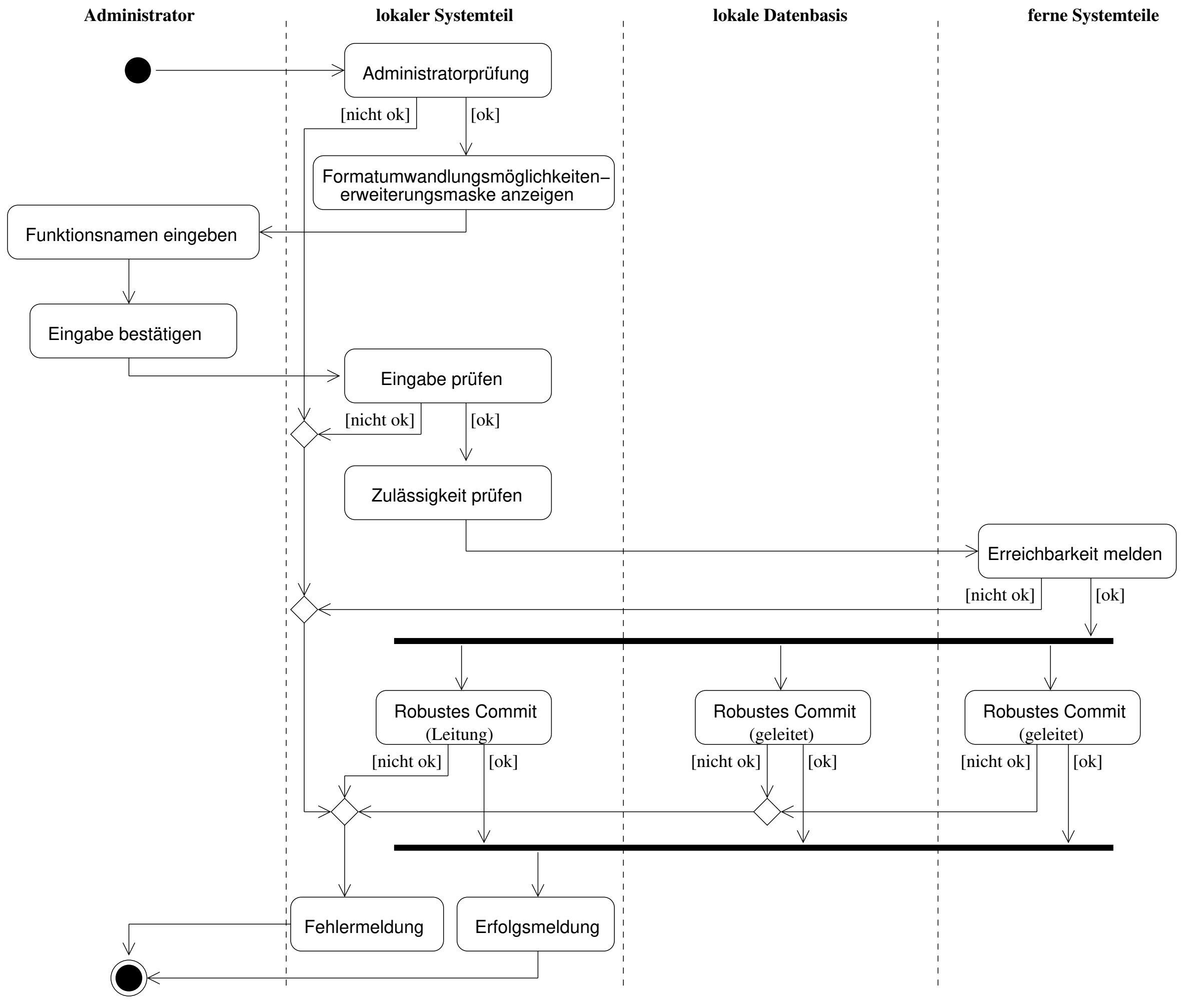

Abbildung 4.15: Aktivitätsdiagramm zu AF8 Erweiterung der Formatumwandlungsmöglichkeiten

3. Der Administrator trägt den Funktionsnamen ein.

4. Der Administrator bestätigt die Eingabe.

5. Das System prüft die Eingabe:
   Das System liest die Funktion ein. Falls diese Aktion fehlschlägt, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

6. Das System prüft die Zulässigkeit:

   Das System liest die Angaben zu Quell- und Zielformat aus der Funktionsbeschreibung. Das System prüft, ob bereits eine Umwandlungsfunktion vom Quell- ins Zielformat bekannt ist. Das Ergebnis der Prüfung wird für die Verwendung in Schritt 8a. vermerkt.

7. Das lokale System informiert die übrigen Standorte:

   Das lokale System kontaktiert die übrigen Standorte. Falls mindestens ein Standort nicht erreichbar ist, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

8. Vom lokalen System wird ein Robustes Commit-Protokoll geführt:

   Das System führt ein Robustes Commit für das Ändern der Daten über bekannte Übersetzungsfunktionen mit den übrigen Standorten durch. Im Zuge des Protokolls wird die neue Formatumwandlungsfunktion aus dem Dateisystem des lokalen Standorts an die übrigen Standorte übermittelt. Im Falle der Abweisung endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

9. Das System meldet den erfolgreichen Abschluss der Erweiterung der Formatumwandlungsmöglichkeiten.

Variationen

8a. Vorhandene Umwandlungsfunktion:

Falls bereits eine Umwandlungsfunktion der betreffenden Formate existierte (dies wurde in Schritt 6 vermerkt), wird die alte Umwandlungsfunktion im Zuge des Robusten Commits aus Schritt 8 vor dem Einbinden der neuen Umwandlungsfunktion entfernt. Weiter bei Schritt 9.

4.5.9 AF9 Erweiterung der Merkmalsextraktionsmöglichkeiten (Administration)

Auslöser und Vorbedingungen

Die Erweiterung der Merkmalsextraktionsmöglichkeiten wird vom Administrator aufgerufen. Der Administrator muss dem System bekannt (angemeldet) sein. Die neue Merkmalsextraktionsfunktion muss im Merkmalsextraktionsspluginverzeichnis des Systems abgelegt worden sein.

Nachbedingungen

Die neue Merkmalsextraktionsfunktion ist dem System bekannt.

Vorgangsbeschreibung

Abbildung 4.16 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System prüft auf Administrationsrechte:
   Das System prüft, ob der Benutzende ein Administrator ist. Falls dies nicht zutrifft, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

2. Das System zeigt die Maske der Merkmalsextraktionsmöglichkeitenerweiterungen.

3. Der Administrator trägt den Funktionsnamen ein.

4. Der Administrator bestätigt die Eingabe.

5. Das System prüft die Eingabe:
   Das System liest die Funktion ein. Falls diese Aktion fehlschlägt, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

6. Das System prüft die Zulässigkeit:
   Das System liest die Angaben zum Quellformat aus der Funktionsbeschreibung.

Das System prüft, ob bereits eine Extraktionsfunktion für das Quellformat bekannt ist. Das Ergebnis der Prüfung wird für die Verwendung in Schritt 8a. vermerkt.

7. Das lokale System informiert die übrigen Standorte:
   Das lokale System kontaktiert die übrigen Standorte. Falls mindestens ein Standort nicht erreichbar ist, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

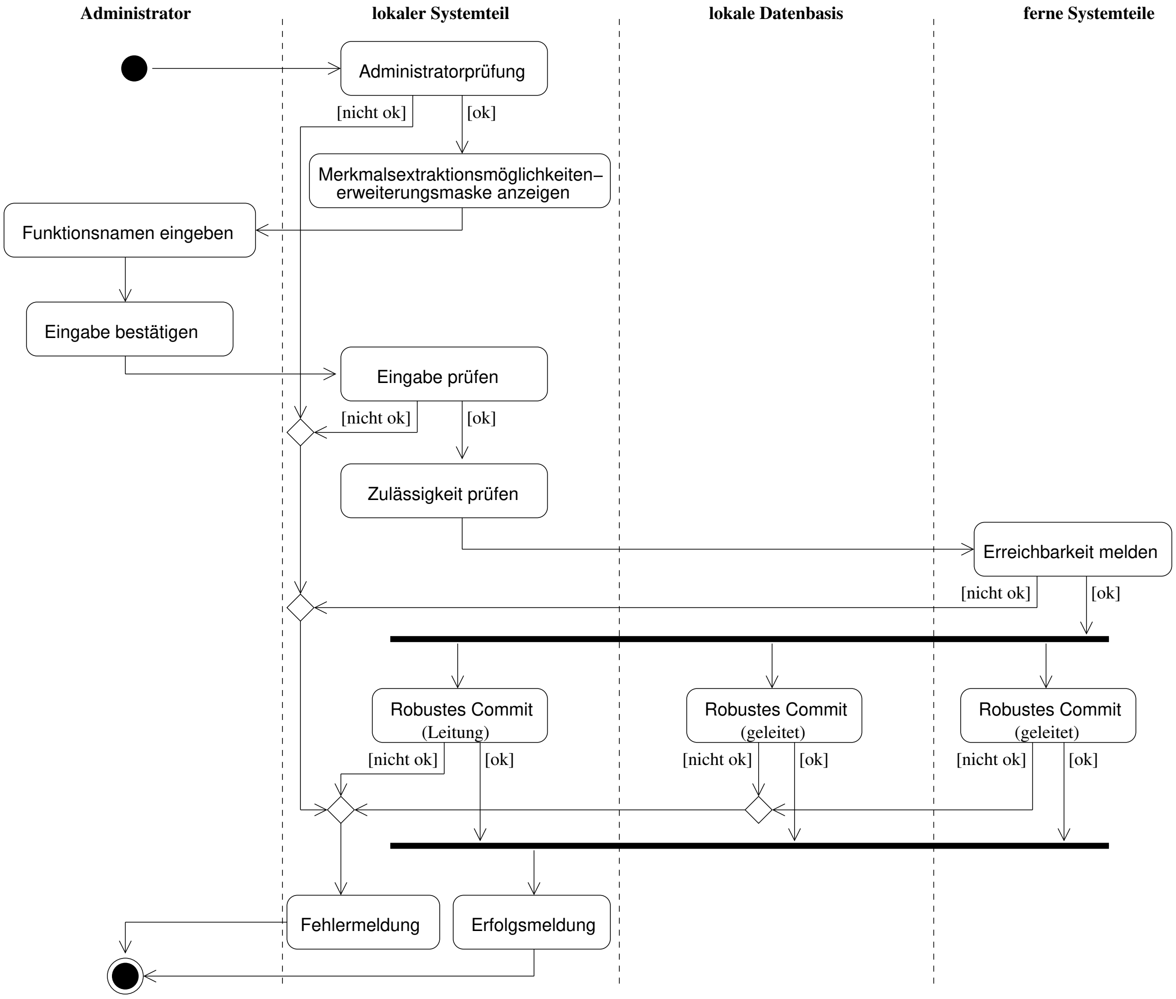

Abbildung 4.16: Aktivitätsdiagramm zu AF9 Erweiterung der Merkmalsextraktionsmöglichkeiten

8. Vom lokalen System wird ein Robustes Commit-Protokoll geführt:
   Das System führt ein Robustes Commit für das Ändern der Daten über bekannte Extraktionsfunktionen mit den übrigen Standorten durch. Im Zuge der Protokollabarbeitung wird die neue Merkmalsextraktionsfunktion aus dem Dateisystem des lokalen Standorts an die übrigen Standorte übermittelt. Im Falle der Abweisung endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

9. Das System meldet den erfolgreichen Abschluss der Erweiterung der Merkmalsextraktionsmöglichkeiten.

Variationen

8a. Vorhandene Extraktionsfunktion:
Falls bereits eine Extraktionsfunktion zum betreffenden Format existierte (dies wurde in Schritt 6 vermerkt), wird die alte Extraktionsfunktion im Zuge des Robusten Commits aus Schritt 8 vor dem Einbinden der neuen Extraktionsfunktion entfernt. Weiter bei Schritt 9.

4.5.10 AF10 Anmeldung

Auslöser und Vorbedingungen

Die Anmeldung wird vom Benutzer ausgelöst. Der Benutzer ist dem System nicht bekannt (nicht angemeldet).

Nachbedingungen

Der Benutzer wurde authentifiziert und gilt als dem System bekannt (angemeldet).

Vorgangsbeschreibung

Abbildung 4.17 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

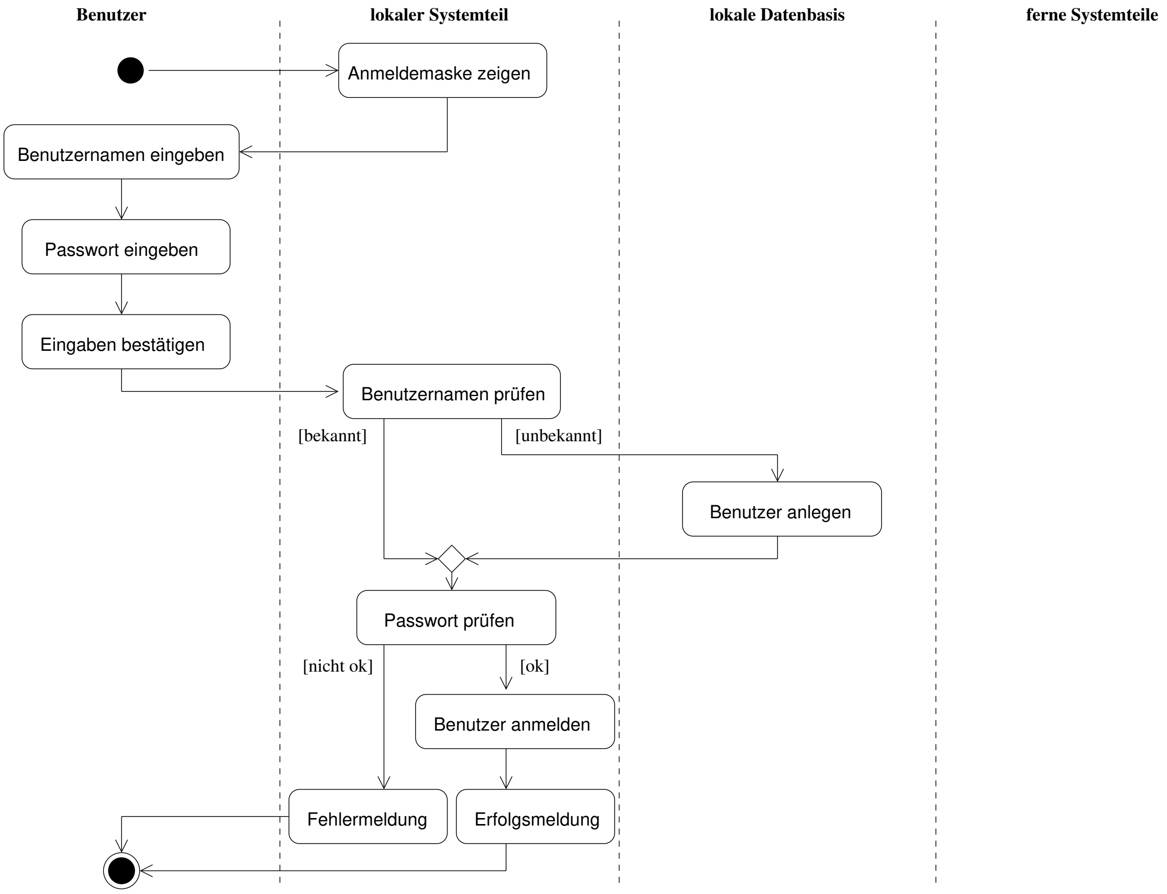

Abbildung 4.17: Aktivitätsdiagramm zu AF10 Anmeldung

1. Das System zeigt die Anmeldemaske.
2. Der Benutzer gibt seinen Benutzernamen ein.
3. Der Benutzer gibt sein Benutzerpasswort ein.
4. Der Benutzer bestätigt die Eingaben.
5. Das System überprüft den Benutzernamen:

   Das System prüft, ob ein Benutzer des angegebenen Namens bekannt ist. Falls das nicht der Fall ist, wird ein neuer Benutzer mit diesem Namen angelegt und der Benutzergruppe „newusers"[16] zugewiesen sowie das Passwort gespeichert.[17]

6. Das System überprüft das Benutzerpasswort:
   Das eingegebene Passwort wird mit dem zum Benutzer gespeicherten Passwort verglichen. Bei einem erfolglosen Vergleich endet der Vorgang hier mit einer entsprechenden Fehlermeldung.
7. Das System führt den Benutzer als angemeldet.
8. Das System meldet den erfolgreichen Abschluss der Anmeldung.

Variationen

-keine-

4.5.11 AF11 Abmeldung

Auslöser und Vorbedingungen

Der Benutzer ist dem System bekannt (angemeldet).

Nachbedingungen

Der Benutzer ist dem System nicht mehr bekannt (abgemeldet).

Vorgangsbeschreibung

Abbildung 4.18 auf der nächsten Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

[16] Diese Benutzergruppe muss permanent vorhanden sein und darf somit in Anwendungsfall AF7 nicht gelöscht werden

[17] Diese Vorgehensweise kann als sicher vor unerlaubter Dateneinsicht eingestuft werden, wenn der Gruppe „newusers" alle entsprechenden Rechte entzogen werden. So hat der Benutzer, bis ihm die Administration eine andere Gruppe zuteilt, keine Möglichkeit die Daten einzusehen, kann aber seine Daten (bei einem nichtprototypischen System werden vermutlich mehr Daten als nur der Name und das Passwort des Benutzers gespeichert) ohne die Hilfe des Administrators in Anspruch nehmen zu müssen, selbständig hinterlegen.

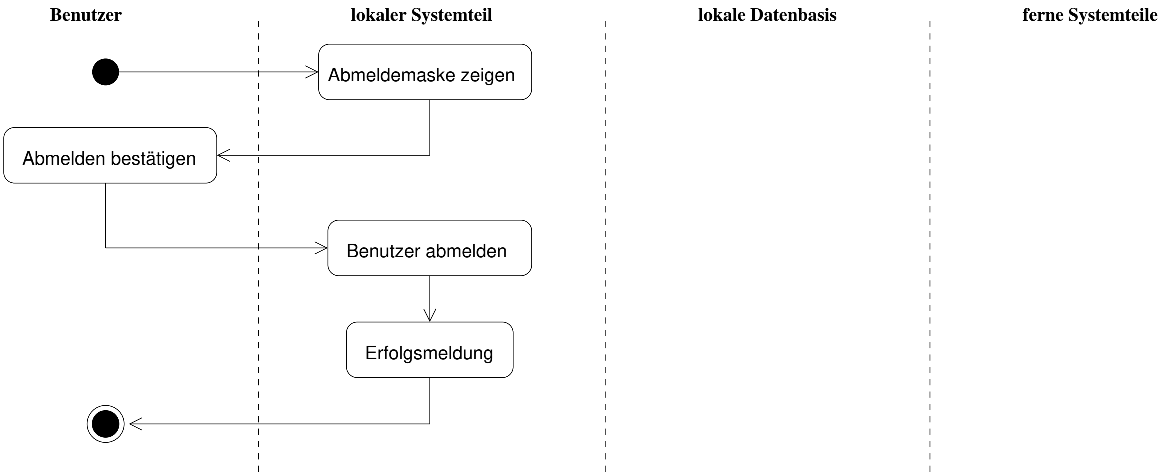

Abbildung 4.18: Aktivitätsdiagramm zu AF11 Abmeldung

1. Das System zeigt die Abmeldemaske.
2. Der Benutzer bestätigt das Abmelden.
3. Das System führt den Benutzer nicht mehr als angemeldet.
4. Das System meldet den erfolgreichen Abschluss der Abmeldung.

Variationen

-keine-

4.5.12 AF12 Standorterweiterung

Auslöser und Vorbedingungen

Die Standorterweiterung wird vom Administrator am neuen Standort ausgelöst. Die Programmdateien müssen am neuen Standort installiert sein. Die vom Programm benötigten Dienste (siehe dazu die Ausführungen in Kapitel 5) müssen verfügbar sein.

Nachbedingungen

Alle Standorte des Gesamtsystems kennen den neuen Standort. Die allgemeinen Metadaten und Verwaltungsdaten sind am neuen Standort vorhanden (repliziert). Das System am neuen Standort ist einsatzbereit. Das Gesamtsystem ist einsatzbereit.

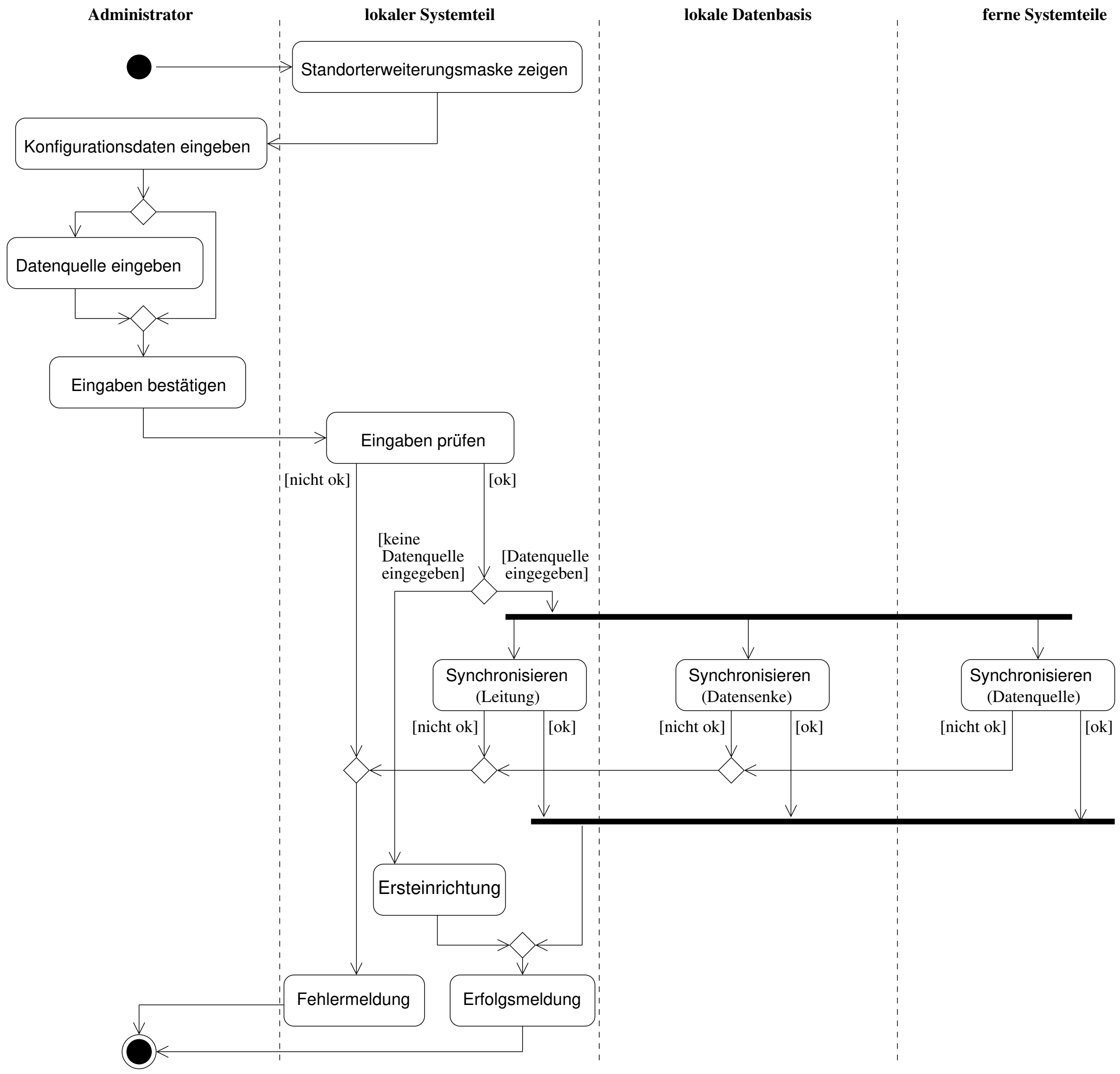

Abbildung 4.19: Aktivitätsdiagramm zu AF12 Standorterweiterung

Vorgangsbeschreibung

Abbildung 4.19 auf der vorherigen Seite zeigt, an welchen Stellen die folgenden Vorgänge aktiv durchgeführt werden.

1. Das System zeigt die Standorterweiterungsmaske.

2. Der Administrator trägt seine Benutzerbezeichnung und sein Passwort ein.

3. Der Administrator trägt Konfigurationsdaten ein:
   Der Administrator trägt Daten zur Konfiguration des neuen Standorts (Position der Datenbank, Datenablageverzeichnis, Hardwareeinbindungsmethoden, ...) ein.

4. Der Administrator trägt einen Standort ein:
   Der Administrator trägt die Position eines bereits im System befindlichen Standorts als Datenquelle für die Synchronisation ein.

5. Der Administrator bestätigt die Eingaben.

6. Das System prüft die Eingaben:
   Das System prüft und speichert die Eingaben aus Schritt 2. Falls die Prüfung fehlschlägt, endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

7. Das System synchronisiert Verwaltungs- und Metadaten:
   Das lokale System kontaktiert das System am als Datenquelle angegebenen Standort und synchronisiert die Verwaltungs- und Metadaten. In diesem Prozess werden vom Datenquellsystem auch die Positionen der weiteren Systemstandorte ermittelt und die eigene Position den anderen Standorten bekanntgegeben. Im Fehlerfall endet der Vorgang hier mit einer entsprechenden Fehlermeldung.

8. Das System meldet den Abschluss der Standorterweiterung.

Variationen

3a Erster Standort:

Der Administrator läßt das Eingabefeld für den Standort der Datenquelle leer. Weiter bei Schritt 4.

6a Erster Standort:

Wurde das Eingabefeld für den Standort der Datenquelle in Schritt 3 bzw. 3a leer gelassen, so wird der Synchronisationsschritt übersprungen. Es werden stattdessen die Benutzerdaten des Administrators aufgenommen sowie die Erstdaten erstellt. Weiter bei Schritt 7.

4.5.13 Anwendungsfalldiagramm

Die Beziehungen zwischen Benutzern und Anwendungsfällen sowie die Beziehungen zwischen den Anwendungsfällen zeigt Abbildung 4.20 auf der nächsten Seite.

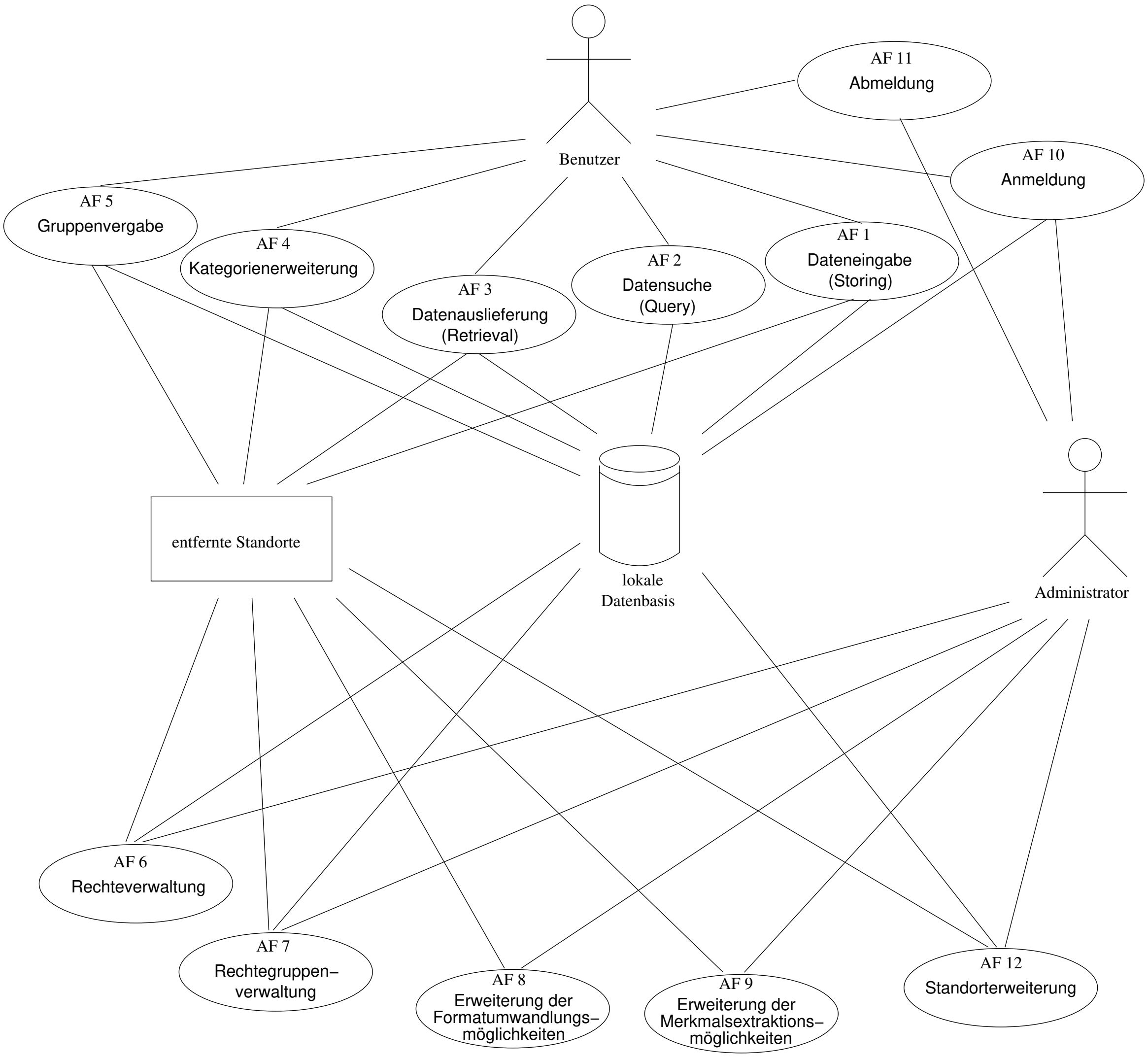

Abbildung 4.20: Beziehungen zwischen Anwendungsfällen und Benutzer, Administrator sowie lokaler Datenbasis und weiteren Systemteilen an entfernten Standorten

Die Anwendungsfälle AF1 bis AF5 werden ausschließlich vom Benutzer ausgelöst und benutzt. Die Anwendungsfälle AF6 bis AF9 sowie AF12 sind ausschließlich für die Administration des Systems vorgesehen und zugänglich zu machen. Die Anwendungsfälle AF10 und AF11 werden sowohl von Administratoren als auch von nicht administrativ tätigen Benutzern verwendet.

4.6 Vergleich mit den Anforderungen

Es bleibt zu zeigen, dass *alle* Anforderungen aus Kapitel 3 vom konzipierten System erfüllt werden:

4.6.1 Verlässlichkeit

Das Hauptaugenmerk bei der Konzeption lag auf der Verlässlichkeit des Systems, welche durch die Forderungen nach Datensicherheit und einheitlicher Rechteverwaltung ausgedrückt wird.

Datensicherheit

Auf Datensicherheit wurde großen Wert gelegt. Das System kann lokal vorhandene Daten in jedem Fall wieder lokal ausliefern, auch wenn die Verbindung zu anderen Standorten gestört sein sollte. Als Schutz vor Veränderungen an den Daten selbst wird mit Prüfsummen durch geeignete Message-Digest-Funktionen gearbeitet. Änderungen an der Datenbasis können nur durch das Einfügen neuer Daten durch Anwendungsfall AF1 geschehen, welches sich selbst aber nicht auf bereits vorhandene Daten auswirkt und der insert-Funktion folgt.
Die Auslieferung der nicht lokal gespeicherten Daten wird mit persistenten Warteschlangen garantiert. Der Forderung nach Eindeutigkeit der Datenidentifikatoren auch bei der Verteilung auf mehrere Standorte dient die in Abschnitt 4.3 entwickelte Methode zur Berechnung der ident-Funktion.

Rechteverwaltung

Die in den Anforderungen beschriebene minimal nötige Rechteverwaltung kann durch das Robuste Commit-Protokoll und dem ACID-Konzept folgende lokale Datenbanksysteme zur Abänderung von Benutzergruppen und Benutzergruppenzugehörigkeit (siehe Anwendungsfall AF7) und der den Benutzergruppen zugeordneten Rechte (siehe Anwendungsfall AF6) direkt umgesetzt werden.

4.6.2 Skalierbarkeit

Eine weiter wichtige Stelle im Konzept ist das Erhalten der Skalierbarkeit bei der verteilten Umsetzung des Gesamtsystems.

Verteiltes System

Den Punkten 1) und 2) aus den Anforderungen zum verteilten Systemcharakter ist Rechnung getragen worden: Daten werden nur auf Anfrage an einen fernen Standort übertragen. Dies geschieht einmalig, denn nach der Übertragung werden sie vom anfragenden Standort lokal gespeichert (siehe Anwendungsfall AF3) und brauchen fortan nicht erneut übertragen werden. Die Metadaten hingegen werden sofort mit ihrer Aufnahme ins System – ebenfalls einmalig – repliziert, da sie für die Datensuche unerlässlich sind. Wäre die Suche anders konzipiert worden, so dass jeder Standort bei einer Suche erst bei den anderen Standorten hätte nachfragen müssen, so stiege dadurch der Kommunikationsbedarf erheblich und unnötigerweise an.
Punkt 3) der Anforderungen zum verteilten Systemcharakter behandelt das Verhalten bei zeitweiliger Nichterreichbarkeit eines oder mehrerer ferner Standorte. Es soll auch bei solch einem Ausfall mit eingeschränktem aber lokal verfügbarem Datenbestand weitergearbeitet werden können. Diese Forderung wird erfüllt:

- Es können neue Daten ins System aufgenommen werden. Durch die Gestaltung der ident-Funktion als lokal berechenbar und der Ergebnisse als global eindeutig werden Konflikte durch gleichnamige Identifikatoren von vornherein vermieden. Die persistenten Warteschlangen sichern die baldestmögliche Zustellung der eingefügten Daten zu.

- Es kann jederzeit eine Datensuche über den gesamten zum Zeitpunkt des Verbindungsausfalls bekannten Datenbestand und darüber hinaus der seitdem lokal eingefügten Daten durchgeführt werden, da alle dazu benötigten Daten lokal repliziert vorliegen.

- Eine Datenauslieferung von am Standort lokal vorhandenen Daten kann während des Verbindungsausfalls ebenfalls geschehen, da hierfür keine weitere Kommunikation mit anderen Standorten erforderlich ist. Eine Auslieferung von an einem abgetrennten Standort gespeicherten Daten führt hingegen zu einer Fehlermeldung.

- Kategorien können nicht bei einem Verbindungsausfall vergeben werden, da dafür eine synchrone Ableitung nötig ist.
- Gruppen können neu erstellt werden, sie werden durch die Eigenschaften der persistenten Warteschlangen baldestmöglich an alle Standorte übermittelt.
- Administrative Aufgaben können in der Ausfallzeit nicht vorgenommen werden, da dafür eine synchrone Behandlung aller Standorte vonnöten ist.

Versionskontrolle

Die Versionskontrolle ist direkt in den Metadaten verankert. Eine neue Version kann bei Bedarf im Zuge des Einfügevorgangs von Anwendungsfall AF1 durch die Angabe des Identifikators der Vorgängerversion gekennzeichnet werden. Das Berechnen der Versionsnummer wird analog zu der in den Anforderungen genannten Funktion version im Schritt 7 „Metadatenabhängigkeiten berechnen" der Dateneingabe in Anwendungsfall AF1 ausgeführt.

Hardwareeinbindung

Die Hardwareeinbindung ist Voraussetzung für die Datenablage. Sie kann frei implementiert werden und zum Wiederauffinden der Daten benötigte Informationen werden im Attribut „Repertorieneintrag" im Zuge des Einfügevorgangs abgelegt. Dabei kann an jedem Standort eine verschiedene Methode benutzt werden. Die Hardwareeinbindung muss neben der Aufnahme der Daten aus dem lokalen Dateisystem auch die Auslieferung dorthin implementieren.

4.6.3 Automation

Formatübersetzung

Die Forderungen zur Formatübersetzung werden erfüllt: Es können bekannte Formate im Datenbestand abgelegt werden. Darauf basierend kann die Information zu Quell- und Zielformat sowie ein von jeder Erweiterung zu definierender Funktionsname verzeichnet werden. Über den Funktionsnamen kann dann ein geeignet zu implementierendes Pluginsystem die Übersetzungsfunktion auffinden. Durch das im Anwendungsfall AF8 benutzte Robuste Commit-Protokoll wird zudem die Verfügbarkeit an allen Standorten hergestellt, wo die Übersetzung dann durch lokale Berechnung im Zuge der Datenauslieferung durchgeführt werden kann.

Merkmalsextraktion

Die Auswahl der richtigen Merkmalsextraktion basiert wie die Formatübersetzung auf dem Bekanntsein des Quellformats. Es wurden analog zur Formatübersetzung ein frei definierbarer Funktionsname für die Merkmalsextraktion in das Datenmodell aufgenommen. Im Anwendungsfall AF9 wird das Robuste Commit-Protokoll zur Zurverfügungstellung der Funktion an allen Standorten benutzt, an denen die Merkmalsextraktion dann durch lokale Berechnung im Zuge der Dateneinlieferung vorgenommen werden kann.

4.6.4 Gestaltungsfreiheit

Kategorisierbarkeit

Als Mittel zur Kategorisierung der Daten stehen die Einordnung selbiger in eine hierarchische Kategorienstruktur sowie die Zu- und Einteilung in Datengruppen zur Auswahl. Beide ordnungsgebenden Maßnahmen können vom Benutzer frei definiert werden, wobei als Mindesteintrag in der Kategorienhierarchie „Allgemeine Daten" als Wurzelknoten vorgegeben wird. Dieser Eintrag entspricht der in den Anforderungen aus Abschnitt 3.5.1 vorgeschriebenen geeigneten Initialisierung der Kategorienmenge, denn er schränkt die Benutzbarkeit und freie Gestaltbarkeit nicht ein.

Allgemeinheit bei Spezialisierbarkeit

Es wurde darauf geachtet, dass kein Teil des Systems die Anwendbarkeit für allgemeine Daten einschränkt. Sind mehr Informationen über das Datenformat bekannt, so kann auf diese Spezialisierung bei Vorhandensein entsprechend passender mit der Ausführung dieser Funktionen reagiert werden. Die Spezialisierbarkeit ist also bei Beibehaltung der allgemeinen Anwendbarkeit gegeben.

Damit sind alle Anforderungen an die Konzeption erfüllt.

4.7 Zusammenfassung

In diesem Kapitel wurden die verschiedenen Aspekte der Konzeption mit Rücksicht auf die Anforderungen betrachtet. Das konzipierte System gliedert sich in Dienste und Benutzerschnittstelle, welche durch ein Dämonprogramm und eine GUI ausgeführt werden. Es baut auf den Diensten eines lokalen Datenbanksystems auf und kann im Funktionsumfang durch neue Formatübersetzungs- und Merkmalsextraktionsfunktionen erweitert werden.
Das Dämonprogramm muss an jedem Standort des Systems möglichst ständig laufen, die GUI kann bei Bedarf dazugeschaltet werden, um Benutzeraktionen zu tätigen. Die GUI kommuniziert nur mit den lokal vorhandenen Diensten, denn für die Kommunikation zwischen den Standorten ist allein der jeweilige Dämon zuständig, welcher mit seinesgleichen Kommunikationsdaten auszutauschen vermag.

Das Gesamtsystem kommt ohne feste Hierarchien aus. Diese werden nur temporär und nur bei Bedarf für kurze Zeit aufgebaut. Dabei fällt dem die jeweilige Aktion auslösenden Standort die Leitung über die und der anderen Teilnehmer zu.

Für die Absicherung der Kommunikation zwischen den Standorten vor nicht dauerhaften Erreichbarkeitsproblemen, beispielsweise durch Ausfall eines Netzwerks oder des Computers, auf dem ein Standortdienst läuft, wird eine Zwischenspeicherung in persistentem Speicher auf verschiedene Weisen genutzt: Eine Garantie der Zustellung von Nachrichten wird durch persistent gesicherte Warteschlangen erreicht. Im neu entwickelten Robusten Commit-Protokoll wird zudem der persistente Speicher für das Verzeichnen des aktuellen Protokollstatus für eine mögliche Wiederherstellung genutzt sowie auf die persistent gesicherten Warteschlangen für den Protokollabschluss zurückgegriffen. Es läßt sich zeigen, dass dadurch eine sichere Kommunikation und Synchronisation erreicht werden kann. Ausgefallene Teile des Gesamtsystems können ihren Zustand mit Hilfe der im persistenten Speicher mitgeschriebenen Informationen sicher wiederherstellen.
Für die Anforderungen zur ident Funktion läßt sich eine praktikable Methode finden, die diese Anforderungen auch in einem verteilten System bei lokaler Berechenbarkeit erfüllt.

Es wurde ferner der Entwurfsprozess für das verwendete Datenmodell des zugrunde liegenden Datenbanksystems dargestellt. In der Anforderungsanalyse wurden zunächst die in der Datenbank abzulegenden Daten abgegrenzt: Verwaltungs- und Metadaten werden in der Datenbank abgelegt und die Archivalien selbst der durch die Hardwareeinbindung verwendeten Archivablage im lokalen Dateisystem überantwortet.
In drei Verzeichnissen, dem Daten-, dem Operations- und dem Ereignisverzeichnis wurden die Fakten über die Daten, die im System möglichen Operationen mit den Daten sowie die die Operationen auslösenden Ereignisse aufgelistet. Im konzeptuellen Entwurf wurden diese in ein semantisches Schema überführt. Daraus wurde dann das logische Datenmodell erstellt. Hierzu war es nötig zu klären, welche Umsetzungsweise für die Baumstruktur der Kategorien dazu geeignet ist, um diese Daten in einem relationalen Modell abzubilden, ohne dabei zu großen Aufwand bei häufig benötigten Operationen betreiben zu müssen. Mit dem Physischen Entwurf zur Leistungsoptimierung konnte die Modellierung abgeschlossen werden.

Die Konzeption wird durch die detaillierte Beschreibung der der Anwendung innewohnenden Anwendungsfälle sowie der Beziehungen zwischen diesen Anwendungsfällen vervollständigt und durch den als letzten Punkt erfolgreichen Vergleich mit den Anforderungen an das System abgeschlossen.

Kapitel 5

Implementation

5.1 Überblick

In diesem Kapitel wird die prototypische Implementierung des in Kapitel 4 konzipierten Systems beschrieben. Dazu werden zuerst die Wahl der Mittel erklärt, dann die Voraussetzungen aufgezeigt und anschließend die Implementierung selbst vorgestellt und dokumentiert. Hierfür werden die zentralen Programmmodule sowie je eine Beispielimplementierung für die in das System einbindbaren Teile Hardwareeinbindung, Formatübersetzung und Merkmalsextraktion besprochen.

5.2 Wahl der Mittel

5.2.1 Wahl der Programmiersprache

Als Programmiersprache wurde Java gewählt. Java ist eine objektorientierte Sprache, die in Anlehnung an die Programmiersprache C++ entwickelt wurde. Java ist jedoch leichter anwendbar als diese. Eine objektorientierte Programmiersprache zeichnet die damit leicht mögliche Wiederverwendbarkeit von einmal erstellten Objekten aus, welche bei korrekter Anwendung zu einer deutlichen Einsparung von nötigem Programmcode und nicht zuletzt dadurch auch zur Fehlerfreiheit beiträgt.
„Mit Java lassen sich komplette Anwendungen erstellen, die sowohl auf einem einzigen

Computer laufen als auch auf Netzwerk-Server und -Clients verteilt sein können.” (aus Magirus „Guide” [65] S. 86).
Der Hauptvorzug von Java gegenüber anderen Programmiersprachen in Bezug auf die Umsetzung des konzipierten Systems liegt in der Objektserialisierungsfunktionalität, mit der die Speicherung von Objekten und deren Zuständen in Dateien sowie die Übertragung von Objekten in einem Netzwerk auf leicht anzuwendende Weise geschehen kann. Es braucht also nicht – wie dies bei anderen Programmiersprachen nötig ist – eine manuelle Umsetzung der Objektdaten und -zustände für den jeweiligen Speicher- oder Übertragungszweck erstellt werden. Dieses wird bei der Speicherung von Objekten und Zuständen im persistenten Speicher sowie für die Kommunikation zwischen den Systemstandorten ausgenutzt. Ein weiterer Grund für die Wahl von Java liegt in den von Java direkt eingebundenen Message-Digest Funktionen. Diese werden für die Berechnung der Identifikatoren verwendet.

5.2.2 Wahl der Datenbanksoftware

Der Funktionalität der dem konzipierten System zugrunde liegenden Datenbank muss eine genauere Betrachtung zukommen. Es wurde ein relationales Datenbankmodell wegen der langjährigen Erfahrungen auf diesem Gebiet ausgewählt. Als relationale Standardsprache gilt SQL, „die mit dem Ziel konzipiert ist, in allen Bereichen des Betriebs einer relationalen Datenbank umfassende (Sprach-)Unterstützung zu gewährleisten. [...] Die Sprache bietet gleichermaßen Konstrukte zur Definition und Verwaltung von Datenbasisschemata wie auch zur Abfrage oder Änderung von Datenbeständen” (aus Lang/Lockemann: „Datenbankeinsatz”[2] S. 485).
Der unter heutigen Datenbanksystemen am weitesten verbreitete Standard ist SQL92, welcher auch als SQL-2 bezeichnet wird. Nicht jedes Datenbanksystem setzt jedoch diesen Standard vollständig um. Weite Verbreitung haben die Datenbanksysteme MySQL und PostgreSQL erfahren, unter anderem sicherlich durch die Tatsache, dass sie für jedermann frei verfügbar sind. Für beide Datenbanken existiert ein JDBC-Treiber, über den aus Java heraus mit den Datenbanken kommuniziert werden kann.

Käster widmet ihnen in [3] einen Vergleich und Test. Demnach ist in vielen Fällen MySQL die schnellere Datenbank bei verschiedenartigen Anfragen. MySQL verzichtet dafür allerdings auf für die Umsetzung des konzipierten Systems wichtige Eigenschaften bis hin zu nur eingeschränkter Transaktionsfähigkeit nach dem ACID-Prinzip. PostgreSQL bieten dagegen eine sehr vollständige Umsetzung des SQL-Standards, wenn auch mit Einbußen in der Geschwindigkeit gegenüber MySQL. Diese wiegen aber nicht so schwer wie die fehlende vollständige Transaktionsunterstützung bei MySQL, so dass die Entscheidung letztendlich auf PostgreSQL fiel.

5.3 Voraussetzungen

Das Dämonprogramm setzt eine Java Virtual Machine, den JDBC-Treiber, die Datenbank PostgreSQL und das Programm „file" sowie das Vorhandensein von persistentem Speicher und dessen Einbindung in das lokale Dateisystem voraus. Für den Betrieb von mehr als einem Standort muss außerdem die Kommunikation über ein TCP/IP basiertes Netzwerk möglich sein. Für die Beispielimplementation der Formatübersetzung wird zusätzlich Ghostscript mit dem Skript „ps2pdf" benötigt. Diese Voraussetzungen gelten für jeden Standort einzeln, an denen das Programm laufen soll.
Das GUI-Programm setzt neben dem Dämonprogramm und dessen Voraussetzungen die graphische Ausgabemöglichkeit und zur Bedienung ein Zeigegerät und eine Tastatur voraus. Zum Dämonprogramm muss eine TCP/IP-Verbindung möglich sein, es braucht aber kein netzwerkweiter Zugriff ermöglicht werden.
Für das Kompilieren des Programmcodes wird das Java Development Kit (JDK) und das Programm Ant benötigt.
Die Implementation wurde mit folgenden Programmen und Programmversionen entwickelt und getestet:

- Linux Kernel 2.4.20 unpatched
- Debian GNU/Linux 3.0 (Woody, basierend auf glibc 2.3.1-11)
- PostgreSQL 7.2.1-2woody

- Java database (JDBC) driver for PostgreSQL 7.2.1
- Java(TM) 2 Software Development Kit, Standard Edition (build 1.4.2-b28)
- Java HotSpot(TM) Client VM (build 1.4.2-b28, mixed mode)
- ps2pdf aus Ghostscript gs-aladdin 6.50-5
- file 3.37-3.1.woody
- Ant 1.4.1-4

Die Dateien `AbsoluteConstraints.java` und `AbsoluteLayout.java` aus dem Paket `org.netbeans.lib.awtextra` werden dem Programmcode beigelegt. Sie enthalten fremden, unveränderten Quellcode. Dieser steht unter der „Sun Public Licence Version 1.0" und wurde mit dem Programm „netBeans IDE 3.5" mitgeliefert. Dieser Programmcode wird vom GUI-Programm zur Anordnung der graphischen Elemente verwendet.

5.4 Das Dienstprogramm

Der implementierte Prototyp für das konzipierte Archivierungssystem trägt den Namen „VDAS". Dies steht abkürzend für „Verteiltes Datenarchivierungssystem".

Zunächst wird eine Übersicht über die wichtigen Funktionen und Bestandteile und die Vorgänge beim Programmstart gegeben. Die detaillierte Betrachtung der Implementation wird dann mit der Dokumentation der verwendeten Datenklassen für die Meta- und Verwaltungsdaten und die Datenübertragung zwischen den Standorten begonnen. Daran anschließend wird die Umsetzung der zentralen Dienste dargelegt, aus denen das implementierte System besteht.
Die zentralen Dienste werden voneinander sowie von den die Anwendungsfälle repräsentierenden Klassen genutzt. Die Betrachtung wird mit der Beschreibung der Umsetzung der Anwendungsfälle abgerundet.

Die Anwendung basiert auf der Verwendung von nebenläufiger Programmausführung, welche durch die in Java vorhandene Threadverwaltung realisiert wird.
In Anhang A werden ausgewählte Teile aus dem Quellcode dargestellt. Der gesamte Quellcode ist zusammen mit den vorkompilierten Jar-Archiven und diesem Text auf der beigefügten CD-ROM (siehe dazu auch Anhang C) vorhanden und einzusehen.

5.4.1 Überblick über Start, wichtige Funktionen und Bestandteile des Dienstprogramms

Die Hauptfunktion des Dienstprogramms ist das Beantworten von Anfragen mit Hilfe seiner zentralen Dienste. Hierzu wartet die Hauptprogrammschleife auf das Eintreffen von entsprechenden Anfragen und startet zu deren Beantwortung jeweils einen neuen Thread.
Abbildung 5.1 zeigt das Vorgehen beim Programmstart. Im Zuge des Programmstarts werden die Wiederherstellungsmethoden der Dienste, die einer Wiederherstellung bedürfen, vor dem Eintritt in die Hauptprogrammschleife aufgerufen. Diese sind der

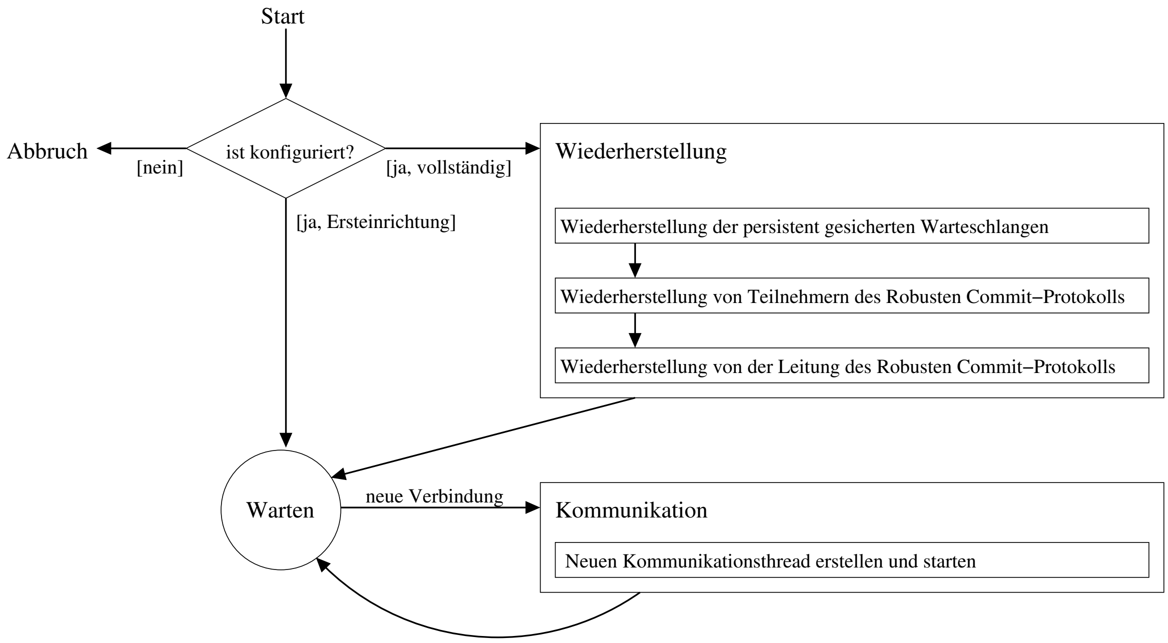

Abbildung 5.1: Vorgänge bei Start und Ablauf des Dienstprogramms

Dienst, der persistent gesicherter Warteschlangen zur Verfügung stellt, sowie die Teilnehmerdienste und der Leitungsdienst des Robusten Commit-Protokolls. Die zentralen Dienste des Dienstprogramms sind

- die Auskunft, welche die Daten aus der Konfigurationsdatei und die des angemeldeten Benutzers bereitstellt,
- der Persistenzdienst mit den Erweiterungen des Logging-Dienstes und der Verwaltung von persistent gesicherten Warteschlangen,
- der Kommunikationsdienst, der alle für die Kommunikation zwischen zwei Dämonen sowie der Kommunikation zwischen Dämon und GUI nötigen Funktionen zur Verfügung stellt,
- der Robuste Commit-Protokoll Dienst sowie
- das Pluginsystem, welches alle für die Hardwareeinbindung, die Formatübersetzungen und Merkmalsextraktionen nötigen Klassen dazu zu laden vermag.

Diese Dienste werden in den folgenden Abschnitten nach der Vorstellung der im System relevanten Daten detailliert beschrieben. Sie bilden zusammen die Funktionalität des Systems auf dem die Behandlung der Anwendungsfälle basiert.

Abbildung 5.2 auf der nächsten Seite zeigt einen Überblick über das implementierte System. Oben sind die die Grundlagen, auf denen das implementierte System aufbaut und darunter das VDAS-System mit seinen zentralen Diensten und den Teilen Dämon und GUI dargestellt. Die Pfeile zeigen an, welche Systemteile aufeinander beruhen. Abbildung 5.3 auf Seite 134 zeigt die Verbindungen und Datenflußrichtungen zwischen Standorten, Dämon und GUI sowie die Verbindungen mit der Datenbank und dem persistenten Speicher. An den Pfeilen ist jeweils die verwendete Technologie verzeichnet.
Dämonen kommunizieren untereinander direkt über das zugrundeliegende Netzwerk. Dazu werden die Funktionen aus java.net genutzt. Eine GUI verbindet sich mit dem lokalen Standort. Diese Verbindung wird durch den gleichen Kommunikationsdienst

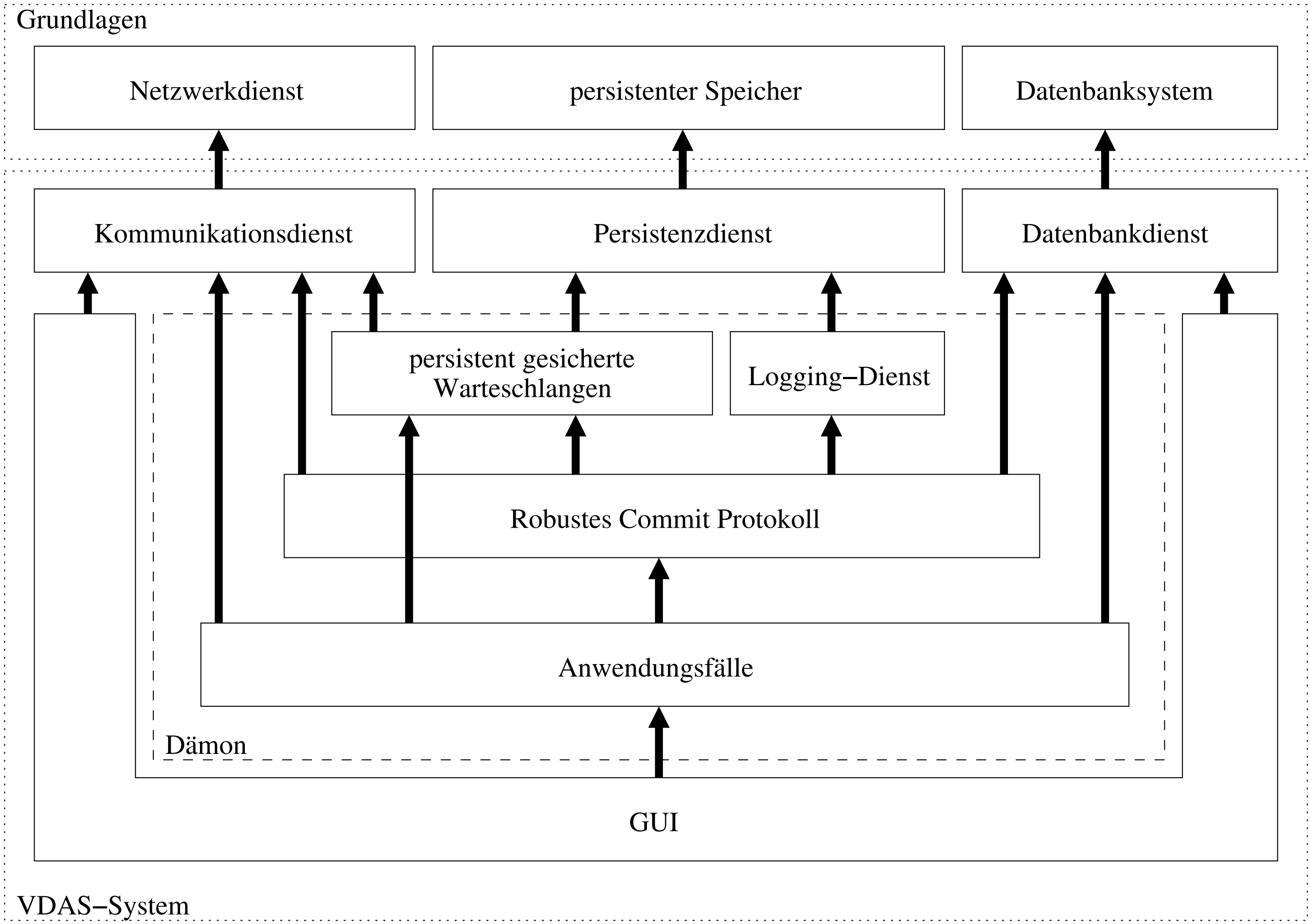

Abbildung 5.2: Übersicht über das implementierte System

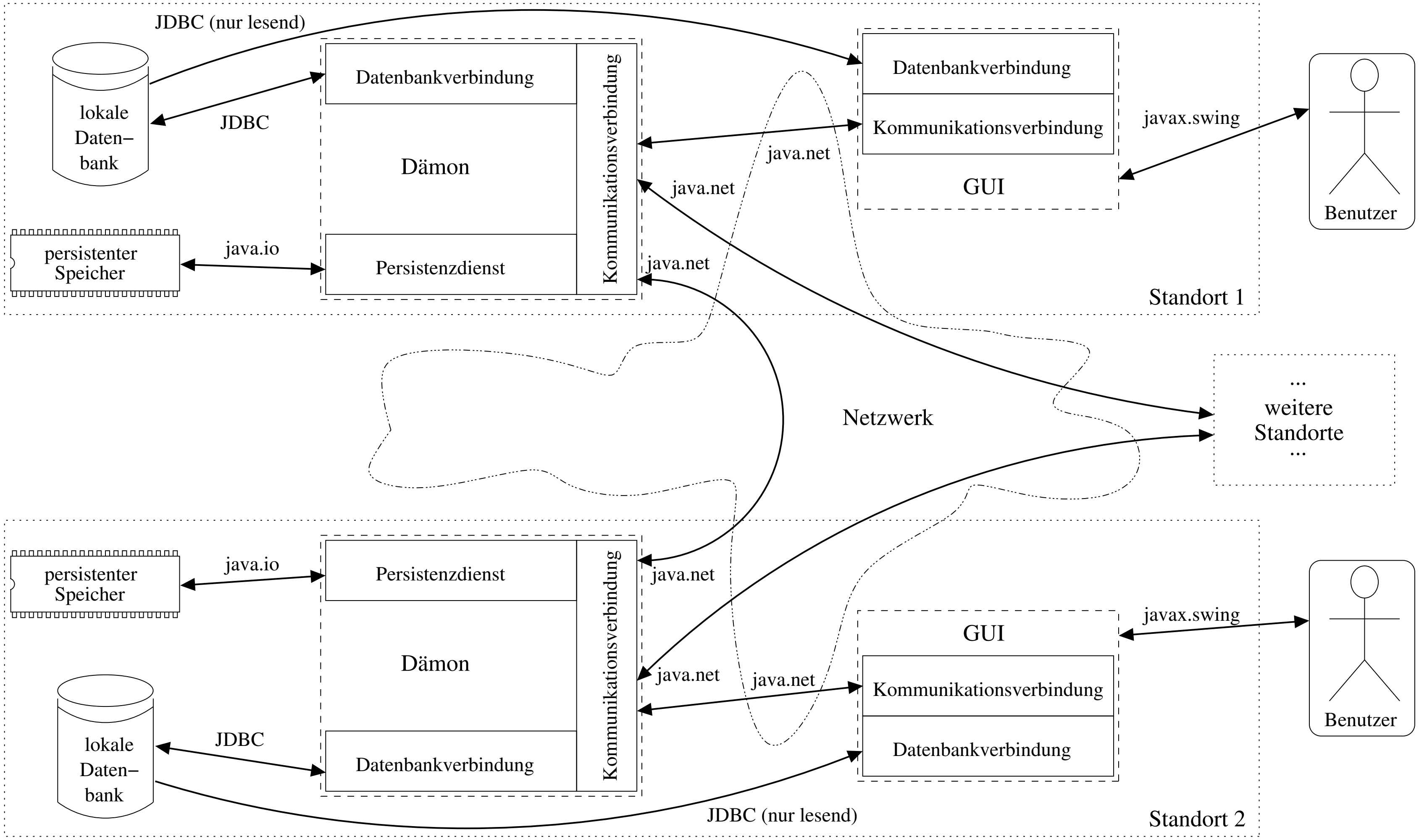

Abbildung 5.3: Übersicht über die Verbindungen im implementierten System

hergestellt, der auch für die Verbindung zwischen den Dämonen sorgt. GUI und Dämon eines Standorts nutzen die Funktionen der Datenbank mit Hilfe des JDBC-Treibers, wobei nur der Dämon auch Schreibzugriffe durchführt. Der persistente Speicher wird nur vom lokalen Dämon verwendet. Dazu werden die Funktionen aus java.io benutzt. Dargestellt sind auch zwei Benutzer. Sie können das System durch die GUI mittels einer in javax.swing realisierten Oberfläche bedienen.

5.4.2 Die Daten

Daten aller Art werden vom implementierten Prototyp nach Möglichkeit erst direkt vor bzw. bei der Verwendung aus der Datenbank gelesen. Zur Herstellung der Datenbankverbindung dient die Klasse `DatenbankGrundlagen` (siehe auch Anhang A.2.1). Daten, welche verändert werden dürfen, werden auf eine bereits durch die Konzeption vorgegebene sichere Weise zurückgeschrieben.
Die in der Datenbank vorkommenden Daten lassen sich gliedern in mit den Archivdaten verknüpfte Daten, Daten zur Kategorisierung, Format- und Funktionsdaten, die für die Rechteverwaltung benötigten Daten und die die Standorte beschreibenden Verwaltungsdaten.
All diese Daten müssen im Falle der Standorterweiterung direkt vom die Datenquelle bildenden Standort zum die Datensenke darstellenden, neuen Standort zu übertragen sein. Die diesem Vorgang gemeinsame Funktionalität wird in der Klasse `Data` implementiert, welche die Oberklasse aller zu übertragenden Daten ist (siehe dazu auch Anhang A.2.3). Sie bietet mit den Methoden

`protected static synchronized Set retrieveAll(Data dummy)` und
`protected static synchronized boolean feedAll(Data dummy, Set data)`

die Funktionalität zum auslesen und speichern aller Daten eines Datentyps. Die beiden Methoden überreichte Instanz einer Unterklasse `dummy` dient dabei zur Ausgestaltung

des Lese- bzw. Schreibvorgangs. Die gelesenen Daten selbst werden in der zurückgelieferten bzw. mit übergebenen Menge gehalten.
Die erwähnte Ausgestaltung wird durch die Unterklassen erreicht, indem diese die folgenden abstrakten Methoden geeignet implementieren:

`protected abstract String getDataBezeichnung()`,

`protected abstract String getDataSelectAll()`,

`protected abstract String getDataInsert()` und

`protected abstract Data getDataData(ResultSet r)`
`throws SQLException`

`getDataBezeichnung` muss dabei die Datenbezeichnung der gespeicherten Daten zurückgeben, `getDataSelectAll` die alle zu lesenden Daten auswählende SQL-`select`-Anfrage. `getDataInsert` muss zu dem in der Unterklasse einzelnen gespeicherten Datensatz eine geeignete SQL-`insert`-Anweisung generieren. Die Methode `getDataData` schließlich muss den vom überreichten `ResultSet` aktuell ausgewählten Datensatz in einer Instanz einer von `Data` abgeleiteten Klasse speichern und diese zurückliefern. Auf diese Weise werden nur die nötigen Daten und Methoden in den Unterklassen selbst implementiert, alle Hauptaufgaben in der Oberklasse, so dass beim etwaigen Auffinden eines Fehlers in der Implementation der Quellcode nur an einer und nicht an vielen Stellen korrigiert werden braucht.

Daten, Identifikatoren und Metadaten

Die Archivalien selbst werden über ihren Identifikator verwaltet und mit Hilfe eines an jedem Standort vorhandenen und dort fest konfigurierten Hardwareeinbindungsplugins gespeichert. Diese Identifikatoren werden mit einer in der Konzeption entwickelten Methode aus den Daten der Archivalien selbst mittels einer Message-Digest-Funktion und der Standortnummer erzeugt.
Die Message-Digest-Funktion kann bei der Standorteinrichtung gewählt werden und muss für alle Standorte gleich lauten. Geeignet sind die Funktionen MD5 und SHA-1. Sie werden durch die von Java bereitgestellte Klasse `MessageDigest` implementiert.

Diese liefern die Berechnungsergebnisse in einem `byte`-Array zurück.
Da die Identifikatoren auch zur direkten Speicherung im Dateinamen im Dateisystem herangezogen werden, dort aber nicht alle mit einem Byte möglichen Zeichen vorkommen dürfen, müssen die Ergebnisse der Message-Digest-Berechnung vor der Verwendung noch umgeformt werden. Dies geschieht mittels einer Base64-Kodierung. Das ist eine Methode, um einen Bytestrom aus Bytes mit 8 Bit, also 256 möglichen Zuständen in einen Zeichenstrom basierend auf 64 möglichen Zeichen eindeutig abzubilden. Das Ergebnis wird in einem `String` abgelegt (siehe hierfür auch Anhang A.2.2).
Wie in der Konzeption im Abschnitt 4.3 beschrieben, muss ein Identifikator neben dem Ergebnis der Message-Digest-Berechnung auch die Standortnummer enthalten. Dies wird direkt durch die in Java mögliche implizite Typumwandlung des `int`-Wertes der Standortnummer in einen `String` und die Konkatenation von Message-Digest-Ergebnis mit diesem umgesetzt. Ein solcher Identifikator sieht dann beispielsweise so aus (die Eins am Ende ist dabei die Standortnummer):

```
XOMvi13jgtHBZXD+olZZnsz_Bc0=1
```

Diese Repräsentation wird so auch in der Datenbank gespeichert.
Dort bildet die Tabelle `DATEN` die zentrale Referenzstelle für im System vorhandene Archivalien. Diese Tabelle wird bei der Standorteinrichtung mit der SQL-Anweisung

```
create table DATEN (
primary key (ID), ID character(32),
REPERTORIENPARAMETER text);
```

erstellt. Neben dem im Feld `ID` gespeicherten Identifikator werden dort die Repertorieneinträge der jeweiligen lokalen Hardwareeinbindung abgelegt.
Die Daten werden durch die Metadaten gekennzeichnet und beschrieben. Die Metadaten wiederum werden in einer Tabelle abgelegt, welche bei der Standorteinrichtung mit der SQL-Anweisung

```
create table METADATEN (
primary key (ID, MBEZ), foreign key (ID) references DATEN (ID),
ID character(32), MBEZ character varying(32), WERT text);
```

eingerichtet wird. `MBEZ` speichert dabei die Metadatenbezeichnung, `ID` referenziert den Identifikator in der Datentabelle und `WERT` nimmt den zugeordneten Wert auf.
Alle Kriterienbezeichnungen und -werte müssen somit in Zeichenform vorliegen. Dieses muss vom Eingebenden sichergestellt sein. Es stellt eine Einschränkung zu den Vorstellungen aus den Anforderungen dar, welche beliebige Wertetypen vorsehen. Jedoch lassen sich beliebige Wertetypen in eine sie repräsentierende Textform umwandeln. Diese Einschränkung ist in der Praxis für das Funktionieren des Systems somit nicht von Bedeutung, schränkt aber die Möglichkeiten eines Benutzers derartig ein, dass dieser selbst für eine geeignete Umwandlung vor der Eingabe in das implementierte System Sorge zu tragen hat. Diese hätte für die Speicherung in einem Computersystem vermutlich ohnehin durchgeführt werden müssen. Alle in Computersystemen vorliegenden Datentypen lassen sich automatisiert in die benötigte Form umwandeln. Eine umfassendere Implementation könnte hierzu eine weitere Pluginmöglichkeit schaffen. In dieser prototypischen Implementation wurde davon allerdings abgerückt, denn die Hauptaspekte dieser Arbeit werden davon nicht beeinträchtigt.

Datengruppen und Kategorien

Datengruppen und Kategorien stellen die ordnungsgebenden Mittel für den Benutzer dar. Ihre Verwendung und Verwendbarkeit wurde ausführlich in der Konzeption besprochen.
Datengruppen werden in einer Tabelle gespeichert, die bei der Standorteinrichtung mit der SQL-Anweisung

```
create table DATENGRUPPEN (
primary key (GBEZ), GBEZ character varying(32));
```

errichtet wird, wobei `GBEZ` die Datengruppenbezeichnung aufnimmt. Für die Kategorien lautet die Anweisung

```
create table KATEGORIEN (
primary key (KBEZ), KBEZ character varying (32),
KATEGORIE text, L int not null, R int not null);.
```

Da auch längere als 32 Zeichen lange Kategorienbezeichnungen aufgenommen werden sollen, die Referenzen darauf aber ebensoviel Speicher verbrauchen würden, wird beim Einfügen einer neuen Kategorie aus der Kategorienbezeichnung eine Kurzbezeichnung berechnet. Hierzu dient die gleiche Message-Digest-Funktion, die bereits bei der Erstellung der Datenidentifikatoren beteiligt war. Die komplette Bezeichnung wird im Feld `KATEGORIE`, die Kurzbezeichnung im Feld `KBEZ` abgelegt. Mit den Feldern `L` und `R` wird, wie in der Konzeption beschrieben, die Position der Kategorie im Kategorienbaum wiedergespiegelt.

Da alle Datengruppen und Kategorien dem Benutzer beim Einfügen neuer Daten zur Auswahl präsentiert werden sollen, können mit zwei Methoden die Bezeichnungsdaten aller Datengruppen bzw. Kategorien aus der Datenbank gelesen werden. Bei Datengruppen handelt es sich im die Methode

```
public static Vector getList()
```

aus der Klasse `Gruppen`. Bei den Kategorien wird aus den Daten in der Datenbank die Baumstruktur wiederhergestellt und diese mit der Methode

```
public static DefaultMutableTreeNode getTree()
```

der Klasse `Kategorien` ausgeliefert.

Formate und Funktionsdaten

Formate werden im implementierten System automatisch erkannt. Dies ist aber kein Verdienst des Systems selbst, sondern fußt auf den Ausgaben des Programms „file". Dieses benutzt ein ausgeklügeltes Verfahren, um aus einem Dateiinhalt auf das genaue, in der Datei enthaltene Datenformat zu schließen. Die Auswertung kann mit Hilfe der in der Klasse `Formate` enthaltenen Methode

```
public static String getFormat(String filename)
```

durchgeführt werden. Damit diese Vorgehensweise funktioniert, muss an allen Standorten sichergestellt werden, dass „file" überall die gleiche Entscheidungsbasis benutzt. Da die Ausgaben von „file" sehr lang werden können, wird für die Speicherung und Referenzierung in der Datenbank eine Kurzbezeichnung generiert und zwar nach dem bereits bei der Betrachtung der Kategorien beschriebenen Verfahren der Message-Digest-Berechnung. Diese Kurzbezeichnung wird im Feld `FBEZ` zusammen mit der kompletten Ausgabe von „file" im Feld `FORMAT` abgelegt. Dies geschieht in der Tabelle `FORMATE`, die bei der Standorteinrichtung mit der SQL-Anweisung

```
create table FORMATE (
primary key (FBEZ), FBEZ character varying(32),
FORMAT text);
```

eingerichtet wird.
Auf das Feld `FBEZ` dieser Tabelle wird aus der Tabelle `FORMATUEBERSETZUNGEN` und der Tabelle `MERKMALSEXTRAKTIONEN` verwiesen. Diese werden bei der Standorteinrichtung mit folgenden SQL-Anweisungen erstellt:

```
create table FORMATUEBERSETZUNGEN (
primary key (QUELLFORMAT, ZIELFORMAT),
foreign key (QUELLFORMAT) references FORMATE (FBEZ)
on update cascade on delete cascade,
foreign key (ZIELFORMAT) references FORMATE (FBEZ)
on update cascade on delete cascade,
QUELLFORMAT character varying(32),
ZIELFORMAT character varying(32), FUNKTIONSNAME text);
```

```
create table MERKMALSEXTRAKTIONEN (
primary key (FORMAT),
foreign key (FORMAT) references FORMATE (FBEZ)
on update cascade on delete cascade,
FORMAT character varying(32), FUNKTIONSNAME text);
```

Die Verwendung und der Aufbau des bei beiden Tabellen vorhandenen Feldes `FUNKTIONSNAME` wurde bereits beim Pluginsystem in Abschnitt 5.4.7 beschrieben.

Die Rechteverwaltung

Bei der Rechteverwaltung spielen die Daten zu Benutzern, Benutzergruppen, der Zugehörigkeit von Benutzern zu Benutzergruppen sowie die Ein-, Such- und Auslieferrechte von Benutzergruppen eine Rolle.
Benutzerdaten werden in einer Tabelle gespeichert, die mit der SQL-Anweisung

```
create table BENUTZER (
primary key (BBEZ), BBEZ character varying(32),
NAME text, PASSWORT character varying(32) not null);
```

bei der Standorteinrichtung erstellt wird. Für die Tabelle der Benutzergruppen sieht die SQL-Anweisung so aus:

```
create table BENUTZERGRUPPEN (
primary key (BGBEZ), BGBEZ character varying(32),
STATUS int);
```

Die Benutzerzugehörigkeit wird in einer Tabelle verzeichnet, die mit der folgenden SQL-Anweisung eingerichtet wird:

```
create table BENUTZERGRUPPENZUGEHOERIGKEIT (
primary key (BGBEZ, BBEZ),
foreign key (BGBEZ) references BENUTZERGRUPPEN (BGBEZ)
on update cascade on delete cascade,
foreign key (BBEZ) references BENUTZER (BBEZ)
on update cascade on delete cascade,
BGBEZ character varying(32), BBEZ character varying(32));
```

Die jeweiligen Bezeichnungsfelder `BBEZ` für den Benutzer und `BGBEZ` für die Benutzergruppen korrespondieren dabei mit den gleichnamigen Feldern in der Benutzerzugehörigkeitstabelle. Diese Gleichbezeichnung läßt sich für eine leichte Erfassung durch einen Natural Join (siehe hierzu auch den Abschnitt 5.4.8 zur Datensuche) in einer SQL-Datenabfrage nutzen.
Die Felder `NAME` und `PASSWORT` der Benutzertabelle sind selbsterklärend. Das Feld `STATUS` der Benutzergruppentabelle dagegen bedarf noch einer genaueren Betrachtung:
Die in dem Statusfeld enthaltenen Bits haben eigenständige Bedeutungen und werden in der Klasse `Benutzergruppen` zur Laufzeitverwendung in eine Statusmenge expandiert, mit der programmiertechnisch leichter im objektorientierten Sinne weitergearbeitet werden kann. Enthalten sind drei der jeweiligen Benutzergruppe zugeordnete Rechte:

- *Administrationsrecht*: Dieses Recht markiert die Benutzergruppe und damit gleichzeitig die der Gruppe zugeordneten Benutzer als zur Administration zugehörig. Ein Benutzer, der zu einer derartig markierten Gruppe gehört, wird vom Programm als Administrator angesehen.

- *Standardauslieferrecht*: Dieses berechtigt die Gruppenmitglieder dazu, alle Daten, bei denen nicht explizit das Auslieferrecht verneint wurde, ausliefern zu dürfen.

- *Standardsuchrecht*: Dieses berechtigt die Gruppenmitglieder dazu, alle Daten, bei denen nicht explizit das Suchrecht verneint wurde, bei der Datensuche auffinden zu dürfen.

Die letzten beiden Rechte können der Administration die Arbeit erleichtern, da so nicht für jede Daten-Benutzergruppen-Kombination ein Recht explizit angegeben werden muss.

Für die detaillierte Einstellung der in den Anforderungen ausgewiesenen Rechte gibt es zwei Rechtetabellen: `EINLIEFERUNGSRECHTE` und `AUSLIEFERUNGSRECHTE`. Such- und Auslieferrechte werden aufgrund ihres gleichartigen Charakters in nur einer Tabelle untergebracht. Die beiden Tabellen werden im Zuge der Standorteinrichtung mit folgenden SQL-Anweisungen erstellt:

```
create table EINLIEFERUNGSRECHTE (
primary key (BGBEZ),
foreign key (BGBEZ) references BENUTZERGRUPPEN (BGBEZ)
on update cascade on delete cascade,
BGBEZ character varying(32), EINLIEFERRECHT boolean);


create table AUSLIEFERUNGSRECHTE (
primary key (ID, BGBEZ),
foreign key (ID) references DATEN (ID)
foreign key (BGBEZ) references BENUTZERGRUPPEN (BGBEZ)
on update cascade on delete cascade,
ID character(32), BGBEZ character varying(32),
AUSLIEFERRECHT boolean, SUCHRECHT boolean);
```

Die Auswertung der Rechte geschieht direkt in den jeweiligen Anwendungsfällen und wird in Abschnitt 5.4.8 beschrieben.

Die Standort- und Verfügbarkeitsdaten

Die Daten zu den verschiedenen Standorten, auch dem lokalen Standort, werden in der Tabellen `STANDORTINFORMATION` geführt. Sie wird bei der Standorteinrichtung mit folgender SQL-Anweisung erstellt. Im Zuge der Standorteinrichtung erfolgt auch der erste Eintrag in diese Tabelle:

```
create table STANDORTINFORMATION (
primary key (STANDORTID),
STANDORTID int, ADRESSE inet, PORT integer);
```

Im Zuge der Standorterweiterung werden die Daten des jeweils neuen Standorts an alle bestehenden Standorte übermittelt. Jene Standorte nehmen die Daten in ihren Datenbestand auf. Das Feld `STANDORTID` wird außerdem durch die die Datenverfügbarkeiten speichernde Tabelle `DATENSTANDORTE` referenziert. Jene Tabelle wird bei der Standorteinrichtung mit der SQL-Anweisung

```
create table DATENSTANDORTE (
primary key (ID, STANDORTID),
ID character(32) references DATEN (ID),
STANDORTID integer references STANDORTINFORMATION (STANDORTID));
```

eingerichtet. Diese Tabelle wird nach und nach mit Daten gefüllt, nämlich immer dann, wenn neue Daten eingefügt werden oder Daten an einen fernen Standort ausgeliefert und im Zuge dieses Vorgangs auch an diesem gespeichert werden.

5.4.3 Die Konfigurationsdatei und die zentrale Auskunft

Das Programm wird über eine Konfigurationsdatei eingerichtet. Diese dient zur Einstellung grundlegender Dienste, wie der Verbindung zur Datenbank. Die Eintragungen haben die Form

```
Schlüssel="Wert".
```

Zeilen, die mit dem Zeichen `#` beginnen und auch leere Zeilen gelten als Kommentare und werden beim Einlesen der Konfigurationsdatei ignoriert.
Beim Programmstart wird nach einer `vdas.conf` benannten Datei im Unterverzeichnis `conf` gesucht. Diese beinhaltet bei einem eingerichteten System die grundlegenden Konfigurationsdaten. Fehlt sie, so wird versucht eine `vdas.default.conf` benannte Datei im selben Unterverzeichnis zu lesen. Diese muss vor der Ersteinrichtung des Standorts

mit mindestens dem Schlüssel-Wert-Paar für den Port gefüllt werden, an dem der Dämon des Standorts auf eine Kommunikationsaufnahme warten soll. Der Schlüssel hierzu lautet `daemon.port`. Fehlt auch diese Konfigurationsdatei, so wird der Programmstart abgebrochen.

Die eingelesenen Konfigurationsdaten werden durch die Klasse `Configuration` bereitgestellt. Dieser Dienst ist über eine zentrale Auskunftsinstanz erreichbar. Diese Auskunft ist als Singleton-Objekt[1] ausgeführt. Neben dem Konfigurationsdatenzugriff werden über das Auskunftsobjekt auch die Daten zum angemeldeten Benutzer sowie der zentrale Zugriff auf die Standortdaten bereitgestellt. Durch die Singleton-Implementierung wird so sichergestellt, dass nicht mehr als ein Benutzer zugleich am lokalen System angemeldet werden kann und dass diese grundlegenden Dienste zur Speicherersparnis nur einmal angelegt und eingerichtet werden.

5.4.4 Der Persistenzdienst

Der Persistenzdienst steht als Mittlerschicht den anderen Programmteilen für den geordneten Zugriff auf den persistenten Speicher zur Verfügung. Er bietet mit den Methoden

```
public static synchronized void writeSingle
(String filename, Serializable object, boolean append)
```
und
```
public static synchronized Serializable readSingle
(String filename)
```

einen direkten Zugriff auf Dateien im persistenten Speicher und kümmert sich im Falle der Methode `writeSingle` um das leeren sämtlicher erreichbarer Pufferspeicher, bevor die Programmausführung fortgesetzt wird. Die Einschränkung auf „erreichbare Pufferspeicher" bedeutet, dass nur Java-interne Puffer unter der Kontrolle des Programms stehen. Deshalb muss bereits vor der Einrichtung des Programms sichergestellt werden,

[1] Durch das Singleton-Entwurfsmuster wird in der objektorientierten Programmierung sichergestellt, dass genau eine Instanz der Objektklasse pro Programminstanz ermöglicht wird, also Objekte dieser Klasse niemals doppelt vorkommen.

dass der genutzte Speicher durch das Betriebssystem direkt angesprochen wird und eine Hardware sowie ein Dateisystem zum Einsatz kommt, bei der bzw. dem Caching-Funktionen ausgeschaltet sind. Welche Art von Speicher hierfür genutzt wird, liegt in der Entscheidungshoheit der Administration.

Der Logging-Dienst

Der Logging-Dienst baut auf dem Persistenzdienst direkt auf. Er bietet die Methoden

`public static synchronized void log`
`(String filename, Serializable object)` und
`public static synchronized List readLog`
`(String filename)`

zum Schreiben eines Logeintrags bzw. zum Lesen aller in der Datei enthaltenen Logeinträge an. Darüber hinaus bietet der Logging-Dienst Methoden zur Verwaltung der Dateinamen mit laufenden Nummern an, so dass die diesen Dienst verwendenden Programmteile nur noch ein sie eindeutig ausweisendes Präfix benutzen müssen und damit von weiteren Verwaltungsvorgängen befreit sind.

Die persistent gesicherten Warteschlangen

Eine persistent gesicherte Warteschlange ist zunächst eine klassische nach dem First-In-First-Out Prinzip (FIFO Prinzip) arbeitende Schlange mit den Methoden

`public synchronized Serializable enqueue`
`(Serializable element)`,
`public synchronized Serializable top()`,
`public synchronized Serializable dequeue()`,
`public synchronized boolean isEmpty()` und
`public synchronized boolean isFull()`.

Die interne Verwaltung geschieht mit Hilfe zweier Variablen, welche als Zeiger auf die Kopf- und die Schwanzposition dienen. Letztere zeigt dabei immer auf die nächste freie

Speicherposition, also direkt hinter den zuletzt hinzugekommenen Eintrag (siehe hierzu auch Anhang A.3) .

Die Daten selbst werden über den Persistenzdienst in Dateien gespeichert, die mit der passenden Positionsnummer benannt sind. Beim Einfügen eines Elements in die Schlange mit `enqueue` wird dieses Element an der Schwanzposition gespeichert und die die Schwanzposition zeigende Variable um 1 erhöht. Es darf nicht eingefügt werden, falls die Schlange voll ist. Dies kann mit der Methode `isFull` erfragt werden.

Bei der Entnahme eines Elements mit `dequeue` wird das Element aus der der Kopfposition zugeordneten Datei gelesen, diese Datei gelöscht und die die Kopfpoisition anzeigende Variable um 1 erhöht. Falls einer der Zeiger eine Position erreicht, die der Maximalgröße der Schlange entspricht, wird dieser wieder auf 0 zurückgesetzt. Abbildung 5.4 verdeutlicht diese Vorgehensweise. Vom implementierten Programm wird

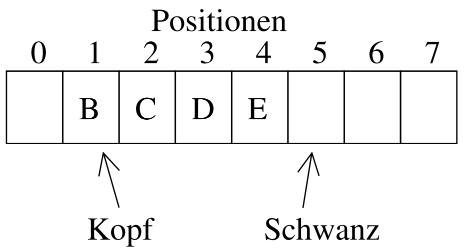

Abbildung 5.4: Eine Schlange mit vier Elementen und Kopf- und Schwanzposition

für jeden Standort eine separate persistent gesicherte Warteschlange mit einem Präfix angelegt, welches aus der Standortnummer und der Kennzeichnung als Warteschlange besteht.

Wiederherstellung

Beim Programmstart wird für jede Warteschlange die Wiederherstellungsprozedur durchgeführt. Die Wiederherstellung des Zustands einer Warteschlange beschränkt sich auf das Bestimmen der Variablenwerte, die die Kopf- und der Schwanzposition beim Systemausfall innehatten. Hierzu bedient sich die Wiederherstellungsprozedur der in den Dateinamen enthaltenen Positionsnummern. In der ansonsten fortlaufenden Nummerierung befindet sich an einer Stelle eine Lücke. Die wiederherzustellende Kopfposition ist das Element am Ende der Lücke, die wiederherzustellende Schwanzposition liegt am Anfang. Es müssen vier Fälle der möglichen Lage der Lücke betrachtet werden:

1. *Überlappung*: Die Nummerierung bricht an der Maximalposition um und geht über 0 voraus. Die Schwanzposition ist beim Durchlaufen der Nummern von klein nach groß bei Beginn der Lücke erreicht, die Kopfposition dann, wenn die fortlaufende Nummerierung nach der Lücke weitergeht.

2. *Startet bei* 0: Die Kopfposition lautet 0, die Schwanzposition ist am Ende der fortlaufenden Nummerierung erreicht.

3. *Endet an Maximalposition*: Die Schwanzposition ist 0 (und war zuletzt gerade umgebrochen worden), die Kopfposition ist bei einem Durchlauf der Nummern von klein nach groß beim Wiedereinsetzen der fortlaufenden Nummerierung erreicht.

4. *Normalfall*: Beim Durchlaufen der möglichen Nummern von klein nach groß wird beim Einsetzen der fortlaufenden Nummerierung die Kopfposition und beim Ende der fortlaufenden Nummerierung die Schwanzposition erreicht.

Im Spezialfall, dass keine Nummern vorliegen, war die Warteschlange leer. Die letzten Kopf- und Schwanznummern können so zwar nicht wiederhergestellt werden, dies ist allerdings auch nicht nötig, da die Schlange leer war und durch Nullsetzen der beiden Variablen wieder neu initialisiert werden kann. Die Aufgabe, eine feste Reihenfolge einzuhalten, wird dadurch nicht gestört.

5.4.5 Die Kommunikationsverbindung

Der Start der Kommunikation wird im implementierten System immer als Anfrage verstanden, welche von einem Client an einen Serverdienst gestellt wird (siehe auch Anhang A.4). Die Kommunikation selbst wird dabei mit einem festen Kommunikationsprotokoll durchgeführt .

Das Kommunikationsprotokoll

Das Kommunikationsprotokoll, `VDASProtokoll` genannt (siehe auch Anhang A.4.1), umfasst den Rahmen, in dem die Kommunikation stattfindet. Es stellt die Methoden

`public abstract Serializable executeLocal()`,

`public abstract Serializable execute()` und

`public boolean verbindungSchliessen()`

zur Verfügung. Beim Eintreffen einer Anfrage wird die `execute` Methode auf der Serverseite ausgeführt und das berechnete Ergebnis als Beantwortung der Anfrage zurückübermittelt. Dort wird – falls nicht inzwischen das Ende der Kommunikation signalisiert wird – die `executeLocal` Methode ausgeführt, um eine weitere Anfrage zu generieren. Das Ende der Kommunikation kann von jeder Seite durch die Methode `verbindungSchliessen` signalisiert werden. Abbildung 5.5 zeigt den beispielhaften Ablauf einer Kommunikation. Eine das Protokoll implementierende Klasse führt die für die Kom-

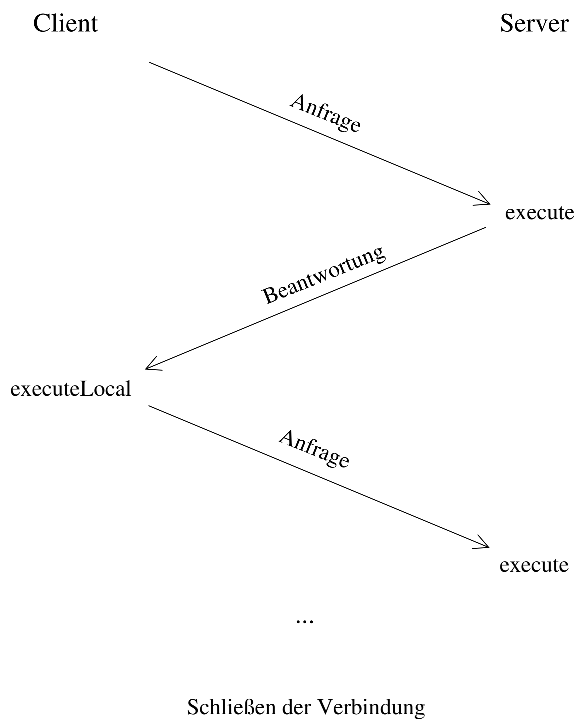

Abbildung 5.5: Beispielhafter Ablauf einer Kommunikation

munikation nötigen Daten mit sich und implementiert die abstrakten Methoden `execute` und `executeLocal` so, dass die gewünschten Operationen auf der jeweiligen Seite durchgeführt werden.

Die Serverseite

Nachdem beim Programmstart des Dämonprogramms die Wiederherstellung aller persistent gesicherten Warteschlangen und Commit-Protokoll-Teilnehmer und -Manager abgeschlossen wurde, öffnet der Dämon den konfigurierten Port und wartet in einer Endlosschleife auf eingehende Anfragen. Bei Eintreffen einer Anfrage wird ein neuer Thread zu deren Beantwortung erstellt. Handelt es sich bei einer Anfrage nicht um eine dem Kommunikationsprotokoll entsprechende Anfrage, so wird sie verworfen, andernfalls wird die `execute` Methode ausgeführt und das Ergebnis über die Verbindung zurückgesendet (siehe hierzu auch Anhang A.4.2).

Die Clientseite

Auf der Clientseite wird zur Durchführung der Anfragen ein Sendedienst zur Verfügung gestellt. Dieser beherrscht sowohl den direkten wie auch den Versand unter Verwendung der jeweiligen persistent gesicherten Warteschlange für den Zielstandort (siehe auch Anhang A.4.3). Zum direkten Versenden werden die Methoden

```
public static VDASProtocol send(Standort to, VDASProtocol msg)
```

und

```
public static Integer send(Standort to, VDASProtocol msg,
Consignee consignee)
```

angeboten. Zum Versenden mit der Absicherung über die persistent gesicherten Wartschlangen dient die Methode

```
public static boolean enqueue(Standort to, VDASProtocol msg).
```

Die erste Methode blockiert die Threadausführung, bis entweder die Antwort von der Serverseite empfangen oder eine angemessene Wartezeit überschritten wird. Als Rückgabe dient entweder die Antwort selbst oder der Wert `null` bei einer Zeitüberschreitung.

Die zweite Methode blockiert die Ausführung des Threads nicht und gibt eine eindeutige Wartenummer zurück, mit der der angegebene Antwortempfänger `consignee` die Antwort der Anfrage wieder zuordnen kann. Diese Möglichkeit wird beispielsweise in der Umsetzung des Robusten Commit-Protokolls genutzt, um die Stimmen der Commit-Teilnehmer einzusammeln.

Wird letztere Methode verwendet, so wird die Programmausführung sofort fortgesetzt, sobald die zu versendende Nachricht in den persistenten Speicher übertragen und somit gesichert worden ist (siehe hierzu auch Anhang A.4.4). Die Antwort, die die Serverseite auf die Anfrage erstellt, wird als Bestätigung für den Erhalt der Nachricht gewertet und die Nachricht daraufhin aus dem persistenten Speicher entfernt.

5.4.6 Das Robuste Commit-Protokoll

Die Umsetzung des Robusten Commit-Protokolls folgt streng der in Abschnitt 4.2.3 vorgestellten Konzipierung. An jedem Standort existiert eine Instanz der `CommitCoordinator` Klasse, welche die Koordination und Leitung des Protokollablaufs überwacht. Die hiervon geleitete Seite der Teilnehmer wird von der `CommitParticipant` Klasse dargestellt, deren Instanzen bei Bedarf errichtet werden.

In dieser prototypischen Implementation kann also pro Standort maximal ein Robustes Commit zur gleichen Zeit durchgeführt werden. Da aber jeder Anwendungsfall, der die Durchführung eines Robusten Commits beinhaltet, vom Benutzer ausgeführt wird und sich an einem Standort maximal ein Benutzer zur gleichen Zeit anmelden kann, stellt dieses keine Einschränkung der Funktionalität dar. Das Wissen hierüber wird ausgenutzt: Sobald ein weiteres Commit-Protokoll angefangen wird, brauchen die Daten der abgeschlossenen, von diesem Standort geleiteten Transaktionen nicht weiter gespeichert zu werden. Sie werden gelöscht, um den persistenten Speicher nicht unnötig zu füllen.

Die an jedem Standort lokal auszuführende Transaktion wird in einer Transaktionsdatenklasse gekapselt über das Kommunikationsprotokoll versendet. Auf diesem Kommunikationsweg gelangen auch die Stimmen der Teilnehmer zur Koordinationsstelle zurück. Die von der Koordinationsstelle getroffene globale Entscheidung auf „Commit" oder „Abort" wird den Teilnehmern über die persistent gesicherten Warteschlangen zugestellt. Es wird dabei nicht zwischen dem lokalen Teilnehmer und fernen Teilnehmern unterschieden.

Transaktionsdaten

Instanzen der Transaktionsdatenklasse nehmen eine aus der Standortnummer des leitenden Standorts und einer fortlaufenden Nummer bestehende Transaktionsnummer auf. Über diese werden die Transaktionen eindeutig bestimmt. Sie wird für die spätere Zuordnung der globalen Entscheidung zur Transaktion bei jedem Teilnehmer verwendet. Darüber hinaus bietet die Transaktionsdatenklasse für die Verwendung durch den jeweiligen Teilnehmer die Methoden

`public boolean setup()`,
`public abstract boolean execute()`,
`public boolean commit()` und
`public boolean abort()`

an. Mit der `setup` Methode wird die Verbindung zur Datenbank hergestellt und für die Nutzung in der `execute` Methode vorbereitet. Diese wiederum wird von der eigentlichen Unterklasse der Transaktionsdatenklasse mit den bei der Transaktion durchzuführenden Aktionen ausgeführt. Mit den Methoden `commit` bzw. `abort` werden nach Erhalt der globalen Entscheidung die Änderungen entweder festgeschrieben oder verworfen.

Normalbetrieb

Die Durchführung eines Robusten Commits wird mit der Methode

```
public static boolean execute(TransactionData tadata)
```

der `CommitCoordinator` Instanz angestoßen (siehe auch Anhang A.5.1). Falls noch keine solche Instanz existierte, wird sie an dieser Stelle angelegt (nach dem Singleton-Prinzip wie bei der Auskunft in Abschnitt 5.4.3 beschrieben). Als ersten Schritt führt der CommitCoordinator das Entfernen von Daten früherer, bereits abgeschlossener Protokollausführungen durch. Danach werden die Daten aller Teilnehmer gelesen und die Transaktionsdaten, die Daten der Teilnehmer und der Eintrag „`BEGIN`" mit Hilfe des Logging-Dienstes im persistenten Speicher vermerkt. Anschließend werden die Transaktionsdaten an alle Teilnehmer übermittelt und die Stimmen der Teilnehmer eingesammelt. Falls eine Stimme auf „`ABORT`" lauten sollte, so wird die Standortnummer über den Logging-Dienst in den persistenten Speicher eingetragen. Mit dem Eintreffen der letzten Stimme wird die erste Phase des Protokolls abgeschlossen.

Es wird folglich die Entscheidung auf „`GLOBAL COMMIT`" oder „`GLOBAL ABORT`" getroffen und eine Liste der noch zu benachrichtigenden Standorte zusammengestellt. Dies sind diejenigen Standorte, welche nicht mit „`ABORT`" gestimmt haben. Die Entscheidung wird umgehend über den Logging-Dienst im persistenten Speicher vermerkt. Den Standorten auf der Liste der zu benachrichtigenden Standorte wird die Entscheidung durch die dem Standort zugehörige persistent gesicherte Warteschlange zugestellt und mit dem Eintrag „`DONE`" durch den Logging-Dienst die zweite Phase abgeschlossen. Die getroffene Entscheidung wird als Ergebnis an den Aufrufer der `execute` Methode übermittelt.

Auf Teilnehmerseite wird unterdessen beim Eintreffen der Transaktionsdaten eine neue Instanz der `CommitParticipant` Klasse errichtet (siehe auch Anhang A.5.2). Diese beginnt nach dem entfernen überflüssig gewordener persistenter Dateien von von dem in den Transaktionsdaten verzeichneten Standort geleiteten abgeschlossenen Transaktionen sofort mit dem Aufruf der Methoden `setup` und `execute` aus der übermittelten Transaktionsdatenklasse. Das Ergebnis der Ausführung, welches im Fehlerfall „`ABORT`" und sonst „`COMMIT`” lautet, wird dem `CommitCoordinator` als Antwort zurückübermittelt. Vor der Übermittlung werden im Erfolgsfall die Transaktionsdaten sowie die Entscheidung durch den Logging-Dienst im persistenten Speicher festgehalten. Damit ist die erste Phase des Protokollablaufs auf Teilnehmerseite abgeschlossen. Die Instanz wartet folglich auf die Zustellung der globalen Entscheidung. Im Fehlerfall wird auch diese Entscheidung und der Abschluss „`DONE`” durch den Logging-Dienst im persistenten Speicher vermerkt, jedoch findet die zweite Phase nicht statt und die Instanz hält sich für die Beantwortung etwaig doppelter Anfragen bereit.

Dem Übermittlungsobjekt der globalen Entscheidung ist die Transaktionsnummer beigefügt, über welche die zugehörige Instanz des `CommitParticipants` wiedergefunden wird. Die Entscheidung wird sofort mit Hilfe des Logging-Dienstes in den persistenten Speicher eingetragen. Daraufhin wird je nach Entscheidung die `commit`- oder die `abort`-Methode der Transaktionsdaten aufgerufen und der Abschluss mit als „`DONE`” über den Logging-Dienst im persistenten Speicher vermerkt. Erst nach dem Vermerken wird der Erhalt der die globale Entscheidung übermittelnden Anfrage bestätigt. Damit ist die zweite und letzte Phase auf Teilnehmerseite abgeschlossen.

Wiederherstellung

Die Wiederherstellung wird bei Programmstart durchgeführt. Dabei werden zuerst die Zustände der Teilnehmer und danach der Zustand des Koordinators wiederhergestellt, damit letzterer sofort mit der weiteren Protokollausführung fortfahren kann und nicht auf die Wiederherstellung des Zustands des lokalen Teilnehmers warten braucht. Die bei Programmabbruch aktiven Teilnehmer lassen sich eindeutig durch die in den Dateinamen jeweils vorhandene Transaktionsnummer und die Standortnummer des leitenden Dämons der im persistenten Speicher abgelegten Logeinträge bestimmen.
Der wiederhergestellte Zustand richtet sich direkt nach dem letzten in der Logdatei vermerkten Eintrag. Lautet dieser auf „`DONE`”, so war die Bearbeitung bereits abgeschlossen. Ein `CommitParticipant` extrahiert daraufhin die letzte verzeichnete Stimme und

hält diese für die Beantwortung etwaig doppelter Anfragen bereit. Ein `CommitCoordinator` braucht in diesem Fall nichts Weiteres zu tun, denn die globale Entscheidung wurde bereits erfolgreich an die persistenten Warteschlangen übermittelt.

Liegt ein anderer Eintrag als letzter Eintrag vor, so richtet sich der wiederhergestellte Zustand nach den in Abschnitt 4.2.3 dargelegten Überlegungen. Ein `CommitParticipant` führt dazu eventuell erneut die Methoden `setup` und `execute` aus der Transaktionsdatenklasse aus, um den Wartezustand und die Sperren in der Datenbank für diese Transaktion wiederzuerlangen.

Ein `CommitCoordinator` fährt direkt nach dem Wiederherstellen des Zustandes mit der Bearbeitung fort. Dadurch werden unter widrigen Umständen eventuell doppelte Nachrichten versendet, welche von der Teilnehmerseite durch die eindeutige Transaktionsnummer einer laufenden Transaktion zugeordnet und somit erkannt werden. Es wird daher kein Schaden angerichtet und die weitere Ausführung nicht behindert.

5.4.7 Das Pluginsystem

Das Pluginsystem besteht aus mehreren statischen Methoden in der Klasse `PluginLoader` und einer Menge von Plugins, welche von der abstrakten Klasse `Plugin` abgeleitet sein müssen, um dadurch die Zugehörigkeit zum Pluginsystem anzuzeigen.

Abbildung 5.6 zeigt die Ableitungen und Abhängigkeit der Klassen im Pluginsystem. Die drei Unterklassen der Klasse `Plugin` werden weiter unten anschließend an die Dokumentation der Grundfunktionen des Pluginsystems vorgestellt.

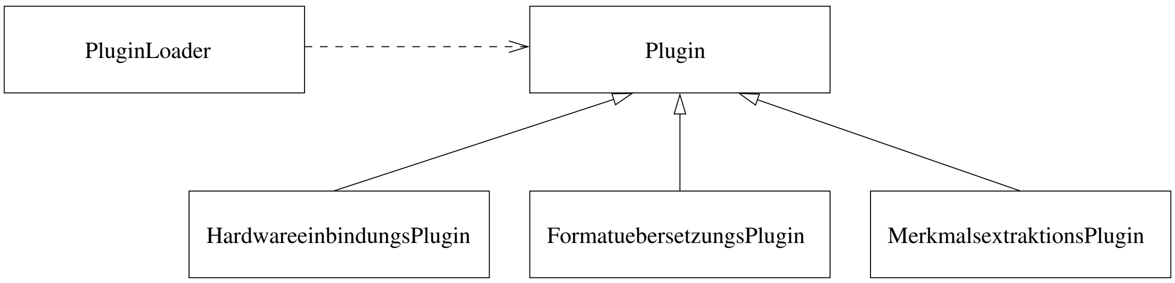

Abbildung 5.6: Ableitungen und Abhängigkeit der Klassen im Pluginsystem

Die Klasse `PluginLoader` stellt zwei Methoden `loadFile` und `saveFile` zur Verfügung, welche zum Auslesen und Speichern einer Plugindatei für die Übertragung von Plugins zwischen den Standorten genutzt werden. Zentraler Punkt des Systems (siehe auch Anhang A.6.1) ist aber die Methode

```
public static Plugin loadPlugin(String pluginName).
```

Sie ermittelt aus dem Inhalt der übergebenen Variable `pluginName` die zu ladende Klasse und den Namen der Jar[2]-Datei, in welcher diese Klasse gespeichert ist. Der Aufbau dieses Strings muss in der Form

```
<Jar-Dateiname>:<vollständiger Klassenpfad und -name>
```

vorliegen. Die Dateiendung `.jar` darf nicht mit angegeben werden. Eine Instanz der genannten Pluginklasse wird von der Methode zurückgegeben, falls das Laden erfolgreich verlief, andernfalls der Wert `null`.

Das Hardwareeinbindungsplugin

Jedes Hardwareeinbindungsplugin muss von der Klasse `HardwareeinbindungsPlugin` abgeleitet werden. Diese bringt die Methoden

```
public String save(String storagename, String filename) und
public boolean load(String repertorienparameter,
String filename)
```

mit. Der `save` Methode wird der Datenidentifikator und der die Datenposition bezeichnende Dateiname übergeben. Aus der so bezeichneten Datei werden die Daten ausgelesen und in die Archivablage übernommen. Die Methode `save` liefert den von der Hardwareeinbindung zum Wiederauffinden benutzten Repertorieneintrag zurück. Die `load` Methode wird mit dem von der `save` Methode beim Einliefern der Daten einst generierten Repertorieneintrag sowie dem die Datenausgabeposition bezeichnenden Dateinamen aufgerufen. Die Hardwareeinbindung muss mit Hilfe des Repertorieneintrags

[2] Jar: Java Archive. Eine Datei, in der (üblicherweise) mehrere vorkompilierte Java-Klassen und bei Bedarf auch Java-Quellcode gespeichert ist.

die Daten wiederfinden und in die angegebene Datei ausgeben.
Abgeleitete Klassen müssen die abstrakten Methoden

`protected abstract String save(String storagename,`
`InputStream in)` und
`protected abstract boolean load(String repertorienparameter,`
`OutputStream out)`

gemäß ihrer Verarbeitungsroutinen geeignet implementieren und sind dabei frei in der Umsetzung.

Das Formatübersetzungsplugin

Jedes Formatübersetzungsplugin muss von der Klasse `FormatuebersetzungsPlugin` abgeleitet werden. Es müssen dabei die Methoden

`public abstract Set quellFormate()`,
`public abstract Set zielFormate()` und
`public abstract boolean translate(String infilename,`
`String outfilename)`

in geeigneter Weise implementiert werden. Mit der Methode `quellFormate` liefert das Plugin alle von ihm als Eingabe akzeptierten Formate aus. Mit der Methode `zielFormate()` geschieht dieses für die sich aus der Übersetzung ergebenen Ausgabeformate. Die Funktion `translate` schließlich führt die eigentliche Übersetzung durch und gibt normalerweise den Wert `true` bei fehlerfreier Ausführung, ansonsten den Wert `false` zurück.

Das Merkmalsextraktionsplugin

Jedes Merkmalsextraktionsplugin muss von der Klasse `MerkmalsextraktionsPlugin` abgeleitet werden. Diese liefert bereits die Methode

`public Map extract(String filename)`

mit, mit deren Hilfe die Kriterien aus der durch den Dateinamen bezeichneten Datei bestimmt werden. Abgeleitete Klassen müssen die Methoden

```
public abstract Set quellFormate() und
public abstract Map extract(InputStream in)
```

auf geeignete Weise implementieren. Mit der Methode `quellFormate` liefert das Plugin alle von ihm als Eingabe akzeptierten Formate aus. Die Methode `extract` führt die eigentliche Merkmalsextraktion aus. Zurückgeliefert wird die Liste aller extrahierten Kriterien bestehend aus Metadatenbezeichnung als Schlüssel und dem extrahierten Metadatenwert als der dem Schlüssel zugeordnete Wert. Konnten keine Kriterien extrahiert werden, so muss eine leere Liste zurückgeliefert werden.

5.4.8 Die Anwendungsfälle

Die Anwendungsfälle wurden strikt nach der jeweiligen Vorgangsbeschreibung der Konzeption ausgeführt. Die graphischen Ausgaben der jeweiligen Eingabemasken und Meldungen und die Verknüpfungen dazwischen sind Sache der Benutzerschnittstelle. Sie werden in Abschnitt 5.5 beschrieben. An dieser Stelle folgt die Beschreibung der Umsetzung der Vorgänge, die hinter den Kulissen der Benutzerschnittstelle ablaufen.

Verarbeitung von AF1 Dateneingabe

In der Klasse `Dateneingabe` wird die Methode

```
public static boolean insert(String filename, Map metadaten)
```

implementiert, die der insert Funktion aus der Konzeption und den Anforderungen direkt entspricht. Ihr wird der Dateiname der und die vom Benutzer eingegebenen Metadaten der einzufügenden Daten übergeben. Die Rückgabe besteht in einem boolschen Wert, der über den Erfolg oder den Fehlschlag der Eingabe informiert. Die gesamte Methode ist in Anhang A.7.1 aufgeführt.

Die Metadatenkriterien werden mit dem Datentyp `Map` gespeichert. Dieser bildet die

Struktur der Metadaten durch Einträge der Form `Bezeichnung=Wert` ab. Vom implementierten System werden Bezeichnungen, die mit zwei Unterstrichen beginnen bei der Eingabe für interne Zwecke verwendet. Sie dürfen nicht vom Benutzer direkt eingegeben werden, können aber dennoch bei der Datensuche ohne Beschränkungen eingesetzt werden. Ohne Beschränkung der Allgemeinheit lassen sich etwaige Benutzerangaben so umwandeln, dass es keine Störung darstellt. Dies wird vom vorliegenden prototypischen System jedoch nicht geleistet.

Die in der Konzeption angesprochene Berechnung der Metadatenabhängigkeiten umfasst in dieser Implementation die Berechnung der Versionsnummer. Ist in den eingegebenen Metadaten der Identifikator der Vorgängerversion durch ein Kriterium mit der Bezeichnung `__vorgänger` verzeichnet, so wird deren Versionsnummer gelesen und um 1 inkrementiert. Andernfalls wird `0` als neue Versionsnummer vergeben. Die Kriteriumsbezeichnung für die Versionsnummer lautet `__version`.

Auf die Berechnung von Metadatenabhängigkeiten folgt das Aufnehmen der Daten durch die Hardwareeinbindung und die lokale Speicherung der Metadaten in der Datenbank. Daraufhin werden alle weiteren Standorte über persistent abgesicherte Warteschlangen über die Eingabe informiert und nehmen die Metadaten sowie die Information, dass die Daten am Eingabestandort verfügbar sind, auf.

Verarbeitung von AF2 Datensuche

Die Datensuche wird mit der Klasse `DatenSuche` implementiert. Sie stellt die Methode

```
public static Set suche(Map kriterien)
```

bereit. Diese Methode verwendet zwei Hilfsmethoden, die aus den übergebenen Kriterien die Suchparameter extrahieren. Von der Methode werden die Suchrechte direkt beachtet. Sie basiert auf verschachtelten mehreren `natural` und `left outer joins`.[3]

[3] Durch einen `natural join` werden Attribute gleicher Bezeichnung auf gleichen Inhalt hin verglichen und die beteiligten Tabellen auf diese Weise automatisch verknüpft. Bei einem `left outer join` wird eine linksassoziative Verknüpfung erstellt, der als Verknüpfungsparameter zusätzlich ein zu einem boolschen Wert auszuwertendender Ausdruck erwartet.

Die vollständige Methode ist mitsamt der beiden Untermethoden in Anhang A.7.2 aufgeführt. Alle übergebenen Suchkriterien werden auf Gleichheit mit den in der Datenbank vorkommenden Metadatenkriterien geprüft und die Einzelvergleiche dabei mit `and` verknüpft. Diese Regel hat eine Ausnahme: Lautet die Suchkriterienbezeichnung auf `__kategorie`, so wird geprüft, ob der zugehörige Suchkriterienwert eine Oberkategorie für die durchsuchten Daten ist. Somit werden alle Daten ins Suchergebnis aufgenommen, die eine Kategorienzuordnung besitzen und zu einer Unterkategorie der gesuchten Kategorie gehören.
Desweiteren wird die Filterung nach den Suchrechten im selben Abfrageschritt durchgeführt, so dass die Suche als sehr häufige Operation mit nur einer SQL-Anfrage erfolgen kann.

Verarbeitung von AF3 Datenauslieferung

Die Datenausgabe wird mit der Klasse `Datenausgabe` implementiert. Die Ausgabe erfolgt durch eine der Methoden

> `public static boolean ausgeben(String id, String filename)` oder
> `public static boolean ausgeben(String id, String filename,`
> `String fbez)`.

Ihnen werden der Datenidentifikator sowie der Ausgabedateiname mitgegeben. Falls keine Formatumwandlung durchgeführt werden soll, wird die erste Methode verwendet, sonst die Zweite. Letzterer wird zudem das gewünschte Ausgabeformat übergeben. Sie verwendet die erste Methode zur Auslieferung an den lokalen Standort und führt anschließend die Formatumwandlung und endgültige Ausgabe in die übergebene Datei durch.
Die erste Methode geht wie folgt vor: Als Erstes wird die Verfügbarkeit der Daten geprüft. Sind sie lokal verfügbar, so wird direkt die Hardwareeinbindung zur Datenausgabe angesprochen und die Daten ausgeliefert. Andernfalls wird die Liste der fernen Standorte, die über die gewünschten Daten verfügen, durchlaufen, bis das Ende der Liste erreicht wurde oder ein Standort zur Auslieferung erreichbar war. Im Falle der

Erreichbarkeit werden die gewünschten Daten übermittelt und in den lokalen Datenbestand durch die Hardwareeinbindung aufgenommen. Diese Verfügbarkeitsänderung wird allen Standorten über persistent gesicherte Warteschlangen mitgeteilt. Anschließend werden die Daten letztendlich an die vom Benutzer gewünschte Position ausgeliefert.
Die Methoden geben als Rückgabewert an, ob die Datenauslieferung erfolgreich war.

Verarbeitung von AF4 Kategorienerweiterung

Da die Kategorienerweiterung direkt auf dem Robusten Commit-Protokoll basiert, existiert zur Kategorienerweiterung eine die Transaktion durchführende Klasse `KategorienTransaction`. Die Kategorienerweiterung wird durch den Aufruf der `execute` Methode des `CommitCoordinator`s mit der die Bezeichnung der Oberkategorie enthaltenen Variable `oberBezeichnung` und der die neue Kategorienbezeichnung enthaltenen Variable `neueBezeichnung` wie folgt durchgeführt (siehe auch Anhang A.7.3):

```
CommitCoordinator.execute(
new KategorienTransaction(neueBezeichnung, oberBezeichnung))
```

Der Rückgabewert ist im Erfolgsfall `true` und ansonsten `false`.

Verarbeitung von AF5 Gruppenvergabe

Die Vergabe einer neuen Datengruppe wird mit der Klasse `Gruppenvergabe` implementiert. Sie stellt eine Methode zur Verfügung, die als Parameter die Bezeichnung der zu vergebenden Gruppe erwartet:

```
public static boolean add(String neueGruppe)
```

Im Zuge der Abarbeitung dieser Methode wird das Einfügen der neuen Bezeichnung in den Datenbestand durch die Übermittlung an alle Standorte über persistent gesicherte Warteschlangen durchgeführt, falls die Bezeichnung am lokalen Standort noch nicht vergeben war. Der Rückgabewert ist im Erfolgsfall `true` und ansonsten `false`.

Verarbeitung von AF6 Rechteverwaltung

Die Rechteverwaltung basiert direkt auf dem Robusten Commit-Protokoll. Daher existiert hierzu die Klasse `RechteverwaltungTransaction`, welche die lokal nötigen Anweisungen im Zuge des Robusten Commits durchführt. Die Rechteverwaltung wird durch die Anweisung

```
CommitCoordinator.execute(new RechteverwaltungTransaction(gruppe))
```

abgeschlossen, der eine Instanz der Klasse `Benutzergruppe` mit den gewünschten Änderungen in Form der Variable `gruppe` übergeben wird. Der Rückgabewert ist im Erfolgsfall `true` und ansonsten `false`.

Verarbeitung von AF7 Rechtegruppenverwaltung

Die Transaktion zur Rechtegruppenverwaltung wird mit der Klasse `RechtegruppenverwaltungTransaction` implementiert. Die Ausführung geschieht analog zu der der Rechteverwaltung mit den gewünschten Änderungen in der Variable `gruppe` durch die Anweisung

```
CommitCoordinator.execute(
new RechtegruppenverwaltungTransaction(gruppe)).
```

Auch hier ist der Rückgabewert im Erfolgsfall `true` und ansonsten `false`.

Verarbeitung von AF8 Erweiterung der Formatumwandlungsmöglichkeiten

Die Erweiterung der Formatumwandlungsmöglichkeiten geschieht in der Klasse `Formatuebersetzungsmoeglichkeitenerweiterung` durch die Methode

```
public static boolean erweitern(String funktionsname).
```

Dieser wird der neue Funktionsname mitgegeben. Sie prüft zunächst, ob das Plugin durch den PluginLoader korrekt geladen wird. Danach führt sie das Robuste Commit mit der Anweisung

```
CommitCoordinator.execute(
new Formatuebersetzungsmoeglichkeitenerweiterung(funktionsname))
```

durch und liefert den Rückgabewert der `execute` Methode zurück. Bei der Erstellung der Transaktionsklasse wird der Inhalt der aus dem Funktionsnamen ersichtlichen Datei am Quellstandort eingelesen und an den anderen Standorten gespeichert. Dazu werden die in Abschnitt 5.4.7 beschriebenen Methoden `loadFile` und `saveFile` verwendet.

Verarbeitung von AF9 Erweiterung der Merkmalsextraktionsmöglichkeiten

Die Erweiterung der Merkmalsextraktionsmöglichkeiten geschieht in der Klasse `Merkmalsextraktionsmoeglichkeitenerweiterung` durch die Methode

```
public static boolean erweitern(String funktionsname)
```

analog zur Erweiterung der Formatumwandlungsmöglichkeiten. Auch hier wird zunächst geprüft, ob das Plugin durch den PluginLoader korrekt geladen wird. Danach führt sie das Robuste Commit mit der Anweisung

```
CommitCoordinator.execute(
new Merkmalsextraktionsmoeglichkeitenerweiterung(funktionsname))
```

durch und liefert den Rückgabewert der `execute` Methode zurück. Bei der Erstellung der Transaktionsklasse wird der Inhalt der aus dem Funktionsnamen ersichtlichen Datei am Quellstandort eingelesen und an den anderen Standorten gespeichert. Dazu werden die in Abschnitt 5.4.7 beschriebenen Methoden `loadFile` und `saveFile` verwendet.

Verarbeitung von AF10 Anmeldung

Der Anmeldeprozess ist zweigeteilt. Das Anmelden geschieht über die in der Klasse `Anmelden` bereitgestellten Methoden

`public static Anmelden anmelden(String name)` und
`public static Anmelden anmelden(Anmelden anmelden)`.

Die erste der beiden Methoden wird mit dem Benutzernamen aufgerufen. Mit dem Ergebnis werden die Benutzerdaten geliefert, falls diese dem System bekannt sind. Die von der ersten Methode gelieferten Anmeldedaten müssen nun noch um das Passwort des Benutzers erweitert werden. Mit dem Ergebnis wird die zweite Methode aufgerufen, welche die eigentliche Anmeldung durchführt. Auf den Erfolg oder Mißerfolg kann durch Abfrage der Methode

```
public boolean istAngemeldet()
```

aus dem Ergebnis der zweiten `anmelden` Methode geschlossen werden.

Verarbeitung von AF11 Abmeldung

Die Abmeldung ist eine reine Formsache, denn es werden dabei keine gespeicherten Daten verändert. Das Abmelden geschieht mit dem Aufruf

```
Sendedienst.send(Auskunft.lokalerStandort(), new Abmelden()).
```

Der Erfolg ist bei Erreichbarkeit des lokalen Dämons garantiert.

Verarbeitung von AF12 Standorterweiterung

Die Standorterweiterung ist die umfangreichste der im System vorkommenden Aktionen in Bezug auf die Datenmenge. Sie wird durch die in der Klasse `Standorterweiterung` implementierte Methode

```
public static boolean standortErweiterung(Map conf)
```

aufgerufen. In der übergebenen Liste müssen die Konfigurationseinträge vollständig eingetragen sein. Außerdem muss die Liste die Einträge `benutzername` und `passwort` umfassen, welche die Benutzerbezeichnung und das Passwort des Administrators enthalten. Mit diesen Daten wird eine Fernauthentifizierung am angegebenen Quellstandort durchgeführt, falls nicht der allererste Standort eingerichtet wird. Bei diesem werden diese Daten als erste Benutzerdaten eingetragen.
Nötig und Voraussetzung für die Ausführung ist es, dass der neue Standort zuvor nicht

über die Konfigurationsdatei `vdas.conf` im Unterverzeichnis `conf` verfügt. Dies dient als Absicherung vor versehentlicher Durchführung, denn im Zuge der Standorterweiterung werden alle Daten in der konfigurierten Datenbank gelöscht und vom Quellstandort übertragene Daten eingefügt.
Im Verlauf der Synchronisation mit den bestehenden Standorten wird eine neue Standortnummer vergeben und den bestehenden Standorten bekannt gemacht.

5.5 Die Benutzerschnittstelle

Die Benutzerschnittstelle ist mit javax.swing realisiert. Sie nutzt für die Erfüllung ihrer Aufgaben die Codebasis mit dem Dienstprogramm gemeinsam und somit die gleichen Dienste, die bereits im vorherigen Abschnitt beschrieben wurden.
Wie ebenfalls zuvor beschrieben, kann sich an jedem Standort maximal ein Benutzer zur gleichen Zeit anmelden. Dieses stellt keine prinzipielle Einschränkung dar, vereinfacht aber die Implementation des Prototypen erheblich. Die im Konzept entwickelten Verfahren und Methoden lassen sich auch mit dieser Vereinfachung uneingeschränkt demonstrieren.
Ausgewählte Teile des Quellcodes sind im Anhang A.8 dargestellt, der gesamte Quellcode ist auf der beigefügten CD-ROM (siehe dazu auch Anhang C) abgelegt und einsehbar.

5.5.1 Der Start

Die GUI ist einfach gehalten und bildet die Anforderungen rudimentär ab. Beim Start der GUI wird zunächst mit einer Kontrollanfrage geprüft, ob der Dämon erreichbar ist. Dieser muss, auch bei der ersten Standorteinrichtung, vor der GUI gestartet werden. Verläuft die Prüfung erfolgreich, so wird die eigentliche Oberfläche geladen. Andernfalls wird das Programm mit einer Fehlermeldung beendet.

5.5.2 Die Oberfläche

Die einzelnen Funktionen sind über eine Menüleiste erreichbar. Diese ist in die Funktionsgruppen „Menü", „Daten", „Administration" und „Hilfe" eingeteilt.

In der Gruppe „Menü" gibt es die Menüpunkte „Anmelden", „Abmelden" und „Programm beenden". Die beiden Erstgenannten lösen die Anwendungsfälle AF10 und AF11 unter Rückgriff auf die in Abschnitt 5.4.8 beschriebenen Funktionen aus, falls die Stand-

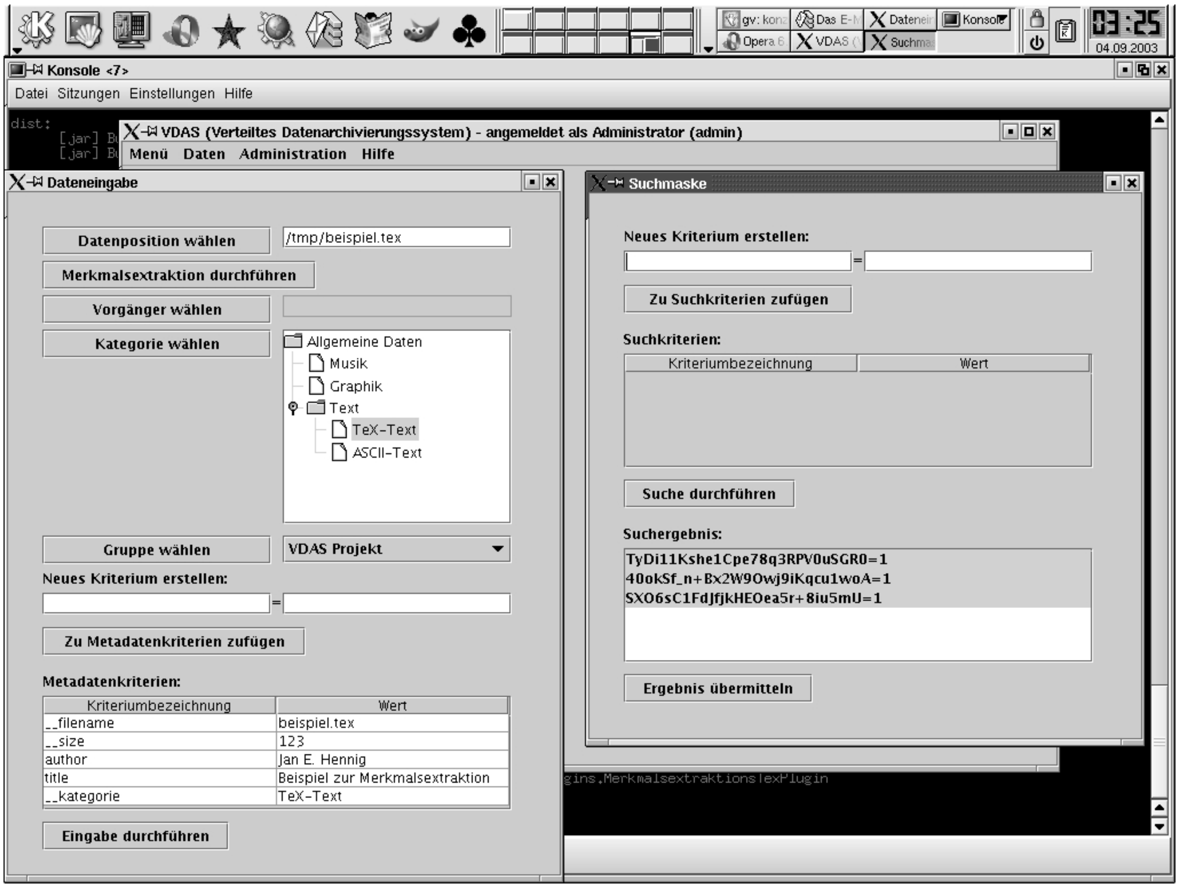

Abbildung 5.7: Bildschirmphoto von der Oberfläche:
Es findet gerade die Dateneingabe einer Beispieldatei im TeX-Format statt, aus der die Merkmale `author` und `title` extrahiert werden konnten. Auf der rechten Seite ist die Suchmaske zur Vorgängerbestimmung zu sehen. Der Benutzer ist als Administrator angemeldet.

orteinrichtung bereits durchgeführt wurde, andernfalls zeigen sie eine Fehlermeldung. Mit „Programm beenden" wird, falls der Benutzer angemeldet ist, eine Zwangsabmeldung ohne abschließende Erfolgsmeldung durchgeführt und in jedem Fall das GUI-Programm beendet.

In der Gruppe „Daten" existieren die Menüpunkte „Eingabe", „Ausgabe", „Kategorienerweiterung" und „Gruppenvergabe". Falls der Benutzer angemeldet ist, lösen sie die Anwendungsfälle AF1, AF3, AF4 und respektive AF5 aus, welche dann ebenfalls unter Rückgriff auf die in Abschnitt 5.4.8 beschriebenen Funktionen durchgeführt werden. Ist der Benutzer nicht angemeldet, so erscheint eine Fehlermeldung, die ihn auf diesen Zustand hinweist. Die Eingabe- und die Ausgabemaske bieten jeweils den Zugriff auf den Anwendungsfall AF2, der Datensuche. Bei der Eingabemaske geschieht dies zur Auswahl eines als Vorgänger zu deklarierenden Datensatzes, bei der Ausgabemaske, um die auszugebenden Daten zu suchen. Abbildung 5.7 auf der vorherigen Seite zeigt ein Bildschirmphoto der GUI während der Vorgängersuche, die im Zuge der Eingabe einer Beispieldatei durchgeführt wird. Falls bei der Datensuche mehr als ein Datensatz ausgewählt wurde, so wird die Auswahl automatisch und ohne weitere Rückmeldung auf einen Datensatz reduziert.

In der Gruppe „Administration" gibt es die Menüpunkte „Rechteverwaltung", „Rechtegruppenverwaltung", „Formatübersetzungsmöglichkeitenerweiterung", „Merkmalsextraktionsmöglichkeitenerweiterung" und „Standorterweiterung". Bis auf die Standorterweiterung lösen sie, falls ein Administrator angemeldet ist, die Anwendungsfälle AF6, AF7, AF8 bzw. AF9 aus, andernfalls wird eine dem Zustand entsprechende Fehlermeldung ausgegeben. Aus der Rechteverwaltungsmaske heraus kann zur Bestimmung der einzustellenden Daten-Rechte-Kombinationen auf den Anwendungsfall AF2 zugegriffen werden, dessen Ausgabe als Filterergebnis dient. Die Standorterweiterung löst den gleichnamigen Anwendungsfall AF12 aus, falls der Standort noch nicht eingerichtet ist und gibt andernfalls eine Fehlermeldung aus. All diese Funktionen werden unter Rückgriff auf die in Abschnitt 5.4.8 beschriebenen Funktionen durchgeführt.

Die Menügruppe „Hilfe" bietet einen Menüpunkt „Information" an, welcher eine kurze Beschreibung des Programms ausgibt.

5.5.3 Wichtige Klassen und Methoden

Das Hauptfenster wird in der Klasse `VDASGui` implementiert. Sie bildet mit den jeweiligen Unterfenstern, deren Oberklasse `VDASDialog` ist, eine Einheit. Bis auf die Suchmaske werden alle Unterfenster mit der Methode

```
private void showDialog(VDASDialog neuerDialog)
```

geöffnet und mit der Methode

```
public void closeDialog(VDASDialog dialog)
```

wieder geschlossen. Durch die Verwendung dieser Methoden wird sichergestellt, dass, bis auf die Ausnahme der Suchmaske, maximal ein Unterfenster gleichzeitig offen ist. Die Hauptfensterklasse stellt zwei weitere Methoden zur Anzeige von Informationsmeldungen und zur Fragestellung an den Benutzer zur Verfügung:

`public void showInfoMessage(String ueberschrift, String txt)` und
`public boolean showQuestion(String ueberschrift, String txt)`

Für die drei Eingabemasken, die die Datensuche verwenden, werden außerdem folgende zwei Methoden angeboten, die die Suchmaske öffnen und anweisen, das Suchergebnis an den jeweiligen Rezeptor zu übermitteln:

`public void executeSuchen(IdentifikatorRezeptor rezeptor)` und
`public void executeSuchen(IdentifikatorenRezeptor rezeptor)`

`IdentifikatorRezeptor` und `IdentifikatorenRezeptor` sind zwei Interfaces, die zur Übermittlung eines bzw. mehrerer Identifikatoren die Implementierung der Methoden

`public abstract void answerRezeptor(String identifikator)` bzw.
`public abstract void answerRezeptor(Set identifikatoren)`

einfordern. Die die jeweilige Maske implementierenden Klassen `Eingabemaske`, `Ausgabemaske` bzw. `Rechteverwaltungsmaske` implementieren dazu auch das jeweils geeignete Interface.

5.6 Beispielimplementierung einer Hardwareeinbindung

Eine sehr einfach gehaltene Beispielimplementierung einer Hardwareeinbindung ist mit der Klasse `HardwareeinbindungSimplePlugin` gegeben. Diese wird in Anhang B.1 wiedergegeben. Sie legt die Daten wie in der Konzeption in Abschnitt 4.4.1 beschrieben direkt im lokalen Dateisystem ab. Als Dateiname dient dabei direkt der die Daten eindeutig bezeichnende Identifikator. Dieser wird von der `save` Funktion direkt als Repertorieneintrag zurückgeliefert und dient somit auch der `load` Funktion zum Wiederfinden der Daten. Der Funktionsname für die Einbindung dieses Plugins lautet (ohne den Umbruch)

```
HardwareeinbindungSimplePlugin:
de.uni_bielefeld.rvs.plugins.HardwareeinbindungSimplePlugin
```

und wird im Rahmen der Standorterweiterung eingegeben.

Eine umfangreichere Hardwareeinbindung beispielsweise für den Zugriff auf ein Bandlaufwerk mit automatischem Bandwechselroboter kann leicht in das System eingebunden und die gespeicherten Daten so mit dem Archivierungssystem direkt verwaltet werden, indem die Bandnummern und Aufzeichnungspositionen in den Repertorieneintrag mit aufgenommen werden. So lassen sich aus Bandnummer und Aufzeichnungsposition wieder die gespeicherten Daten zurückgewinnen.
Das Hardwareeinbindungspluginsystem abstrahiert also und trennt, wie in den Anforderungen gewünscht, das Archivierungssystem leicht von der verwendeten Hardware und macht es so davon unabhängig.

5.7 Beispielimplementierung einer Formatübersetzung

Die Beispielimplementierung eines Formatübersetzungsplugins ist mit der Klasse `FormatuebersetzungsPsToPdfPlugin` umgesetzt worden. Sie benötigt das Skript

```
/usr/bin/ps2pdf
```

aus dem Programm „Ghostscript" und demonstriert damit, wie sich externe Übersetzungsprogramme in das Archivierungssystem einbinden lassen. Als Eingabeformat wird basierend auf der Ausgabe von „file" „`PostScript document text conforming at level 2.0`" vorausgesetzt. Das Zielformat lautet „`PDF document, version 1.2`". Anhang B.2 zeigt den ganzen Quellcode dieser Klasse. Als Funktionsname muss (ohne den Umbruch)

```
FormatuebersetzungsPsToPdfPlugin: ↩
de.uni_bielefeld.rvs.plugins.FormatuebersetzungsPsToPdfPlugin
```

in der Maske zur Formatumwandlungsmöglichkeitenerweiterung angegeben werden.

5.8 Beispielimplementierung einer Merkmalsextraktion

Die Beispielimplementierung eines Merkmalsextraktionsplugins ist in der Klasse `MerkmalsextraktionsTexPlugin` implementiert worden. Mit einer einfachen Zustandsmaschine wird eine im Format „`LaTeX 2e document text`" gespeicherte Datei nach dem Vorkommen von Autorenname und Titelbezeichnung durchsucht und bei Erfolg als extrahierte Kriterien `author` und `title` an den Aufrufer zurückgeliefert. Anhang B.3 zeigt den ganzen Quellcode dieser Klasse. Als Funktionsname muss (ohne den Umbruch)

```
MerkmalsextraktionsTexPlugin: ↩
de.uni_bielefeld.rvs.plugins.MerkmalsextraktionsTexPlugin
```

in der Maske zur Merkmalsextraktionsmöglichkeitenerweiterung angegeben werden.

5.9 Vergleich mit der Konzeption und Fazit aus der Implementation

In diesem Kapitel wurde das in Kapitel 4 konzipierte System implementiert. Es handelt sich um ein Verteiltes System. Dieses besteht aus einem Dienstprogramm, dem Dämon, von dem je eine Instanz an jedem Systemstandort ständig laufen sollte. Die Dämonen kommunizieren über ein festes Protokoll miteinander. Darüber hinaus gehört eine Benutzerschnittstelle, die GUI, zum System. Diese kommuniziert nur mit dem am lokalen Systemstandort laufenden Dämon. Die Kommunikation findet direkt statt, kein Dämon spricht einen anderen Dämon *über* einen dritten Dämon an.

Das System basiert auf Persistenzdienst, Kommunikations- und Datenbankverbindung. Darauf bauen die höhere Dienste Logging-Dienst, persistent gesicherte Warteschlange und das Robuste Commit-Protokoll auf. All diese Dienste werden für die Durchführung der Funktionen der Anwendungsfälle benötigt und benutzt.

Mit den persistent gesicherten Warteschlangen existiert nun ein System, das die Nachrichtenzustellung zwischen den Standorten zu garantieren vermag, auch wenn zwischenzeitlich Teile des Systems oder die Verbindungen zwischen den Systemteilen ausfallen sollten.
Mit dem Robusten Commit-Protokoll wurde ein System geschaffen, das durch Verwendung von lokalen Transaktionen, die die ACID-Anforderungen erfüllen, diese ACID-Anforderungen auch auf einer verteilten Ebene umsetzen kann. Dabei brauchen die Systemteile nur für die Dauer des Robusten Commits einer einstufigen Hierarchie unterworfen werden, welche zudem erst bei Bedarf eingerichtet werden kann.
Eine im verteilten System Eindeutigkeit garantierende Funktion, die ident-Funktion, konnte unter Beibehaltung der lokalen Berechenbarkeit und der Eigenschaft der Kontextabhängigkeit verwirklicht werden. Sie folgt dem in der Konzeption eingeschlagenen Weg.

Das konzipierte Datenmodell konnte direkt umgesetzt werden. Auch die Anwendungsfälle wurden strikt nach der jeweiligen Vorgangsbeschreibung der Konzeption ausgeführt. Damit erfüllt die Implementation die Konzeption.

Zu dem System selbst wurden drei Beispielimplementationen für eine Hardwareeinbindung, eine Formatübersetzung und eine Merkmalsextraktion geschaffen. Es zeigte sich dadurch, dass sich das System leicht im Funktionsumfang erweitern läßt.

Das implementierte System ist noch nicht perfekt. So bestehen bei der GUI noch Defizite in der Bedienbarkeit, denn es werden dem Benutzer nur die abstrakten Datenidentifikatoren angezeigt. Dieser Mißstand könnte in einem erweiterten System durch die Präsentation der zugehörigen Metadaten recht leicht behoben werden.
Auch stehen noch ausführliche Tests von Verarbeitungsgeschwindigkeiten in einem größeren Stil und für langsame Verbindungen zwischen den Systemstandorten aus. Durch das verwendete Schema, nach Möglichkeit weniger oft benötigte und potentiell große Datenmengen nicht zu übertragen, oft benötigte Daten hingegen sofort zu replizieren, kann aber erwartet werden, dass gegenüber Systemen, die beispielsweise für die häufig vorkommende Suchoperation an allen Systemstandorten nachfragen muss, ein deutlicher Geschwindigkeitsvorteil besteht.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein System zur Archivierung von Daten vorgestellt. Dabei galt es folgende Aufgaben zu erfüllen:

1. Langzeitarchivierung von allgemeinen Daten
2. Verlässlichkeit vor Datenverlust und Datenveränderung
3. Verlässlichkeit vor unberechtigtem Datenzugriff
4. Verteilung des Systems auf mehrere Standorte
5. Erweiterbarkeit des Systems durch Zufügen neuer Standorte
6. Abstraktion von der verwendeten Hardware
7. Versionsverwaltung der Daten
8. Freie Kategorisierbarkeit der Daten
9. Freie Gruppierbarkeit der Daten
10. Erweiterbare Merkmalsextraktionsmöglichkeit
11. Erweiterbare Formatübersetzungseinbindung

Die Aufgaben ergaben sich aus der allgemeinen Problemstellung eines Archivierungssystems und aus den Ergebnissen des Vergleichs von bestehenden Systemen. Aus dem Vergleich wiederum ergab sich in Kapitel 2, dass keines davon den Spagat zwischen Spezialisierung bei Beibehaltung der allgemeinen Anwendbarkeit beherrscht. Daraufhin wurden die grundlegenden, nützlichen Eigenschaften der bestehenden Systeme zusammengenommen und als Anforderungen an das in dieser Arbeit zu konzipierende System gestellt.

Die Konzeption und der folgend implementierte Prototyp zeigen nun, dass sich durchaus ein System erstellen läßt, das diese hohen Anforderungen erfüllt. Es besteht also keine prinzipielle Unverträglichkeit zwischen den einzelnen Anforderungen.

Die Langzeitarchivierung der Daten selbst wird durch die Zusicherung der Unversehrtheit der Daten gesichert. Dazu dient nicht nur das vorgestellte System allein, sondern auch ein das System umgebendes Konzept zum regelmäßigen Datenträgertausch. Die Verwendbarkeit der lange archivierten Daten kann durch geeignete, aber für jeden Einzelfall zu erstellende Formatübersetzung auch bei Veralten der Software, mit der die Daten ursprünglich verarbeitet wurden, sichergestellt werden. Datenveränderung wird durch die Verwendung geeigneter Prüfungen, im Falle des vorgestellten Systems durch Message-Digest-Funktionen, erkannt werden. Vor unberechtigtem Datenzugriff schützt die fest in das System integrierte Rechteverwaltung.

Das System selbst baut auf der Verteilung auf mehrere Standorte auf und nutzt die dadurch entstehenden Eigenschaften in geeigneter Weise zur Vermeidung unnötiger Kommunikation aus. In Systemen mit fester Hierarchie wäre eine Kommunikation mit dem zentralen Server unumgänglich. Fiele die Verbindung zu diesem zentralen Server aus, so könnte der abgetrennte Systemteil oder gar das gesamte System seine Arbeit nicht mehr durchführen. Durch das Verwenden von nur bei Bedarf (on demand) und für kurze Zeit errichteten Hierarchien und der genauen Trennung zwischen synchronisationsbedürftigen und nicht synchronisationsbedürftigen Daten kann beim vorliegenden System auch bei Ausfall von Systemteilen die Arbeit trotzdem teils uneingeschränkt, teils eingeschränkt weitergeführt werden. Dies ist einem kompletten Arbeitsausfall sicherlich vorzuziehen.

Durch das Zufügen neuer Standorte kann auch die Verarbeitungskapazität des Gesamtsystems gesteigert werden. Dieses wird durch die Kommunikationsgestaltung erreicht, welche versucht, häufig benötigte Daten einmalig zu replizieren und selten benötigte Daten nur bei Bedarf zu übertragen, um damit den Kommunikationsbedarf zu minimieren. Ein derartiges System skaliert besser als ein System, bei dem häufige Operationen die Kommunikationslast ansteigen lassen. Eine weitere Verbesserung des aktuell implementierten Zustands könnte in einigen Konstellationen durch das Verwenden von Multicasttechniken für die Übermittlung gemeinsamer Daten erreicht werden.

Die Verteilung der Daten geschieht auf sicherem Wege. Dazu dienen einerseits die neu entwickelten Techniken der persistent gesicherten Warteschlangen. Diese können die Zustellung einer Nachricht durch das Verwenden von persistentem Speicher garantieren. Weiterhin wurde das Robuste Commit-Protokoll ersonnen. Es ist für das Durchführen homogener verteilter Transaktionen geeignet. Das Robuste Commit-Protokoll basiert auf einem herkömmlichen Zwei-Phasen-Commit-Protokoll, löst aber das Problem der dort ungesicherten Übermittlung der globalen Entscheidung.

Das vorgestellte System abstrahiert von der verwendeten Hardware. Dadurch läßt es sich auch in heterogenen Einsatzgebieten betreiben. Durch die flexible Gestaltung lassen sich zudem neue Einsatzgebiete erschließen, an die zum Erstellungszeitpunkt noch nicht gedacht wurde oder gedacht werden konnte.

Eine Versionsverwaltung von Daten ist möglich und direkt in das verwendete Metadatenkonzept integriert. Ebenfalls integriert sind vom Benutzer frei zu gestaltende Ordnungsmittel. Ein Mittel ist das Einteilen der Daten in vom Benutzer eingerichtete Kategorien und die Suche nach ganzen Kategoriezweigen. Ein anderes Ordnungsmittel stellt die Gruppierbarkeit der Daten dar. Die Gruppen können dabei vom Benutzer frei definiert werden.

Mit dem flexiblen Pluginsystem und der ebenso flexiblen Gestaltung der Funktionsbezeichnungen ließen sich die Erweiterung des Systems um Formatübersetzungen und Merkmalsextraktionen erzielen, welche nachträglich die Funktionsfähigkeiten erweitern. Mit der Formatübersetzung lassen sich neben einer Erleichterung der alltäglichen Arbeit mit unterschiedlichen Formaten auch schwierigere Probleme, wie der Schutz der Daten vor dem Veralten der sie verwaltenden Software lösen. Die Merkmalsextraktion kann dem Benutzer zudem viel Arbeit für die Eingabe von Datenmerkmalen als Metadatenkriterien ersparen.

Es hat sich gezeigt, dass eine möglichst exakte Ausformulierung der Anforderungen sehr hilfreich bei der nachfolgenden Konzeption und Implementation ist. Werden Anforderungen nur grob und undifferenziert gestellt, so führt dies bei der Systementwicklung anschließend leichter zu Verständnisschwierigkeiten und Fehlentwicklungen. Die am Anfang stehende Mehrarbeit zahlt sich am Ende aus. Dieses Vorgehen ist nicht nur auf das Gebiet der Archivierungssysteme beschränkt, und daher allgemeingültig vorzuziehen.

Das hier entwickelte System kann als guter Ausgangspunkt dienen, um die Möglichkeiten von verteilten Archivierungssystemen weiter zu erforschen. So könnte es interessant sein, zu untersuchen, ob sich weitere Gebiete der Archivierung automatisieren und den Benutzern so lästige Arbeiten ersparen lassen.
Aber auch die für den Betrieb des Systems entwickelten Dienste für sich können auf anderen Gebieten, wie der Absicherung von Netzwerkkommunikation, dem Sicherstellen der Funktionsfähigkeit von (höheren) verteilten Protokollen bis hin zur Erforschung von verteilten Algorithmen dienlich sein.

Somit bietet der vorliegende Ansatz, Spezialisierung bei Beibehaltung der Verwendbarkeit für allgemeine Daten zu schaffen und die Bearbeitung auch großer Datenmengen durch Beachtung der praktischen Operationshäufigkeiten und durch Datenreplikations- und Übertragungsvermeidungsmittel zu ermöglichen sowie das Sicherstellen einer verlässlichen Übertragung und Synchronisation, eine gute Basis zur Entwicklung leistungsfähigerer und benutzerfreundlicherer Datenverarbeitungssysteme, insbesondere von Datenarchivierungssystemen.

Literaturverzeichnis

[1] Bacon, Harris: Operating Systems Concurrent And Distributed Software Design. Addison-Wesley, 2003, ISBN 0-321-11789-1

[2] S. M. Lang und P. C. Lockemann: Datenbankeinsatz. Springer-Verlag Berlin Heidelberg, 1995, ISBN 3-540-58558-3

[3] Käster: Diplomarbeit: Konzeption und Implementierung eines SQL-Datenbank-Backends zur Speicherung von Multimediadaten; Universität Bielefeld, März 2001

[4] Risnes, André Larsen: Asynchronous Lock Distribution in the New File Repository. Universitetet I Tromsø, Dezember 2000
`http://www.vermicelli.pasta.cs.uit.no/ipv6/students/andrer/doc/html/`
Verweis zur Beschreibung des Zwei-Phasen-Commit Protokolls:
`http://www.vermicelli.pasta.cs.uit.no/ipv6/students/andrer/doc/html/node18.html`
(URLs geprüft am 30.05.2003)

[5] Reference Model for an Open Archival Information System (OAIS). Blue Book; 01.01.2002; Adopted as ISO 14721:2002
`http://ssdoo.gsfc.nasa.gov/nost/isoas/ref_model.html`
(URL geprüft am 10.03.2003)

[6] Sietmann: Digitales Alzheimer - Maßnahmen gegen den Gedächtnisschwund bei digitalen Bilbiotheken. c't 25/2002 S. 52f, Verlag Heinz Heise, Hannover

[7] Rötzer: Wider das digitale Vergessen. Telepolis Magazin der Netzkultur, 18.02.2003
http://www.heise.de/tp/deutsch/inhalt/te/14211/1.html
(URL geprüft am 16.06.2003)

[8] Sietmann: Licht ins Darknet - Multimediadaten suchen und finden; c't 8/2003 S. 80f., Verlag Heinz Heise, Hannover

[9] Eckstein: Gemeinsames Dokumentenformat: Keine Einheitssprache in Sicht; Chip 6/2003 S. 98, Verlag Vogel Burda, München

[10] Lowell, Chen: Persistent Messages in Local Transactions. 17th ACM Symposium on Principles of Distributed Computing, Juni 1998
http://research.compaq.com/wrl/people/dlowell/papers/vistagrams.pdf
(URL geprüft am 04.07.2003)

[11] Hua, Chaoyang, Yafei, Bin, Xiaoming: A Scheme to Construct Global File System. CNDSL Peking University
http://net.cs.pku.edu.cn/~hh/pub/Han_scheme.pdf
(URL geprüft am 03.04.2003)

[12] Saltzer, Reed, Clark: End-to-End Arguments in System Design. Massachusetts Institute of Technology
http://www.reed.com/Papers/EndtoEnd.html
(URL geprüft am 10.03.2003)

[13] Patilla: Security Must Be Baked In, Not Painted On. eWeek, 06.08.2001
http://www.eweek.com/print_article/0,3688,a=11401,00.asp
(URL geprüft am 10.03.2003)

[14] Younger: The New BFS (Brendan File System). OSNews.com, 07.02.2003
http://www.osnews.com/printer.php?news_id=2762
(URL geprüft am 10.03.2003)

[15] Meyers Konversationslexikon, Eine Encyklopädie des allgemeinen Wissens, vierte Auflage, Leipzig, 1888-1889.
http://susi.e-technik.uni-ulm.de:8080/meyers/servlet/index
Der Eintrag „Archiv" erstreckt sich über zwei Druckseiten:
http://susi.e-technik.uni-ulm.de:8080/meyers/servlet/showSeite?
SeiteNr=0775&BandNr=1&textmode=false
http://susi.e-technik.uni-ulm.de:8080/meyers/servlet/showSeite?
SeiteNr=0776&BandNr=1&textmode=false
(URLs geprüft am 02.05.2003)

[16] Meyers grosses Taschenlexikon in 24 Bänden. Meyers Lexikonverlag, Mannheim 1981, Band 12, ISBN 3-411-01932-8

[17] Colsmann, Karin: Implementation und Test eines Spracherkennungsverfahrens für die biometrische Authentikation. Projektbericht im Rahmen des Projekts „Biometrik" am Arbeitsbereich AGN des FB Informatik der Universität Hamburg, 26.08.2001
http://agn-www.informatik.uni-hamburg.de/papers/doc/
projber_karin_colsman.ps
(URL geprüft am 18.06.2003)

[18] Getting a Grip on Access Control Terms; Camelot Information Technologies Ltd.; veröffentlicht in TECS (The Encyclopedia of Computer Security), 26.07.2001
http://www.itsecurity.com/papers/camelot.htm
(URL geprüft am 17.06.2003)

[19] Osborn, Sandhu, Munawer: Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. ACM Transactions on Information and Systems Security, vol.3, no. 2, ACM Inc. New York, 2000
http://www.csd.uwo.ca/faculty/sylvia/models.ps
(URL geprüft am 18.06.2003)

[20] Programm „Archie” der PROMUC GmbH
http://www.aboutit.de/view.php?ziel=/sguide/prog_h/ph_0800.htm
(URL geprüft am 20.03.2003)

[21] Archivierungssystem „smartDax” der Firma MetaSystems
http://www.meta-systems.de/Seite_in_gruen/smartdax2.html
(URL geprüft am 31.03.2003)

[22] Informations- und Archivierungssystem „MIAMI” (Münstersche Informations- und Archivsystem für multimediale Inhalte) an der Universitäts- und Landesbibliothek (ULB) Münster
http://miami.uni-muenster.de/
Berichte über MIAMI:
http://www.dl-forum.de/news/miami.html
http://idw-online.de/public/zeige_pm.html?pmid=55205
(URLs geprüft am 11.04.2003)

[23] EASY Archivierungssystem der EASY Software AG
http://www.ces-dresden.de/menu/loesungen/easy/easy.htm
(URL geprüft am 21.03.2003)

[24] AdAkta® Archivierungssystem für Praxis und Klinik der DAISY Archivierungssysteme GmbH
http://www.adakta.de/home/kontakt.htm
(URL geprüft am 02.04.2003)

[25] Artec Webbasiertes Archivierungssystem für das Evangelische Zentralarchiv (EZA) in Berlin der arTec - Gesellschaft für computerunterstützte Darstellungstechnik mbH
http://www.artec-berlin.de/html/gesamt.htm
(URL geprüft am 19.03.2003)

[26] PostgreSQL Datenbank
http://advocacy.postgresql.org/about/
(URL geprüft am 16.06.2003)

[27] Archivierungs- und Workflow-Management-System ARCFLOW der Hohsoft Produkte AG
http://www.hohsoft.ch/PRD_archiv.htm
(URL geprüft am 19.03.2003)

[28] Microfilm and Microfiche by Steve Dalton
http://www.nedcc.org/plam3/tleaf51.htm
(URL geprüft am 19.03.2003)

[29] DiVAN: Verteiltes audivisuelles Archivierungssystem der Technischen Universität Darmstadt
http://www.ito.tu-darmstadt.de/projects/divan/
(URL geprüft am 26.03.2003)

[30] DSpace der Massachusetts Institute of Technology Libraries
http://dspace.org/
(URL geprüft am 13.03.2003)

[31] Dublin Core Metadata Format
http://dublincore.org/
(URL geprüft am 13.03.2003)

[32] NDLTD - Networked Digital Library of Theses and Dissertations
http://www.ndltd.org/
(URL geprüft am 12.03.2003)

[33] Request for comments RFC1807
http://www.ietf.org/rfc/rfc1807.txt?number=1807
(URL geprüft am 12.03.2003)

[34] CVS-Versionskontrolle
http://www.cvshome.org/
(URL geprüft am 20.03.2003)

[35] BitKeeper Source Management von BitMover
http://www.bitkeeper.com/
(URL geprüft am 11.04.2003)

[36] Network File System (NFS) (hier: Linux-spezifische Seite)
http://nfs.sourceforge.net/
(URL geprüft am 21.03.2003)

[37] ps2pdf-Konverter, ein Teil von Ghostscript
http://www.cs.wisc.edu/~ghost/
(URL geprüft am 21.03.2003)

[38] eDonkey2000, dezentrales Filesharing beliebiger Dateien
http://www.edonkey2000.com/documentation/whatisit.html
(URL geprüft am 21.03.2003)

[39] AudioGalaxy, zentralisiertes Filesharing von mp3-Musik
http://www.audiogalaxy.com
(URL geprüft am 20.03.2003)

[40] SQL Tutorial (kurze Einführung)
http://www.1keydata.com/sql/sql.html
(URL geprüft am 21.03.2003)

[41] cdrdao (DOS) von Golden Hawk Technology
http://www.goldenhawk.com/
(URL geprüft am 11.04.2003)

[42] AudioID des Fraunhofer IIS (Institut Integrierte Schaltungen)
http://www.emt.iis.fhg.de/produkte/
(URL geprüft am 16.06.2003)

[43] Karamanolis, Mahalingam, Muntz, Zhang: DiFFS: a Scalable Distributed File System. HP Laboratories Palo Alto, 24.01.2001
http://www.hpl.hp.com/techreports/2001/HPL-2001-19.pdf
(URL geprüft am 07.03.2003)

[44] Sit, Morris: Security Considerations for Peer-to-Peer Distributed Hash Tables. Laboratory for Computer Science, Massachusetts Institute of Technology
http://www.pdos.lcs.mit.edu/papers/chord:security02/chord:security02.pdf
(URL geprüft am 16.06.2003)

[45] Grönvall, Westerlund, Pink: The Design of a Multicast-based Distributed File System. Swedish Institute of Computer Science
http://www.usenix.org/publications/library/proceedings/osdi99/full_papers/gronvall/gronvall.pdf
(URL geprüft am 16.06.2003)

[46] Compaq White Paper: Cluster File System in Compaq TruCluster Server. Compaq, August 2000
http://www.tru64unix.compaq.com/cluster/CFS_wp_Sept01.pdf
(URL geprüft am 16.06.2003)

[47] Thekkath, Mann, Lee: Frangipani: A Scalable Distributed File System. Systems Research, DEC, 2001
http://www.thekkath.org/papers/frangipani.pdf
(URL geprüft am 16.06.2003)

[48] Akinlar, Mukherjee: A Scalable Bandwidth Guaranteed Distributed Multimedia File System Using Network Attached Autonomous Disks. Panasonic Technologies Inc
http://citeseer.nj.nec.com/433008.html
(URL geprüft am 16.06.2003)

[49] Akinlar, Mukherjee: A Scalable Distributed Multimedia File System Using Network Attached Autonomous Disks. Panasonic Technologies Inc
http://citeseer.nj.nec.com/compress/0/papers/cs/20014/
http:zSzzSzwww.cs.umd.eduzSzuserszSzsaritmzSzpaperszSzdfs-
conf-paper.ps.gz/a-scalable-distributed-multimedia.ps
(URL geprüft am 16.06.2003)

[50] Mukherjee, Akinlar, Aref, Kamel: A Scalable High-Bandwidth Distributed File System for Multimedia Applications. Panasonic Technologies Lab, 1999
http://citeseer.nj.nec.com/408274.html
(URL geprüft am 16.06.2003)

[51] Cluster File Systems, Inc: Lustre: A Scalable, High-Performance File System. Lustre.org
http://www.lustre.org/docs/whitepaper.pdf
(URL geprüft am 16.06.2003)

[52] Miner: New Advances in the Filesystem Space; University of Minnesota, 2003
http://cda.mrs.umn.edu/~mine0057/fs.pdf
(URL geprüft am 16.07.2003)

[53] Rowstron, Druschel: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Heidelberg, November 2001.
http://www.cs.rice.edu/~druschel/publications/Pastry.pdf
(URL geprüft am 16.06.2003)

[54] Maymounkov, Mazières: Kademlia: A Peer-to-peer Information System Based on the XOR Metric. New York University, 2002
`http://kademlia.scs.cs.nyu.edu`
(URL geprüft am 16.04.2003)

[55] Stoica, Morris, Liben-Nowell, Karger, Kaashock, Dabek, Balakrishnan: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. University of California, Berkely / Massachusetts Institute of Technology Laboratory for Computer Science, 10.01.2002
`http://pdos.lcs.mit.edu/chord/`
(URL geprüft am 16.04.2003)

[56] Burkard: Herodotus: A Peer-to-Peer Web Archival System. Massachusetts Institute of Technology, Mai 2002
`http://www.pdos.lcs.mit.edu/papers/chord:tburkard-meng.pdf`
(URL geprüft am 16.06.2003)

[57] Dabek: A Cooperative File System. Massachusetts Institute of Technology, September 2001
`http://citeseer.nj.nec.com/dabek01cooperative.html`
(URL geprüft am 16.06.2003)

[58] „What is Adobe PDF?" – Beschreibung des Portable Document Format der Adobe Systems Incorporated
`http://www.adobe.com/products/acrobat/adobepdf.html`
(URL geprüft am 11.06.2003)

[59] „Adobe PostScript Developer Resources" der Adobe Systems Incorporated
`http://partners.adobe.com/asn/tech/ps/index.jsp`
(URL geprüft am 11.06.2003)

[60] FOLDOC (Free On-Line Dictionary Of Computing) des Imperial College London, Eintrag zu „lifted domain"

http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?lifted+domain
(URL geprüft am 11.06.2003)

[61] Erklärungsseite zu „Authentifizierung – Was bedeutet das?” der Integralis GmbH zu Heilbronn
http://www.articon.de/products_services/authentication.php
(URL geprüft am 12.06.2003)

[62] Weblexikon von www.fantastic-websites.de zu Schleswig, Eintrag zu „Dämon”
http://www.fantastic-websites.de/weblexikon.htm#daemon
(URL geprüft am 13.06.2003)

[63] Lucke: Diplomarbeit: Erweiterung des Hypercomputers um eine Komponente zum Dateitransfer und um eine Schnittstelle zu DQS/Codine oder NQS; Universität Rostock, Fachbereich Informatik, Juli 1999
http://wwwtec.informatik.uni-rostock.de/~lucke/diplom/diplom.ps.gz
(URL geprüft am 14.07.2003)

[64] Notationsübersicht zur Unified Modeling Language (UML) 1.4; oose.de GmbH, 2002
http://www.oose.de/uml
(URL geprüft am 10.07.2003)

[65] Magirus Guide - IBM DB2 Data Management Software; Magirus International GmbH, Stuttgart 2002.

Anhang

Anhang A

Dokumentation des VDAS Systems

Im Folgenden werden ausgewählte Klassen und Methoden des VDAS Systems vorgestellt.

A.1 Quellcodepakete

Der Quellcode des VDAS Systems ist in mehrere Pakete unterteilt:

- `auskunft`: Klassen zur zentralen Auskunft des Systems und zur Standortinformation: `Auskunft`, `Configuration`, `Standort` und `StandortData`.

- `awf`: Klassen zur Durchführung der Anwendungsfälle: `Abmelden`, `Anmelden`, `Datenausgabe`, `Dateneingabe`, `Formatuebersetzungsmoeglichkeitenerweiterung`, `Gruppenvergabe`, `KategorienTransaction`, `Merkmalsextraktionsmoeglichkeitenerweiterung`, `RechtegruppenverwaltungTransaction`, `RechteverwaltungTransaction` und `Standorterweiterung`.

- `benutzerverwaltung`: Klassen zur Rechte- und Benutzerverwaltung des Systems: `AuslieferrechteData`, `Benutzer`, `BenutzerData`, `Benutzergruppe`, `BenutzergruppenData`, `BenutzergruppenzugehoerigkeitenData` und `EinlieferrechteData`.

- client: Klassen und Interfaces zur Clientseite des Kommunikationsdienstes: Consignee, EnqueueLocal, Sendedienst und Sendeschlange.
- commit: Klassen des Robusten Commit-Protokolls: CommitCommon, CommitCoordinator, CommitParticipant, ExecuteLocalCommit, TransactionData, TransactionDataDummy und TransactionProtocol.
- daten: Klassen zur Verwaltung von Daten und Metadaten sowie zur Datensuche: Data, Daten, DatenData, DatenSuche, Identifikator und MetadatenData.
- datenbank: Die Klasse DatenbankGrundlagen, dem Datenbankverbindungsdienst.
- formatuebersetzung: Klassen zu Formaten und der Formatübersetzung: Formate, FormateData, Formatuebersetzung und FormatuebersetzungData.
- gruppen: Klassen zur Verwaltung der Datengruppen: Gruppen und GruppenData.
- gui: Klassen und Interfaces der Benutzerschnittstelle (GUI): AboutDialog, Anmeldemaske, Ausgabemaske, Eingabemaske, Formatuebersetzungsmoeglichkeitenerweiterungsmaske, Gruppenvergabemaske, IdentifikatorRezeptor, IdentifikatorenRezeptor, Kategorienerweiterungsmaske, Merkmalsextraktionsmoeglichkeitenerweiterungsmaske, Rechtegruppenverwaltungsmaske, Rechteverwaltungsmaske, RechteverwaltungsmaskeEinzelrechteTableModel, Standorterweiterungsmaske, StartGUI, Suchmaske, VDASDialog und VDASGui.
- hardwareeinbindung: Die Klasse Hardwareeinbindung.
- kategorien: Klassen zur Verwaltung der Kategorien: Kategorien und KategorienData.
- merkmalsextraktion: Klassen zur Merkmalsextraktion: Merkmalsextraktion und MerkmalsextraktionData.
- persistenz: Klassen für die Verwaltung des persistenten Speichers und der persistent gesicherten Warteschlangen: LaufnummernFilter, PersistentQueue und Persistenz.

- `pluginsystem`: Klassen des Pluginsystems: `FormatuebersetzungsPlugin`, `HardwareeinbindungsPlugin`, `MerkmalsextraktionsPlugin`, `Plugin` und `PluginLoader`.
- `server`: Klassen zur Serverseite des Kommunikationsdienstes: `StartDaemon`, `VDASBurstInformation`, `VDASDaemon`, `VDASDaemonThread`, `VDASInformation` und `VDASProtocol`.
- `util`: Hilfs- und Werkzeugklassen: `BooleanTuple`, `Err` und `Hash`.

Da die Diplomarbeit unter der Betreuung der Arbeitsgruppe Rechnernetze und Verteilte Systeme (RVS) der Universität Bielefeld entstanden ist, lautet das Prefix zu allen Paketnamen `de.uni_bielefeld.rvs`.

A.2 Wichtige Methoden und Klassen der Dienste

A.2.1 Der Datenbankverbindungsdienst

Der Datenbankverbindungsdienst wird durch die Klasse `DatenbankGrundlagen` implementiert. Wichtige Methoden sind `createConnection`, `count`, `countZero`, `getListSet`, `getListVector` und `execute`:

```
public static Connection createConnection() {
    try {
        Class.forName(Auskunft.get("metadaten.datenbank.treiber"));
        return DriverManager.getConnection(Auskunft.get("metadaten.datenbank.url"),
                                           Auskunft.get("metadaten.datenbank.benutzer"),
                                           Auskunft.get("metadaten.datenbank.passwort"));
    }
    catch (ClassNotFoundException e) {
        Err.msg("Verbindungsaufbau zur Datenbank fehlgeschlagen",e);
    }
    catch (SQLException e) {
        Err.msg("Verbindungsaufbau zur Datenbank fehlgeschlagen",e);
    }
    return null;
}

public static boolean countZero(String sqlquery) {
    return (count(sqlquery) == 0);
}
```

```
public static long count(String sqlquery) {
    long result = 0;
    try {
        Connection c = createConnection();
        Statement s = c.createStatement();
        ResultSet r = s.executeQuery(sqlquery);
        if (r.first()) {
            result = r.getLong("ANZAHL");
        }
        r.close();
        s.close();
        c.close();
    }
    catch (SQLException e) {
        Err.msg("Datenabfrage fehlgeschlagen: "+sqlquery,e);
    }
    return result;
}

public static Set getListSet(String sqlquery) {
    Set result = new LinkedHashSet();
    try {
        Connection c = createConnection();
        Statement s = c.createStatement();
        ResultSet r = s.executeQuery(sqlquery);
        r.beforeFirst();
        while (r.next()) {
            result.add(r.getString("EINTRAG"));
        }
        r.close();
        s.close();
        c.close();
    }
    catch (SQLException e) {
        Err.msg("Lesen der Liste fehlgeschlagen: "+sqlquery,e);
    }
    finally {
    }
    return result;
}

public static Vector getListVector(String sqlquery) {
    return new Vector(getListSet(sqlquery));
}

public static boolean execute(String sql) {
    boolean result = false;
    try {
        Connection c = createConnection();
        c.setAutoCommit(false);
        Statement s = c.createStatement();
        s.execute(sql);
        c.commit();
        s.close();
        c.close();
        result = true;
    }
    catch (SQLException e) {
        Err.msg("Execute fehlgeschlagen: "+sql,e);
    }
    finally {
    }
    return result;
}
```

A.2.2 Die Message-Digest-Berechnung

Die Message-Digest-Berechnung von Dateiinhalten und Texten erfolgt durch die Klasse `Hash`:

```
package de.uni_bielefeld.rvs.util;

import java.io.*;
import java.security.*;
import de.uni_bielefeld.rvs.auskunft.Auskunft;

public class Hash {

    private static MessageDigest createDigest() {
        String alg = Auskunft.get("hash.algorithmus");
        MessageDigest md = null;
        try {
            md = MessageDigest.getInstance(alg);
        }
        catch (NoSuchAlgorithmException e) {
            Err.msg("MessageDigest-Algorithmus "+alg+" nicht gefunden");
        }
        finally {
        }
        return md;
    }

    public static byte[] rawhashFile(String filename) {
        MessageDigest md = createDigest();
        if (md != null) {
            try {
                FileInputStream f = new FileInputStream(filename);
                int i = 0;
                byte[] buffer = new byte[131072];
                while (i >= 0) {
                    md.update(buffer,0,i);
                    i = f.read(buffer);
                }
                f.close();
            }
            catch (IOException e) {
                Err.msg("Ein-/Ausgabefehler bei MessageDigest-Berechnung von Datei "+filename,e);
            }
            finally {
            }
            return md.digest();
        } else {
            return null;
        }
    }
    public static String hashFile(String filename) {
        return (new sun.misc.BASE64Encoder().encode(rawhashFile(filename))).replace('/','_');
    }

    public static byte[] rawhash(String txt) {
        MessageDigest md = createDigest();
        if (md != null) {
            return md.digest(txt.getBytes());
        } else {
            return null;
        }
    }
```

```
    public static String hash(String txt) {
        return (new sun.misc.BASE64Encoder().encode(rawhash(txt))).replace('/','_');
    }
}
```

A.2.3 Die Klasse Data

Die Klasse Data ist die Oberklasse für alle Datentransferklassen. Die Datentransferklassennamen enden auf Data.

```
package de.uni_bielefeld.rvs.daten;

import java.io.*;
import java.sql.*;
import java.util.*;
import de.uni_bielefeld.rvs.datenbank.*;
import de.uni_bielefeld.rvs.util.*;

public abstract class Data implements Serializable {

    protected abstract String getDataBezeichnung();
    protected abstract String getDataSelectAll();
    protected abstract String getDataInsert();
    protected abstract Data getDataData(ResultSet r) throws SQLException;

    protected static synchronized Set retrieveAll(Data dummy) {
        Set data = new LinkedHashSet();
        try {
            Connection c = DatenbankGrundlagen.createConnection();
            c.setAutoCommit(false);
            Statement s = c.createStatement();
            Err.dbg("Lese alle "+dummy.getDataBezeichnung()+"...");
            ResultSet r = s.executeQuery(dummy.getDataSelectAll());
            r.beforeFirst();
            while (r.next()) {
                data.add(dummy.getDataData(r));
            }
            r.close();
            s.close();
            c.close();
        }
        catch (SQLException e) {
            Err.msg("SQL Fehler beim Lesen von "+dummy.getDataBezeichnung(),e);
        }
        finally {
        }
        return data;
    }

    protected static synchronized boolean feedAll(Data dummy, Set data) {
        boolean result = false;
        Data d = null;
        try {
            Connection c = DatenbankGrundlagen.createConnection();
            c.setAutoCommit(false);
            Statement s = c.createStatement();
            Err.dbg("Füge alle "+dummy.getDataBezeichnung()+" zu...");
            for (Iterator i = data.iterator(); i.hasNext(); ) {
                d = (Data) i.next();
                s.executeUpdate(d.getDataInsert());
            }
```

```
            c.commit();
            s.close();
            c.close();
            result = true;
        }
        catch (SQLException e) {
            Err.msg("SQL Fehler beim Einfügen von "+dummy.getDataBezeichnung()+" "+d,e);
        }
        finally {
        }
        return result;
    }
}
```

A.3 Die persistent gesicherte Warteschlange

Die Klasse `PersistentQueue` implementiert die persistent gesicherte Warteschlange auf Basis des Persistenzdienstes:

```
package de.uni_bielefeld.rvs.persistenz;

import java.io.*;
import java.util.*;
import de.uni_bielefeld.rvs.util.*;

public class PersistentQueue {

    private static int QUEUESIZE = 63;
    private static String queueExtension = ".vdasqueue";

    private int front;
    private int tail;
    private String prefix;

    public PersistentQueue(String filenamePrefix) {
        this.prefix = filenamePrefix;
        recover();
    }

    public String toString() {
        return "[front="+front+", tail="+tail+", prefix="+prefix+"]";
    }

    private void recover() {
        int[] leftOver = Persistenz.getStrippedIds(prefix, queueExtension);
        TreeSet candidates = new TreeSet();
        for (int i = 0; i < leftOver.length; i++) {
            candidates.add(new Integer(leftOver[i]));
        }

        boolean[] buckets = new boolean[QUEUESIZE];
        boolean empty = true;
        String debug = "";
        for (int i = 0; i < QUEUESIZE; i++) {
            buckets[i] = candidates.contains(new Integer(i));
            if (buckets[i]) {
                empty = false;
            }
            debug += (buckets[i]) ? "1" : "0";
        }
        Err.dbg(debug+" "+empty);
```

```
        if (empty) {
            front = 0;
            tail = 0;
        } else {
            if (buckets[0]) {
                if (buckets[QUEUESIZE-1]) {
//                  Err.dbg("Recovery case 1: overlapping");
                    tail = 1;
                    while (buckets[tail]) {
                        tail++;
                    }
                    front = tail;
                    while (!buckets[front]) {
                        front++;
                    }
                } else {
//                  Err.dbg("Recovery case 2: starts at 0");
                    front = 0;
                    tail = 1;
                    while (buckets[tail]) {
                        tail++;
                    }
                }
            } else {
                if (buckets[QUEUESIZE-1]) {
//                  Err.dbg("Recovery case 3: ends at QUEUESIZE-1");
                    tail = 0;
                    front= 1;
                    while (!buckets[front]) {
                        front++;
                    }
                } else {
//                  Err.dbg("Recovery case 4: normal case");
                    front = 1;
                    while (!buckets[front]) {
                        front++;
                    }
                    tail = front;
                    while (buckets[tail]) {
                        tail++;
                    }
                }
            }
        }
        Err.dbg("Recovery: "+this);
    }

    private int next(int actual) {
        return (actual + 1) % QUEUESIZE;
    }

    public synchronized boolean isEmpty() {
        Err.dbg("isEmpty="+(front == tail));
        return (front == tail);
    }

    public synchronized boolean isFull() {
        // Einen Speicherplatz für Recover-Funktion freilassen
        return (tail == front - 2);
    }

    public synchronized Serializable top() {
        return Persistenz.readSingle(prefix+front+queueExtension);
    }
```

```
    public synchronized Serializable enqueue(Serializable element) {
        if (!isFull()) {
            Persistenz.writeSingle(prefix+tail+queueExtension, element);
            tail = next(tail);
            return element;
        } else {
            return null;
        }
    }

    public synchronized Serializable dequeue() {
        if (!isEmpty()) {
            Serializable element = top();
            Persistenz.deleteFile(prefix+front+queueExtension);
            front = next(front);
            return element;
        } else {
            return null;
        }
    }
}
```

A.4 Der Kommunikationsdienst

Der Kommunikationsdienst besteht aus einer Protokolloberklasse und den client- und serverseitigen Verarbeitungsmethoden.

A.4.1 Die Protokollklasse

Die Oberklasse aller Protokollklassen stellt `VDASProtocol` dar.

```
package de.uni_bielefeld.rvs.server;

import java.io.*;

public abstract class VDASProtocol implements Serializable {

    public String helo = "VDAS Protocol v1.0";

    private boolean shallCloseConnection = false;

    public boolean verbindungSchliessen() {
        return shallCloseConnection;
    }

    protected void schliesseVerbindung() {
        shallCloseConnection = true;
    }

    public abstract Serializable execute();

    public abstract Serializable executeLocal();
}
```

A.4.2 Die Empfängermethode

Auf der Serverseite der Kommunikationsverbindung wird eine eingehende Anfrage mit der Ausführung eines `VDASDaemonThread` verarbeitet:

```
package de.uni_bielefeld.rvs.server;

import java.io.*;
import java.net.*;
import de.uni_bielefeld.rvs.util.*;

public class VDASDaemonThread extends Thread {

    private static final int TIMEOUT = 10000;

    private Socket socket = null;
    private int instance = 0;

    public VDASDaemonThread(Socket socket, int instance) {
        super("VDASDaemonThread"+instance);
        this.socket = socket;
        this.instance = instance;
    }

    public void run() {
        try {
            socket.setSoTimeout(TIMEOUT);
            ObjectInput in = new ObjectInputStream(new BufferedInputStream(socket.getInputStream()));
            ObjectOutput out = new ObjectOutputStream(socket.getOutputStream());
            Object o = null;

            yield();
            while((o = in.readObject()) != null) {
                VDASProtocol inp = (VDASProtocol) o;
                Err.dbg("Verarbeite Anfrage von "+socket.getInetAddress()+":"+socket.getPort()+" #"+
                   instance+"...");
                Serializable outp = inp.execute();
                if (outp != null) {
                    Err.dbg("Antworte auf Anfrage von "+socket.getInetAddress()+":"+socket.getPort()+
                    " #"+instance+"...");
                    out.writeObject(outp);
                }
                if (inp.verbindungSchliessen()) {
                    break;
                }
            }
            out.close();
            in.close();
            socket.close();
        }
        catch (ClassNotFoundException e) {
            Err.msg("Instanz "+instance+": Klasse nicht gefunden",e);
        }
        catch (SocketTimeoutException e) {
            Err.msg("Überschreitung der Wartezeit, Instanz "+instance,e);
        }
        catch (SocketException e) {
            Err.msg("Verbindung wurde zurückgesetzt, Instanz "+instance,e);
        }
        catch (IOException e) {
            Err.msg("Ein-/Ausgabefehler, Instanz "+instance,e);
        }
        System.gc();
    }
}
```

A.4.3 Die Sendemethoden

Auf der Clientseite der Kommunikationsverbindungen existieren drei Methoden zum Absenden einer Nachricht:

```
    public static VDASProtocol send(Standort to, VDASProtocol msg) {
        Integer id = getId();
        new Sendedienst(id,to,msg).run();
        return waitFor(id);
    }

    public static Integer send(Standort to, VDASProtocol msg, Consignee consignee) {
        Integer id = getId();
        new Sendedienst(id,to,msg,consignee).start();
        return id;
    }

    public static boolean enqueue(Standort to, VDASProtocol msg) {
        VDASProtocol answer = send(Auskunft.lokalerStandort(), new EnqueueLocal(to, msg));
        return ((EnqueueLocal) answer).result();
    }
```

A.4.4 Die Sendeschlange

Die Sendeschlange fußt auf der Klasse `PersistentQueue` zur Absicherung vor einem Systemausfall:

```
package de.uni_bielefeld.rvs.client;

import java.util.*;
import de.uni_bielefeld.rvs.auskunft.*;
import de.uni_bielefeld.rvs.persistenz.*;
import de.uni_bielefeld.rvs.server.*;
import de.uni_bielefeld.rvs.util.*;

public class Sendeschlange extends Thread {

    private static final int DELAY = 20000;
    private static Map sendeschlangen = new LinkedHashMap();
    private PersistentQueue queue;
    private Standort to;

    public Sendeschlange(Standort to) {
        this.to = to;
        queue = new PersistentQueue("queue_to_"+to.id()+"_");
    }

    public synchronized boolean enqueue(VDASProtocol msg) {
        if (Auskunft.standortLock(to)) {
            return false;
        } else {
            if (queue.enqueue(msg) != null) {
                notify();
                return true;
            } else {
                return false;
            }
        }
    }
```

```
private synchronized VDASProtocol getNext() {
    while (queue.isEmpty()) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
        finally {
        }
    }
    return (VDASProtocol) queue.top();
}

public void run() {
    VDASProtocol nextMessage;
    VDASProtocol reply;
    while (true) {
        nextMessage = getNext();
        reply = Sendedienst.send(to, nextMessage);
        if (reply != null) {
            queue.dequeue();
        } else {
            try {
                sleep(DELAY);
            }
            catch (InterruptedException e) {
            }
            finally {
            }
        }
    }
}

public static boolean isEmpty(Standort to) {
    return getQueue(to).queue.isEmpty();
}

public static synchronized Sendeschlange getQueue(Standort to) {
    if (!sendeschlangen.containsKey(new Integer(to.id()))) {
        Sendeschlange neu = new Sendeschlange(to);
        sendeschlangen.put(new Integer(to.id()), neu);
        neu.start();
    }
    return (Sendeschlange) sendeschlangen.get(new Integer(to.id()));
}

public static synchronized boolean enqueue(Standort to, VDASProtocol msg) {
    return getQueue(to).enqueue(msg);
}

public static void recoverAll() {
    if (!Auskunft.isFirsttime()) {
        Standort[] ferneStandorte = Auskunft.ferneStandorte();
        for (int i = 0; i < ferneStandorte.length; i++) {
            getQueue(ferneStandorte[i]);
        }
        getQueue(Auskunft.lokalerStandort());
    }
}
```

A.5 Das Robuste Commit-Protokoll

Das Robuste Commit-Protokoll besteht aus Koordinator und Teilnehmer. Beide speichern die Transaktionsdaten in einer Variable `tadata`.

A.5.1 Wichtige Methoden im `CommitCoordinator`

```
public boolean firstStage() {
    switch (state) {
        case CCS_INIT:
            clearExpired();
            participants = collectParticipants();
            logBegin();
            return firstStage();
        case CCS_BEGIN:
            for (Iterator i = participants.iterator(); i.hasNext(); ) {
                send((Standort) i.next());
            }
            return collectVotes();
        default:
            return false;
    }
}

public boolean secondStage(boolean globalCommit) {
    Err.dbg("Second Stage aufgerufen mit "+globalCommit);
    switch (state) {
        case CCS_BEGIN:
            if (globalCommit) {
                logCommit();
            } else {
                logAbort();
            }
            return secondStage(globalCommit);
        case CCS_ABORT:
        case CCS_COMMIT:
            for (Iterator i = commitVoters.iterator(); i.hasNext(); ) {
                sendPermanent((Standort) i.next(), globalCommit);
            }
            logDone();
            return globalCommit;
        default:
            return false;
    }
}
```

A.5.2 Wichtige Methoden im `CommitParticipant`

```
public boolean firstStage() {
    switch (state) {
        case CPS_INIT:
            if (tadata.setup() && executeTransaction()) {
                logCommit();
                return true;
            } else {
                logAbort();
                logDone();
                return false;
            }
```

```
            case CPS_STAGE1ABORT:
                return false;
            case CPS_STAGE1COMMIT:
                return true;
            default:
                return false;
        }
    }

    public boolean secondStage(boolean globalCommit) {
        switch (state) {
            case CPS_STAGE1COMMIT:
                if (globalCommit) {
                    logGlobalCommit();
                    tadata.commit();
                    logDone();
                    return true;
                } else {
                    logGlobalAbort();
                    tadata.abort();
                    logDone();
                    return true;
                }
            default:
                return false;
        }
    }

    private void recover(List logEntries) {
        state = getLogState(logEntries);
        switch (state) {
            case CPS_STAGE1ABORT:
                logDone();
                break;
            case CPS_STAGE1COMMIT:
                tadata = getTransactionData(logEntries);
                tadata.setup();
                Thread.yield();
                executeTransaction();
                break;
            case CPS_STAGE2ABORT:
                logDone();
                break;
            case CPS_STAGE2COMMIT:
                tadata = getTransactionData(logEntries);
                tadata.setup();
                Thread.yield();
                executeTransaction();
                tadata.commit();
                logDone();
                break;
            case CPS_DONE:
            default:
                break;
        }
    }
```

A.6 Das Pluginsystem

Das Pluginsystem besteht aus der Klasse `Plugin` und dem `PluginLoader`. Die Klasse `Plugin` ist die Oberklasse aller Plugins und dient als Indikator für die Zugehörigkeit einer Klasse zum Pluginsystem.

A.6.1 Die Methode `loadPlugin`

```
public static Plugin loadPlugin(String pluginName) {
    Plugin plugin = null;
    String pluginFullName = pluginFullName(pluginName);
    String pluginFileName = pluginFileName(pluginName);
    try {
        URL url = (new File(pluginFileName+".jar")).toURL();
        URL[] urls = new URL[1];
        urls[0] = url;
        ClassLoader cload = new URLClassLoader(urls);
        Class cl = cload.loadClass(pluginFullName);
        Constructor con = cl.getConstructor(new Class[]{});
        plugin = (Plugin) con.newInstance(new Object[]{});
        return plugin;
    }
    catch (ClassNotFoundException e) {
        Err.msg("Pluginklasse nicht gefunden: "+pluginFullName);
    }
    catch (Exception e) {
        Err.msg("Fehler beim Laden des Plugins: "+pluginFullName,e);
    }
    return null;
}
```

A.7 Funktionen der Anwendungsfallbearbeitung

A.7.1 Methoden zu AF1 Dateneingabe

Folgend ist die Methode `insert` aus der Klasse `Dateneingabe` dargestellt mit der neue Daten zusammen mit den zugehörigen Metadaten ins System aufgenommen werden.

```
public static boolean insert(String filename, Map metadaten) {
    boolean result = false;

    // Datenidentifikator vergeben
    Identifikator id = new Identifikator(filename);

    if (id.istNeu()) {
        // Metadatenabhängigkeiten berechnen
        if (metadaten.containsKey("__vorgänger")) {
            metadaten.put("__version",""+(Daten.version((String) metadaten.get("__vorgänger"))+1));
        } else {
            metadaten.put("__version","0");
        }

        // Daten lokal aufnehmen
        id.speichern();
        id.writeToDB();

        // Metadaten aufnehmen
        Set standorte = Auskunft.ferneStandorteSet();
        standorte.add(Auskunft.lokalerStandort());
```

```
        result = true;
        for (Iterator i = standorte.iterator(); i.hasNext(); ) {
            if (!Sendedienst.enqueue((Standort) i.next(),
                        new Dateneingabe(Auskunft.lokalerStandort(), id.id(), metadaten))) {
                result = false;
            }
        }
    } else {
        Err.dbg("Datenidentifikator wurde bereits vergeben. Abbruch.");
    }

    return result;
}
```

A.7.2 Methoden zu AF2 Datensuche

Die Klasse `DatenSuche` besteht aus der Methode `suche` sowie zwei Hilfsmethoden zur Bestimmung der Suchparameter anhand der übergebenen Suchkriterien:

```
package de.uni_bielefeld.rvs.daten;

import java.sql.*;
import java.util.*;
import de.uni_bielefeld.rvs.auskunft.*;
import de.uni_bielefeld.rvs.benutzerverwaltung.*;
import de.uni_bielefeld.rvs.datenbank.*;
import de.uni_bielefeld.rvs.util.*;

public class DatenSuche {

    private static String firstPart(Map kriterien) {
        String result = "";
        int count;
        for (count = 1; count <= kriterien.size(); count++) {
            result += " left outer join METADATEN metadaten"+count+" on DATEN.ID = metadaten"+count+".ID ";
        }
        return result;
    }

    private static String secondPart(Map kriterien) {
        String result = "";
        int count = 0;
        Map.Entry entry;
        for (Iterator i = kriterien.entrySet().iterator(); i.hasNext(); ) {
            count++;
            entry = (Map.Entry) i.next();
            result += " and metadaten"+count+".MBEZ = '"+entry.getKey()+"' and metadaten"+count+".WERT ";
            if ((entry.getKey()).equals("__kategorie")) {
                result += "in (select kat2.KATEGORIE from KATEGORIEN as kat1, KATEGORIEN as kat2 "+
                    "where kat2.L between kat1.L and kat1.R and kat1.KATEGORIE = '"+entry.getValue()+"') ";
            } else {
                result += "= '"+entry.getValue()+"' ";
            }
        }
        return result;
    }
```

```
    public static Set suche(Map kriterien) {
        Set result = new LinkedHashSet();
        try {
            Connection connection = DatenbankGrundlagen.createConnection();
            Statement statement = connection.createStatement();
            String bbez = Auskunft.getLokalenBenutzer().getData().bezeichnung;
            int suchbit = Benutzergruppe.BENUTZERGRUPPENSTATUSBIT_STANDARDSUCHRECHT;
            String query;
            if (Auskunft.getLokalenBenutzer().istAdministrator()) {
                query = "select DATEN.ID from DATEN "+firstPart(kriterien)+" where true "+
                   secondPart(kriterien)+";";
            } else {
                query =
                "select DATEN.ID from DATEN "+firstPart(kriterien)+
                    "where DATEN.ID not in "+
                        "(select ID from AUSLIEFERUNGSRECHTE where BGBEZ in "+
                   "(select BGBEZ from BENUTZERGRUPPENZUGEHOERIGKEIT where BBEZ = '"+bbez+"')) "+
                    "and DATEN.ID in "+
                        "(select case "+
                            "when true in "+
                                "(select (STATUS & "+suchbit+") = "+suchbit+
                                " from BENUTZERGRUPPEN natural join BENUTZERGRUPPENZUGEHOERIGKEIT "+
                    "where BBEZ = '"+bbez+"') "+
                            "then DATEN.ID end from DATEN) "+secondPart(kriterien) +
            "union "+
                "select DATEN.ID from DATEN natural join AUSLIEFERUNGSRECHTE "+firstPart(kriterien)+
                    "where BGBEZ in "+
                        "(select BGBEZ from BENUTZERGRUPPEN natural join BENUTZERGRUPPENZUGEHOERIGKEIT "+
                   "where BBEZ = '"+bbez+"') "+"and SUCHRECHT = true "+secondPart(kriterien)+";";
            }
            Err.dbg("query="+query);
            ResultSet r = statement.executeQuery(query);
            r.beforeFirst();
            while (r.next()) {
                result.add(r.getString("ID"));
            }
            r.close();
            statement.close();
            connection.close();
        }
        catch (SQLException e) {
            Err.msg("SQL-Fehler beim Durchführen der Datensuche",e);
        }
        finally {
        }
        return result;
    }
}
```

A.7.3 Methoden zu AF4 Kategorienerweiterung

Die Kategorienerweiterung nutzt das Robuste Commit-Protokoll und wird hier stellvertretend für die dieses Protokoll ebenfalls benutzenden Anwendungsfälle AF6, AF7, AF8 und AF9 dargestellt:

```
package de.uni_bielefeld.rvs.awf;

import java.sql.*;
import de.uni_bielefeld.rvs.kategorien.*;
import de.uni_bielefeld.rvs.commit.*;
import de.uni_bielefeld.rvs.client.*;
import de.uni_bielefeld.rvs.server.*;
import de.uni_bielefeld.rvs.util.*;

public class KategorienTransaction extends TransactionData {

    private String neueKategorie;
    private String oberKategorie;

    public KategorienTransaction(String neueKategorie, String oberKategorie) {
        super();
        this.neueKategorie = neueKategorie;
        this.oberKategorie = oberKategorie;
    }

    public String toString() {
        return "[kategorien="+super.toString()+" neu="+neueKategorie+", ober="+oberKategorie+"]";
    }

    public boolean execute() {
        Err.dbg("Transaction "+transactionID+" EXECUTE");
        boolean result = false;
        try {
            Kategorien.executeTransaction(connection, statement, neueKategorie, oberKategorie);
            result = true;
        }
        catch (Exception e) {
            Err.dbg("KategorienTransaktion fehlgeschlagen: "+neueKategorie);
        }
        finally {
        }
        return result;
    }
}
```

A.8 Methoden der VDAS GUI

A.8.1 Start der GUI

Der Start der Benutzerschnittstelle geschieht über die Klasse `StartGUI`.

```
package de.uni_bielefeld.rvs.gui;

import de.uni_bielefeld.rvs.auskunft.*;
import de.uni_bielefeld.rvs.client.*;
import de.uni_bielefeld.rvs.server.*;
import de.uni_bielefeld.rvs.util.*;

public class StartGUI {

    private static final int EXIT_CONFIGDEFECT = -1;

    public static void main(String[] args) {
        startGUI();
    }
```

```
    private static boolean serverErreichbar() {
        Standort lokal = null;
        if (Auskunft.isFirsttime()) {
            int port = Auskunft.getInt("daemon.port");
            if (port == 0) {
                Err.msg("Dämon-Port nicht gefunden. Stellen Sie sicher, dass in der "+
                "Konfigurationsdatei ein gültiger Wert für daemon.port eingetragen ist "+
                "und starten Sie den lokalen Dämon vor der GUI!");
                return false;
            }
            lokal = new Standort(-1, "localhost", port);        // behelfsmäßigen Standort bauen
        } else {
            lokal = Auskunft.lokalerStandort();
        }
        if (Sendedienst.send(lokal, new VDASInformation()) == null) {
            Err.msg("Konnte den Dämon nicht erreichen. Überprüfen Sie die Konfiguration "+
            "und starten Sie den Dämon vor der GUI!");
            return false;
        } else {
            return true;
        }
    }

    private static void startGUI() {
        if (!serverErreichbar()) {
            System.exit(EXIT_CONFIGDEFECT);
        }
        if (Auskunft.isFirsttime()) {
            Err.dbg("Das Programm wurde noch nicht konfiguriert. Führen Sie die "+
            "Standorterweiterung durch!");
        }
        Err.dbg("Starte GUI...");
        new VDASGui().show();
    }
}
```

A.8.2 Wichtige Methoden der Klasse VDASGui

Die Klasse VDASGui stellt das Hauptfenster und die Verbindungen zu den jeweiligen Eingabemasken dar:

```
    public void closeDialog(VDASDialog dialog) {
        if (modalDialog == dialog) {
            modalDialog.hide();
            modalDialog.dispose();
            modalDialog = null;
        }
    }

    private void showDialog(VDASDialog neuerDialog) {
        if (modalDialog == null) {
            modalDialog = neuerDialog;
            modalDialog.show();
        }
    }

    private boolean istAngemeldet() {
        return (Auskunft.getLokalenBenutzer() != null);
    }
```

```
private boolean istAdministrator() {
    return Auskunft.getLokalenBenutzer().istAdministrator();
}

private boolean istEingerichtet() {
    return !Auskunft.isFirsttime();
}

public void showInfoMessage(String ueberschrift, String txt) {
    JOptionPane.showMessageDialog(null, txt, ueberschrift+" Information",
                 JOptionPane.INFORMATION_MESSAGE);
}

public boolean showQuestion(String ueberschrift, String txt) {
    return (JOptionPane.showConfirmDialog(null, txt, ueberschrift+" Frage",
            JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION);
}
```

Anhang B

Dokumentation der Beispielimplementierungen

B.1 Die Beispielimplementierung einer Hardwareeinbindung

Die Beispielimplementierung der Hardwareeinbindung besteht in der Klasse `HardwareeinbindungSimplePlugin`. Sie speichert die Daten in einem separaten Unterverzeichnis direkt im Dateisystem des Wirtsrechners. Als Dateiname dient der Identifikator der Daten. Dieser wird auch für den Repertorieneintrag von der Methode `save` zurückgeliefert.

```
package de.uni_bielefeld.rvs.plugins;

import java.io.*;
import de.uni_bielefeld.rvs.pluginsystem.*;
import de.uni_bielefeld.rvs.util.Err;

public class HardwareeinbindungSimplePlugin extends HardwareeinbindungsPlugin {

    private static final String STORAGEDIR = "."+File.separatorChar+"datenbasis"+File.separatorChar;

    public HardwareeinbindungSimplePlugin() {
        super();
        File f = new File(STORAGEDIR);
        if (!f.exists()) {
            Err.dbg("Erstelle "+STORAGEDIR);
            f.mkdirs();
        }
    }
```

```
    protected String save(String storagename, InputStream in) {
        String result = null;
        try {
            File f = new File(STORAGEDIR+storagename);
            if (f.createNewFile()) {
                int i = 0;
                FileOutputStream out = new FileOutputStream(f);
                FileDescriptor fd = out.getFD();
                byte[] buffer = new byte[131072];
                while (i >= 0) {
                    out.write(buffer,0,i);
                    i = in.read(buffer);
                }
                out.flush();
                fd.sync();
                out.close();
                result = storagename;
            } else {
                Err.msg("Hardwareeinbindung: Dateiname wurde bereits vergeben: "+storagename);
            }
        }
        catch (IOException e) {
            Err.msg("Ein-/Ausgabefehler bei Hardwareeinbindung",e);
        }
        return result;
    }

    protected boolean load(String repertorienparameter, OutputStream out) {
        boolean result = false;
        try {
            File f = new File(STORAGEDIR+repertorienparameter);
            if (f.exists()) {
                int i = 0;
                FileInputStream in = new FileInputStream(f);
                byte[] buffer = new byte[131072];
                while (i >= 0) {
                    out.write(buffer,0,i);
                    i = in.read(buffer);
                }
                out.flush();
                result = true;
            } else {
                Err.msg("Hardwareeinbindung: Datei wurde nicht aufgefunden: "+repertorienparameter);
            }
        }
        catch (IOException e) {
            Err.msg("Ein-/Ausgabefehler bei Hardwareeinbindung (Lesen)",e);
        }
        finally {
        }
        return result;
    }
}
```

B.2 Die Beispielimplementierung einer Formatübersetzung

Dies ist die Klasse `FormatuebersetzungsPsToPdfPlugin`. Sie verwendet das Skript `/usr/bin/ps2pdf` für die Durchführung der Übersetzung von PostScript nach PDF.

```
package de.uni_bielefeld.rvs.plugins;

import java.util.Set;
import java.util.LinkedHashSet;
import de.uni_bielefeld.rvs.pluginsystem.FormatuebersetzungsPlugin;
import de.uni_bielefeld.rvs.util.Err;

public class FormatuebersetzungsPsToPdfPlugin extends FormatuebersetzungsPlugin {

    public FormatuebersetzungsPsToPdfPlugin() {
        super();
    }

    public boolean translate(String infilename, String outfilename) {
        try {
            Process p;
            Runtime rt = Runtime.getRuntime();
            String[] s = new String[3];
            s[0] = "/usr/bin/ps2pdf";
            s[1] = infilename;
            s[2] = outfilename;
            p = rt.exec(s);
            p.waitFor();
            return true;
        }
        catch (Exception e) {
            Err.msg("Fehler bei Formatübersetzung von Postscript nach PDF",e);
        }
        return false;
    }

    public Set quellFormate() {
        Set result = new LinkedHashSet();
        result.add("PostScript document text conforming at level 2.0");
        return result;
    }

    public Set zielFormate() {
        Set result = new LinkedHashSet();
        result.add("PDF document, version 1.2");
        return result;
    }
}
```

B.3 Die Beispielimplementierung einer Merkmalsextraktion

Die Beispielimplementation einer Merkmalsextraktion wird durch mehrere Klassen ausgeführt. Die Haupt- und Pluginklasse ist `MerkmalsextraktionsTexPlugin`. Sie verwendet drei Hilfsklassen zur Zustandsverwaltung beim Parsen der Eingabedatei. Die Implementation vermag aus einer Latex-Datei die dort vorhandenen Angaben zu Autor und Titel zu extrahieren.

```
package de.uni_bielefeld.rvs.plugins;

import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Map;
import java.util.Set;
import java.util.LinkedHashSet;
import de.uni_bielefeld.rvs.util.Err;
import de.uni_bielefeld.rvs.pluginsystem.MerkmalsextraktionsPlugin;

public class MerkmalsextraktionsTexPlugin extends MerkmalsextraktionsPlugin {

    private StateObserver so = new StateObserver();

    public MerkmalsextraktionsTexPlugin() {
        so.addState(0,0);
        so.add(0,'\\',1);
        so.addState(1,0);
        so.add(1,'a',10);
        so.addState(10,0);
        so.add(10,'u',11);
        so.addState(11,0);
        so.add(11,'t',12);
        so.addState(12,0);
        so.add(12,'h',13);
        so.addState(13,0);
        so.add(13,'o',14);
        so.addState(14,0);
        so.add(14,'r',15);
        so.addState(15,0);
        so.add(15,'{',16);
        so.addStateCollector(16,'{','}',"author");

        so.add(1,'t',20);
        so.addState(20,0);
        so.add(20,'i',21);
        so.addState(21,0);
        so.add(21,'t',22);
        so.addState(22,0);
        so.add(22,'l',23);
        so.addState(23,0);
        so.add(23,'e',24);
        so.addState(24,0);
        so.add(24,'{',25);
        so.addStateCollector(25,'{','}',"title");
    }
```

```
    public Map extract(InputStream in) {
        try {
            InputStreamReader sourceReader = new InputStreamReader(in);
            int i;
            while ((i = sourceReader.read()) >= 0) {
                so.parse((char) i);
            }
        }
        catch (Exception e) {
            Err.msg("Fehler bei Merkmalsextraktion",e);
        }
        return so.getEntries();
    }

    public Set quellFormate() {
        Set result = new LinkedHashSet();
        result.add("LaTeX 2e document text");
        return result;
    }
}
```

Die Hilfsklasse `State`:

```
package de.uni_bielefeld.rvs.plugins;

import java.util.*;
import java.lang.*;

public class State {

    protected int num = -1;
    private int defaultTo = -1;
    private LinkedHashMap to = new LinkedHashMap();

    public State(int num, int defaultTo) {
        this.num = num;
        this.defaultTo = defaultTo;
    }

    public int next(char c) {
        if (to.containsKey(new Character(c))) {
            return ((Integer) to.get(new Character(c))).intValue();
        } else {
            return defaultTo;
        }
    }

    public void add(char key, int state) {
        to.put(new Character(key), new Integer(state));
    }
}
```

Die Hilfsklasse `StateCollector`:

```
package de.uni_bielefeld.rvs.plugins;

public class StateCollector extends State {

    private int brackets = 0;
    private String key = "";
    private String collection = "";
    private char left;
    private char right;
```

```
    public StateCollector(int num, char left, char right, String key) {
        super(num,-num);
        this.left = left;
        this.right = right;
        this.key = key;
    }

    public void resetCollection() {
        collection = "";
    }

    public int next(char c) {
        if (c == left) {
            brackets++;
            collection += c;
            return num;
        } else {
            if (c == right) {
                if (brackets == 0) {
                    return super.next(c);
                } else {
                    brackets--;
                    collection += c;
                    return num;
                }
            } else {
                collection += c;
                return num;
            }
        }
    }
    public String getKey() {
        return key;
    }

    public String getCollection() {
        return collection;
    }
}
```

Die Hilfsklasse `StateObserver`:

```
package de.uni_bielefeld.rvs.plugins;

import java.util.*;

public class StateObserver {

    private LinkedHashMap states = new LinkedHashMap();
    private LinkedHashMap entries = new LinkedHashMap();
    private int state = 0;

    public void addState(int num, int defaultTo) {
        states.put(new Integer(num),new State(num,defaultTo));
    }

    public void addStateCollector(int num, char left, char right, String key) {
        states.put(new Integer(num),new StateCollector(num,left,right,key));
    }

    public void add(int num, char c, int next) {
        ((State) states.get(new Integer(num))).add(c,next);
    }
```

```
    public void parse(char c) {
        state = ((State) states.get(new Integer(state))).next(c);
        if (state < 0) {
            StateCollector col = ((StateCollector) states.get(new Integer(-state)));
            entries.put(col.getKey(),col.getCollection());
            col.resetCollection();
            state = 0;
        }
    }

    public Map getEntries() {
        return entries;
    }
}
```

Anhang C

Die CD-ROM

Dieser Diplomarbeit ist eine CD-ROM beigelegt. Es wird das ISO9660 Format mit Joliet- und RockRidge-Erweiterung verwendet. Die CD-ROM enthält

- die Diplomarbeit selbst im PostScript und PDF Format,
- den Quellcode des VDAS Systems,
- den Quellcode der Beispielimplementationen von Hardwareeinbindung, Formatübersetzung und Merkmalsextraktion und
- Beispieldaten zum Testen der Formatübersetzung und der Merkmalsextraktion.

Im Wurzelverzeichnis gibt es die Datei „`README.TXT`”, die eine genaue Auskunft über den Inhalt der CD-ROM enthält.